



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

© 2015 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.

Acknowledgments

This specification is the result of the contributions of many people. Here is a partial list of the contributors, including the companies that they represented at the time of their contribution.

- Michael Bedy, AMD
- Paul Blinzer, AMD
- Boleslaw Ciesielski, AMD
- Eric Finger, AMD
- Mark Fowler, AMD
- Mike Houston, AMD
- Nikolay Haustov, AMD
- Mark Herdeg, AMD
- Bill Licea-Kane, AMD
- Leonid Lobachev, AMD
- Mike Mantor, AMD
- Vicki Meagher, AMD
- Stanislav Mekhanoshin, AMD
- Dmitry Preobrazhensky, AMD
- Valery Pykhtin, AMD
- Chris Reeve, 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
- Konstantin Zhuravlyov, AMD
- Jem Davies, ARM
- Ian Devereux, ARM
- Robert Elliott, ARM
- Alexander Galazin, ARM
- Rune Holm, ARM
- Kurt Shuler, Arteris
- Andrew Richards, Codeplay
- Paul D'Arcy, General Processor Technologies
- 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
- Rahul Agarwal, MediaTek Inc.
- 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
- Lihan Bin, Qualcomm
- P.J. Bostley, Qualcomm
- Alex Bourd, Qualcomm
- Ken Dockser, Qualcomm

- Jamie Esliger, Qualcomm
- Ben Gaster, Qualcomm
- Andrew Gruber, Qualcomm
- Lee Howes, Qualcomm
- Wilson Kwan, Qualcomm
- Jack Liu, Qualcomm
- Bob Rychlik, Qualcomm
- Robert J. Simpson, Qualcomm
- Sumesh Udayakumaran, Qualcomm

- Ignacio Llamas, Samsung Electronics Co, Ltd
- Soojung Ryu, Samsung Electronics Co, Ltd

- Matthew Locke, Texas Instruments

- Chelsi Odegard, VTM Group

Contents

Acknowledgments	3
About the HSA Programmer's Reference Manual	19
Audience	19
Document Conventions	19
HSA Information Sources	19
Chapter 1. Overview	20
1.1 What Is HSAIL?	20
1.2 HSAIL Virtual Language	21
1.3 HSAIL Experimental Features	22
Chapter 2. HSAIL Programming Model	23
2.1 Overview of Grids, Work-Groups, and Work-Items	23
2.2 Work-Groups	25
2.2.1 Work-Group ID	25
2.2.2 Work-Group Flattened ID	26
2.3 Work-Items	26
2.3.1 Work-Item ID	26
2.3.2 Work-Item Flattened ID and Current Work-Item Flattened ID	27
2.3.3 Work-Item Absolute ID	27
2.3.4 Work-Item Flattened Absolute ID	27
2.4 Scalable Data-Parallel Computing	28
2.5 Active Work-Groups and Active Work-Items	28
2.6 Wavefronts, Lanes, and Wavefront Sizes	29
2.6.1 Example of Contents of a Wavefront	29
2.6.2 Wavefront Size	30
2.7 Types of Memory	30
2.8 Segments	31
2.8.1 Types of Segments	31
2.8.2 Shared Virtual Memory	35
2.8.3 Addressing for Segments	35
2.8.4 Memory Segment Access Rules	36
2.8.5 Memory Segment Isolation	39
2.9 Small and Large Machine Models	39
2.10 Base and Full Profiles	40
2.11 Race Conditions	40
2.12 Divergent Control Flow	41
2.12.1 Uniform Instructions	42
2.12.2 Using the Width Modifier with Control Transfer Instructions	44
2.12.3 (Post-)Dominator and Immediate (Post-)Dominator	45
Chapter 3. Examples of HSAIL Programs	46
3.1 Vector Add Translated to HSAIL	46
3.2 Transpose Translated to HSAIL	47
Chapter 4. HSAIL Syntax and Semantics	48
4.1 Two Formats	48

4.2 Program, Code Object, and Executable	48
4.2.1 Finalization	49
4.2.2 Loading	51
4.2.3 Execution	52
4.3 Module	53
4.3.1 Annotations	55
4.3.2 Kernel	56
4.3.3 Function	58
4.3.4 Signature	60
4.3.5 Code Block	61
4.3.6 Arg Block	62
4.3.7 Instruction	63
4.3.8 Variable	64
4.3.9 Fbarrier	68
4.3.10 Declaration and Definition Qualifiers	69
4.4 Source Text Format	74
4.5 Strings	76
4.6 Identifiers	77
4.6.1 Syntax	77
4.6.2 Scope	78
4.7 Registers	79
4.8 Constants	81
4.8.1 Integer Constants	82
4.8.2 Floating-Point Constants	83
4.8.3 Typed Constants	87
4.8.4 Aggregate Constants	91
4.8.5 How Text Format Constants Are Converted to Bit String Constants	92
4.9 Labels	94
4.10 Variable Initializers	94
4.11 Storage Duration	96
4.12 Linkage	97
4.12.1 Program Linkage	97
4.12.2 Module Linkage	98
4.12.3 Function Linkage	98
4.12.4 Arg Linkage	99
4.12.5 None Linkage	99
4.13 Data Types	99
4.13.1 Base Data Types	99
4.13.2 Packed Data Types	100
4.13.3 Opaque Data Types	101
4.13.4 Array Data Types	101
4.14 Packing Controls for Packed Data	101
4.14.1 Ranges	102
4.14.2 Packed Type Constants	103
4.15 Subword Sizes	104
4.16 Operands	104
4.16.1 Operand Compound Type	105
4.16.2 Rules for Operand Registers	105
4.17 Vector Operands	106
4.18 Address Expressions	106
4.19 Floating Point	107
4.19.1 Floating-Point Numbers	109
4.19.2 Floating-Point Rounding	109
4.19.3 Flush to Zero (ftz)	110
4.19.4 Not A Number (NaN)	111

4.19.5 Floating Point Exceptions	112
4.19.6 Unit of Least Precision (ULP)	112
4.20 Dynamic Group Memory Allocation	112
4.21 Kernarg Segment	114
Chapter 5. Arithmetic Instructions	116
5.1 Overview of Arithmetic Instructions	116
5.2 Integer Arithmetic Instructions	116
5.2.1 Syntax	116
5.2.2 Description	118
5.3 Integer Optimization Instruction	121
5.3.1 Syntax	121
Description	121
5.4 24-Bit Integer Optimization Instructions	122
5.4.1 Syntax	122
Description	122
5.5 Integer Shift Instructions	123
5.5.1 Syntax	123
5.5.2 Description for Standard Form	124
5.5.3 Description for Packed Form	124
5.6 Individual Bit Instructions	125
5.6.1 Syntax	125
Description	125
5.7 Bit String Instructions	127
5.7.1 Syntax	127
Description	127
5.8 Copy (Move) Instructions	130
5.8.1 Syntax	130
Description	131
5.8.2 Additional Information About Ida	132
5.9 Packed Data Instructions	133
5.9.1 Syntax	133
Description	134
5.9.2 Controls in src2 for shuffle Instruction	136
5.9.3 Common Uses for shuffle Instruction	137
5.9.4 Examples of unpacklo and unpackhi Instructions	139
5.10 Bit Conditional Move (cmov) Instruction	140
5.10.1 Syntax	140
Description	140
5.11 Floating-Point Arithmetic Instructions	140
5.11.1 Syntax	140
Description	142
5.12 Floating-Point Optimization Instruction	144
5.12.1 Syntax	144
Description	145
5.13 Floating-Point Bit Instructions	146
5.13.1 Syntax	146
Description	147
5.14 Native Floating-Point Instructions	148
5.14.1 Syntax	148
Description	149
5.15 Multimedia Instructions	150
5.15.1 Syntax	150
Description	150

5.16 Segment Checking (segmentp) Instruction	153
5.16.1 Syntax	153
Description	153
5.17 Segment Conversion Instructions	154
5.17.1 Syntax	154
Description	154
5.18 Compare (cmp) Instruction	155
5.18.1 Syntax	156
Description	157
5.19 Conversion (cvt) Instruction	159
5.19.1 Overview	159
5.19.2 Syntax	161
5.19.3 Rules for Rounding for Conversions	162
5.19.4 Description of Integer Rounding Modes	162
5.19.5 Description of Floating-Point Rounding Modes	164
Chapter 6. Memory Instructions	166
6.1 Memory and Addressing	166
6.1.1 How Addresses Are Formed	166
6.1.2 Memory Hierarchy	167
6.1.3 Alignment	168
6.1.4 Equivalence Classes	168
6.2 Memory Model	169
6.2.1 Memory Order	169
6.2.2 Memory Scope	170
6.2.3 Memory Synchronization Segments	170
6.2.4 Non-Memory Synchronization Segments	171
6.2.5 Agent Allocation	171
6.2.6 Course Grain Allocation	172
6.2.7 Kernel Dispatch Memory Synchronization	172
6.2.8 Execution Barrier	173
6.2.9 Flat Addresses	173
6.3 Load (ld) Instruction	173
6.3.1 Syntax	173
6.3.2 Description	174
6.3.3 Additional Information	176
6.4 Store (st) Instruction	177
6.4.1 Syntax	177
6.4.2 Description	178
6.4.3 Additional Information	179
6.5 Atomic Memory Instructions	180
6.6 Atomic (atomic) Instructions	181
6.6.1 Syntax	181
6.6.2 Description of Atomic and Atomic No Return Instructions	182
6.7 Atomic No Return (atomicnoret) Instructions	185
6.7.1 Syntax	185
6.7.2 Description	186
6.8 Notification (signal) Instructions	187
6.8.1 Syntax	188
6.8.2 Description of Signal Instructions	190
6.9 Memory Fence (memfence) Instruction	192
6.9.1 Syntax	192
6.9.2 Description	192

Chapter 7. Image Instructions	194
7.1 Images in HSAIL	194
7.1.1 Why Use Images?	194
7.1.2 Image Overview	195
7.1.3 Image Geometry	196
7.1.4 Image Format	198
7.1.4.1 Channel Order	198
7.1.4.1.1 x-Form Channel Orders	199
7.1.4.1.2 Standard RGB (s-Form) Channel Orders	200
7.1.4.2 Channel Type	200
7.1.4.3 Bits Per Pixel (bpp)	204
7.1.5 Image Access Permission	204
7.1.6 Image Coordinate	206
7.1.6.1 Coordinate Normalization Mode	206
7.1.6.2 Addressing Mode	207
7.1.6.3 Filter Mode	209
7.1.7 Image Creation and Image Handles	211
7.1.8 Sampler Creation and Sampler Handles	214
7.1.9 Using Image Instructions	216
7.1.10 Image Memory Model	218
7.2 Read Image (rdimage) Instruction	219
7.2.1 Syntax	219
Description	220
7.3 Load Image (ldimage) Instruction	221
7.3.1 Syntax	221
Description	222
7.4 Store Image (stimage) Instruction	222
7.4.1 Syntax	222
Description	223
7.5 Query Image and Query Sampler Instructions	224
7.5.1 Syntax	224
7.5.2 Description	224
7.6 Image Fence (imagefence) Instruction	225
7.6.1 Syntax	225
Description	226
Chapter 8. Branch Instructions	227
8.1 Syntax	227
8.2 Description	227
Chapter 9. Parallel Synchronization and Communication Instructions	229
9.1 Barrier Instructions	229
9.1.1 Syntax	229
9.1.2 Description	229
9.2 Fine-Grain Barrier (fbarrier) Instructions	230
9.2.1 Overview: What Is an Fbarrier?	230
9.2.2 Syntax	231
9.2.3 Description	231
9.2.4 Additional Information About Fbarrier Instructions	234
9.2.5 Pseudocode Examples	235
9.3 Execution Barrier	238
9.4 Cross-Lane Instructions	240
9.4.1 Syntax	240

9.4.2 Description	241
Chapter 10. Function Instructions	243
10.1 Functions in HSAIL	243
10.1.1 Example of a Simple Function	243
10.1.2 Example of a More Complex Function	243
10.1.3 Functions That Do Not Return a Result	244
10.2 Function Call Argument Passing	244
10.3 Function Declarations, Function Definitions, and Function Signatures	247
10.3.1 Function Declaration	247
10.3.2 Function Definition	247
10.3.3 Function Signature	248
10.4 Variadic Functions	248
10.5 align Qualifier	249
10.6 Direct Call (call) Instruction	250
10.6.1 Syntax	250
10.6.1 Description	250
10.7 Switch Call (scall) Instruction	251
10.7.1 Syntax	251
10.7.2 Description	251
10.8 Indirect Call (icall) Instruction	252
10.8.1 Syntax	253
10.8.2 Description	253
10.9 Return (ret) Instruction	254
10.9.1 Syntax	255
10.9.2 Description	255
10.10 Allocate Memory (alloca) Instruction	255
10.10.1 Syntax	255
10.10.2 Description	256
Chapter 11. Special Instructions	257
11.1 Kernel Dispatch Packet Instructions	257
11.1.1 Syntax	257
11.1.2 Description	258
11.2 Exception Instructions	260
11.2.1 Syntax	260
11.2.2 Description	261
11.2.3 Additional Information	261
11.3 User Mode Queue Instructions	262
11.3.1 Syntax	262
11.3.2 Description	263
11.4 Miscellaneous Instructions	264
11.4.1 Syntax	264
11.4.2 Description	265
Chapter 12. Exceptions	269
12.1 Kinds of Exceptions	269
12.2 Hardware Exceptions	269
12.3 Hardware Exception Policies	271
12.4 Debug Exceptions	272
12.5 Handling Signaled Exceptions	272
12.5.1 HSA Runtime Debug Interface Not Active	272
12.5.2 HSA Runtime Debug Interface Active	272

12.5.2.1 Sample Debug Interface	272
Chapter 13. Directives	274
13.1 extension Directive	274
13.1.1 extension CORE	274
13.1.2 extension IMAGE	274
13.1.3 How to Set Up Finalizer Extensions	275
13.2 loc Directive	276
13.3 pragma Directive	276
13.4 Control Directives for Low-Level Performance Tuning	278
Chapter 14. module Header	284
14.1 Syntax of the module Header	284
Chapter 15. Libraries	286
15.1 Library Restrictions	286
15.2 Library Example	286
Chapter 16. Profiles	288
16.1 What Are Profiles?	288
16.2 Profile-Specific Requirements	289
16.2.1 Base Profile Requirements	289
16.2.2 Full Profile Requirements	290
Chapter 17. Guidelines for Compiler Writers	292
17.1 Register Pressure	292
17.2 Using Lower-Precision Faster Instructions	292
17.3 Functions	292
17.4 Frequent Rounding Mode Changes	293
17.5 Wavefront Size	293
17.6 Control Flow Optimization	293
17.7 Memory Access	294
17.8 Unaligned Access	295
17.9 Constant Access	295
17.10 Segment Address Conversion	296
17.11 When to Use Flat Addressing	296
17.12 Arg Arguments	296
17.13 Exceptions	296
Chapter 18. BRIG: HSAIL Binary Format	298
18.1 What Is BRIG?	298
18.2 BRIG Module	299
18.3 Support Types	300
18.3.1 Section Offsets	300
18.3.2 BrigAlignment	301
18.3.3 BrigAllocation	301
18.3.4 BrigAluModifierMask	301
18.3.5 BrigAtomicOperation	302
18.3.6 BrigBase	302
18.3.7 BrigCompareOperation	302

18.3.8 BrigControlDirective	303
18.3.9 BrigExceptionsMask	303
18.3.10 BrigExecutableModifierMask	304
18.3.11 BrigImageChannelOrder	304
18.3.12 BrigImageChannelType	304
18.3.13 BrigImageGeometry	305
18.3.14 BrigImageQuery	305
18.3.15 BrigKind	305
18.3.16 BrigLinkage	306
18.3.17 BrigMachineModel	307
18.3.18 BrigMemoryModifierMask	307
18.3.19 BrigMemoryOrder	307
18.3.20 BrigMemoryScope	307
18.3.21 BrigModuleHeader	307
18.3.22 BrigOpcode	308
18.3.23 BrigPack	311
18.3.24 BrigProfile	311
18.3.25 BrigRegisterKind	311
18.3.26 BrigRound	311
18.3.27 BrigSamplerAddressing	312
18.3.28 BrigSamplerCoordNormalization	312
18.3.29 BrigSamplerFilter	313
18.3.30 BrigSamplerQuery	313
18.3.31 BrigSectionIndex	313
18.3.32 BrigSectionHeader	313
18.3.33 BrigSegCvtModifierMask	314
18.3.34 BrigSegment	314
18.3.35 BrigType	314
18.3.36 BrigUInt64	317
18.3.37 BrigVariableModifierMask	317
18.3.38 BrigVersion	318
18.3.39 BrigWidth	318
18.4 hsa_data Section	319
18.5 hsa_code Section	320
18.5.1 Directive Entries	320
18.5.1.1 Declarations and Definitions in the Same Module	321
18.5.1.2 BrigDirectiveArgBlock	321
18.5.1.3 BrigDirectiveComment	321
18.5.1.4 BrigDirectiveControl	322
18.5.1.5 BrigDirectiveExecutable	322
18.5.1.6 BrigDirectiveExtension	324
18.5.1.7 BrigDirectiveFbarrier	324
18.5.1.8 BrigDirectiveLabel	325
18.5.1.9 BrigDirectiveLoc	325
18.5.1.10 BrigDirectiveModule	326
18.5.1.11 BrigDirectiveNone	326
18.5.1.12 BrigDirectivePragma	327
18.5.1.13 BrigDirectiveVariable	327
18.5.2 Instruction Entries	329
18.5.2.1 BrigInstBase	329
18.5.2.2 BrigInstAddr	330
18.5.2.3 BrigInstAtomic	330
18.5.2.4 BrigInstBasic	331
18.5.2.5 BrigInstBr	331
18.5.2.6 BrigInstCmp	332

18.5.2.7 BrigInstCvt	332
18.5.2.8 BrigInstImage	333
18.5.2.9 BrigInstLane	333
18.5.2.10 BrigInstMem	334
18.5.2.11 BrigInstMemFence	335
18.5.2.12 BrigInstMod	335
18.5.2.13 BrigInstQueryImage	336
18.5.2.14 BrigInstQuerySampler	336
18.5.2.15 BrigInstQueue	337
18.5.2.16 BrigInstSeg	337
18.5.2.17 BrigInstSegCvt	338
18.5.2.18 BrigInstSignal	338
18.5.2.19 BrigInstSourceType	338
18.6 hsa_operand Section	339
18.6.1 Constant Operands	340
18.6.2 BrigOperandAddress	341
18.6.3 BrigOperandAlign	341
18.6.4 BrigOperandCodeList	342
18.6.5 BrigOperandCodeRef	342
18.6.6 BrigOperandConstantBytes	343
18.6.7 BrigOperandConstantImage	344
18.6.8 BrigOperandConstantOperandList	345
18.6.9 BrigOperandConstantSampler	346
18.6.10 BrigOperandOperandList	346
18.6.11 BrigOperandRegister	347
18.6.12 BrigOperandString	347
18.6.13 BrigOperandWavesize	347
18.7 BRIG Syntax for Instructions	348
18.7.1 BRIG Syntax for Arithmetic Instructions	348
18.7.1.1 BRIG Syntax for Integer Arithmetic Instructions	348
18.7.1.2 BRIG Syntax for Integer Optimization Instruction	349
18.7.1.3 BRIG Syntax for 24-Bit Integer Optimization Instructions	349
18.7.1.4 BRIG Syntax for Integer Shift Instructions	349
18.7.1.5 BRIG Syntax for Individual Bit Instructions	349
18.7.1.6 BRIG Syntax for Bit String Instructions	350
18.7.1.7 BRIG Syntax for Copy (Move) Instructions	350
18.7.1.8 BRIG Syntax for Packed Data Instructions	350
18.7.1.9 BRIG Syntax for Bit Conditional Move (cmov) Instruction	351
18.7.1.10 BRIG Syntax for Floating-Point Arithmetic Instructions	351
18.7.1.11 BRIG Syntax for Floating-Point Optimization Instruction	352
18.7.1.12 BRIG Syntax for Floating-Point Bit Instructions	352
18.7.1.13 BRIG Syntax for Native Floating-Point Instructions	353
18.7.1.14 BRIG Syntax for Multimedia Instructions	353
18.7.1.15 BRIG Syntax for Segment Checking (segmentp) Instruction	353
18.7.1.16 BRIG Syntax for Segment Conversion Instructions	354
18.7.1.17 BRIG Syntax for Compare (cmp) Instruction	354
18.7.1.18 BRIG Syntax for Conversion (cvt) Instruction	354
18.7.2 BRIG Syntax for Memory Instructions	354
18.7.3 BRIG Syntax for Image Instructions	355
18.7.4 BRIG Syntax for Branch Instructions	356
18.7.5 BRIG Syntax for Parallel Synchronization and Communication Instructions	356
18.7.6 BRIG Syntax for Function Instructions	357
18.7.7 BRIG Syntax for Special Instructions	358
18.7.7.1 BRIG Syntax for Kernel Dispatch Packet Instructions	358
18.7.7.2 BRIG Syntax for Exception Instructions	358

18.7.7.3 BRIG Syntax for User Mode Queue Instructions	358
18.7.7.4 BRIG Syntax for Miscellaneous Instructions	359
Chapter 19. HSAIL Grammar in Extended Backus-Naur Form	360
19.1 HSAIL Lexical Grammar in Extended Backus-Naur Form (EBNF)	360
19.2 HSAIL Syntax Grammar in Extended Backus-Naur Form (EBNF)	361
Appendix A. Limits	374
Appendix B. Glossary of HSAIL Terms	376
Index	383

Figures

Figure 2-1 A Grid and Its Work-Groups and Work-Items	23
Figure 2-2 TOKEN_WAVESIZE Syntax Diagram	30
Figure 4-1 HSA Runtime Support for HSAIL Life Cycle	49
Figure 4-2 module Syntax Diagram	54
Figure 4-3 moduleHeader Syntax Diagram	54
Figure 4-4 profile Syntax Diagram	54
Figure 4-5 machineModel Syntax Diagram	54
Figure 4-6 defaultFloatRounding Syntax Diagram	55
Figure 4-7 moduleDirective Syntax Diagram	55
Figure 4-8 moduleStatement Syntax Diagram	55
Figure 4-9 annotations Syntax Diagram	56
Figure 4-10 annotation Syntax Diagram	56
Figure 4-11 TOKEN_COMMENT Syntax Diagram	56
Figure 4-12 kernel Syntax Diagram	57
Figure 4-13 kernelHeader Syntax Diagram	57
Figure 4-14 kernFormalArgumentList Syntax Diagram	58
Figure 4-15 kernFormalArgument Syntax Diagram	58
Figure 4-16 function Syntax Diagram	59
Figure 4-17 functionHeader Syntax Diagram	59
Figure 4-18 funcOutputFormalArgumentList Syntax Diagram	59
Figure 4-19 funcInputFormalArgumentList Syntax Diagram	59
Figure 4-20 funcFormalArgumentList Syntax Diagram	60
Figure 4-21 funcFormalArgument Syntax Diagram	60
Figure 4-22 signature Syntax Diagram	60
Figure 4-23 sigOutputFormalArgumentList	60
Figure 4-24 sigInputFormalArgumentList Syntax Diagram	60
Figure 4-25 sigFormalArgumentList Syntax Diagram	61
Figure 4-26 sigFormalArgument Syntax Diagram	61
Figure 4-27 codeBlock Syntax Diagram	61
Figure 4-28 codeBlockDirective Syntax Diagram	61
Figure 4-29 codeBlockDefinition	62
Figure 4-30 codeBlockStatement Syntax Diagram	62
Figure 4-31 argBlock Syntax Diagram	62
Figure 4-32 argBlockDefinition	63
Figure 4-33 argBlockStatement Syntax Diagram	63
Figure 4-34 moduleVariable Syntax Diagram	65
Figure 4-35 codeBlockVariable Syntax Diagram	65
Figure 4-36 argBlockVariable Syntax Diagram	65

Figure 4-37 variable Syntax Diagram	66
Figure 4-38 variableSegment Syntax Diagram	66
Figure 4-39 dataTypeMod Syntax Diagram	67
Figure 4-40 optArrayDimension Syntax Diagram	67
Figure 4-41 moduleFbarrier Syntax Diagram	68
Figure 4-42 codeBlockFbarrier Syntax Diagram	69
Figure 4-43 fbarrier Syntax Diagram	69
Figure 4-44 optDeclQual Syntax Diagram	69
Figure 4-45 declQual Syntax Diagram	69
Figure 4-46 linkageQual Syntax Diagram	69
Figure 4-47 optAlignQual Syntax Diagram	70
Figure 4-48 optAllocQual Syntax Diagram	70
Figure 4-49 optConstQual Syntax Diagram	70
Figure 4-50 TOKEN_STRING Syntax Diagram	76
Figure 4-51 TOKEN_GLOBAL_IDENTIFIER Syntax Diagram	77
Figure 4-52 TOKEN_LOCAL_IDENTIFIER Syntax Diagram	77
Figure 4-53 TOKEN_LABEL_IDENTIFIER Syntax Diagram	77
Figure 4-54 identifier Syntax Diagram	77
Figure 4-55 TOKEN_CREGISTER Syntax Diagram	79
Figure 4-56 TOKEN_SREGISTER Syntax Diagram	79
Figure 4-57 TOKEN_DREGISTER Syntax Diagram	79
Figure 4-58 TOKEN_QREGISTER Syntax Diagram	79
Figure 4-59 registerNumber Syntax Diagram	80
Figure 4-60 initializerConstant Syntax Diagram	81
Figure 4-61 immediateOperand Syntax Diagram	81
Figure 4-62 TOKEN_INTEGER_CONSTANT Syntax Diagram	82
Figure 4-63 decimalIntegerConstant Syntax Diagram	82
Figure 4-64 hexIntegerConstant Syntax Diagram	82
Figure 4-65 octalIntegerConstant Syntax Diagram	82
Figure 4-66 integerConstant Syntax Diagram	83
Figure 4-67 TOKEN_HALF_CONSTANT Syntax Diagram	83
Figure 4-68 TOKEN_SINGLE_CONSTANT Syntax Diagram	84
Figure 4-69 TOKEN_DOUBLE_CONSTANT Syntax Diagram	84
Figure 4-70 decimalFloatConstant Syntax Diagram	84
Figure 4-71 hexFloatConstant Syntax Diagram	84
Figure 4-72 ieeeHalfConstant Syntax Diagram	84
Figure 4-73 ieeeSingleConstant Syntax Diagram	84
Figure 4-74 ieeeDoubleConstant Syntax Diagram	85
Figure 4-75 floatConstant Syntax Diagram	85
Figure 4-76 halfConstant Syntax Diagram	86
Figure 4-77 singleConstant Syntax Diagram	86
Figure 4-78 doubleConstant Syntax Diagram	86
Figure 4-79 typedConstant Syntax Diagram	87
Figure 4-80 integerTypedConstant Syntax Diagram	88
Figure 4-81 floatTypedConstant Syntax Diagram	88
Figure 4-82 signalTypedConstant Syntax Diagram	88
Figure 4-83 arrayTypedConstant Syntax Diagram	89
Figure 4-84 integerArrayTypedConstant Syntax Diagram	89
Figure 4-85 halfArrayTypedConstant Syntax Diagram	90
Figure 4-86 singleArrayTypedConstant Syntax Diagram	90
Figure 4-87 doubleArrayTypedConstant Syntax Diagram	90
Figure 4-88 packedArrayTypedConstant Syntax Diagram	91
Figure 4-89 imageArrayTypedConstant Syntax Diagram	91
Figure 4-90 samplerArrayTypedConstant Syntax Diagram	91
Figure 4-91 signalArrayTypedConstant Syntax Diagram	91

Figure 4-92 aggregateConstant Syntax Diagram	91
Figure 4-93 aggregateConstantItem Syntax Diagram	92
Figure 4-94 aggregateConstantAlign Syntax Diagram	92
Figure 4-95 optInitializer Syntax Diagram	94
Figure 4-96 packedTypeConstant Syntax Diagram	103
Figure 5-1 Example of Broadcast	138
Figure 5-2 Example of Rotate	138
Figure 5-3 Example of Unpack	139
Figure 6-1 Memory Hierarchy	168
Figure 13-1 pragma Syntax Diagram	276
Figure 13-2 pragmaOperand Syntax Diagram	277

Tables

Table 2-1 Wavefront 0 Through 6	29
Table 2-2 Memory Segment Access Rules	36
Table 2-3 Machine Model Data Sizes	40
Table 4-1 Text Constants and Results of the Conversion	92
Table 4-2 Base Data Types	99
Table 4-3 Packed Data Types and Possible Lengths	100
Table 4-4 Opaque Data Types	101
Table 4-5 Packing Controls for Instructions With One Source Input	101
Table 4-6 Packing Controls for Instructions With Two Source Inputs	102
Table 5-1 Syntax for Integer Arithmetic Instructions	117
Table 5-2 Syntax for Packed Versions of Integer Arithmetic Instructions	117
Table 5-3 Syntax for Integer Optimization Instruction	121
Table 5-4 Syntax for 24-Bit Integer Optimization Instructions	122
Table 5-5 Syntax for Integer Shift Instructions	123
Table 5-6 Syntax for Individual Bit Instructions	125
Table 5-7 Inputs and Results for popcount Instruction	126
Table 5-8 Syntax for Bit String Instructions	127
Table 5-9 Inputs and Results for firstbit and lastbit Instructions	130
Table 5-10 Syntax for Copy (Move) Instructions	131
Table 5-11 Syntax for Shuffle and Interleave Instructions	133
Table 5-12 Syntax for Pack and Unpack Instructions	134
Table 5-13 Bit Selectors for shuffle instruction	136
Table 5-14 Syntax for Bit Conditional Move (cmov) Instruction	140
Table 5-15 Syntax for Floating-Point Arithmetic Instructions	141
Table 5-16 Syntax for Packed Versions of Floating-Point Arithmetic Instructions	141
Table 5-17 Syntax for Floating-Point Optimization Instruction	144
Table 5-18 Class Instruction Source Operand Condition Bits	146
Table 5-19 Syntax for Packed Versions of Floating-Point Bit Instructions	147
Table 5-20 Syntax for Native Floating-Point Instructions	148
Table 5-21 Syntax for Multimedia Instructions	150
Table 5-22 Syntax for Segment Checking (segmentp) Instruction	153
Table 5-23 Syntax for Segment Conversion Instructions	154
Table 5-24 Syntax for Compare (cmp) Instruction	156
Table 5-25 Syntax for Packed Version of Compare (cmp) Instruction	156
Table 5-26 Floating-Point Comparisons	158
Table 5-27 Conversion Methods	160
Table 5-28 Notation for Conversion Methods	160
Table 5-29 Syntax for Conversion (cvt) Instruction	161
Table 5-30 Rules for Rounding for Conversions	162
Table 5-31 Integer Rounding Modes	164
Table 6-1 Syntax for Load (ld) Instruction	174

Table 6-2 Syntax for Store (st) Instruction	177
Table 6-3 Syntax for Atomic Instructions	181
Table 6-4 Syntax for Atomic No Return Instructions	185
Table 6-5 Syntax for Signal Instructions	189
Table 6-6 Syntax for memfence Instruction	192
Table 7-1 Image Geometry Properties	196
Table 7-2 Channel Order Properties	199
Table 7-3 Channel Type Properties	201
Table 7-4 Channel Order, Channel Type, and Image Geometry Combination	205
Table 7-5 Image Handle Properties	212
Table 7-6 Image Instruction Combinations	216
Table 7-7 Syntax for Read Image Instruction	219
Table 7-8 Syntax for Load Image Instruction	221
Table 7-9 Syntax for Store Image Instruction	222
Table 7-10 Syntax for Query Image and Query Sampler Instructions	224
Table 7-11 Explanation of imageProperty modifier	224
Table 7-12 Explanation of samplerProperty modifier	225
Table 7-13 Syntax for imagefence Instruction	225
Table 8-1 Syntax for Branch Instructions	227
Table 9-1 Syntax for Barrier Instructions	229
Table 9-2 Syntax for fbar Instructions	231
Table 9-3 Syntax for Cross-Lane Instructions	240
Table 10-1 Syntax for direct call Instruction	250
Table 10-2 Syntax for switch call Instruction	251
Table 10-3 Syntax for indirect call Instruction	253
Table 10-4 Syntax for ret Instruction	255
Table 10-5 Syntax for Allocate Memory (alloca) Instruction	255
Table 11-1 Syntax for Kernel Dispatch Packet Instructions	257
Table 11-2 Syntax for Exception Instructions	260
Table 11-3 Syntax for Exception Instructions	262
Table 11-4 Syntax for Miscellaneous Instructions	265
Table 13-1 Control Directives for Low-Level Performance Tuning	278
Table 18-1 Formats of Directives in the hsa_code Section	321
Table 18-2 Formats of Instructions in the hsa_code Section	329
Table 18-3 Formats of Operands in the hsa_operand Section	339
Table 18-4 BRIG Syntax for Integer Arithmetic Instructions	348
Table 18-5 BRIG Syntax for Integer Optimization Instruction	349
Table 18-6 BRIG Syntax for 24-Bit Integer Optimization Instructions	349
Table 18-7 BRIG Syntax for Integer Optimization Instructions	349
Table 18-8 BRIG Syntax for Individual Bit Instructions	349
Table 18-9 BRIG Syntax for Bit String Instructions	350
Table 18-10 BRIG Syntax for Copy (Move) Instructions	350
Table 18-11 BRIG Syntax for Packed Data Instructions	350
Table 18-12 BRIG Syntax for Bit Conditional Move (cmov) Instruction	351
Table 18-13 BRIG Syntax for Floating-Point Arithmetic Instructions	351
Table 18-14 BRIG Syntax for Floating-Point Optimization Instruction	352
Table 18-15 BRIG Syntax for Floating-Point Classify (class) Instructions	352
Table 18-16 BRIG Syntax for Native Floating-Point Instructions	353
Table 18-17 BRIG Syntax for Multimedia Instructions	353
Table 18-18 BRIG Syntax for Segment Checking (segmentp) Instruction	353
Table 18-19 BRIG Syntax for Segment Conversion Instructions	354
Table 18-20 BRIG Syntax for Compare (cmp) Instruction	354
Table 18-21 BRIG Syntax for Conversion (cvt) Instruction	354
Table 18-22 BRIG Syntax for Memory Instructions	354
Table 18-23 BRIG Syntax for Image Instructions	355

Table 18-24 BRIG Syntax for Branch Instructions	356
Table 18-25 BRIG Syntax for Parallel Synchronization and Communication Instructions	356
Table 18-26 BRIG Syntax for Instructions Related to Functions	357
Table 18-27 BRIG Syntax for Kernel Dispatch Packet Instructions	358
Table 18-28 BRIG Syntax for Exception Instructions	358
Table 18-29 BRIG Syntax for User Mode Queue Instructions	358
Table 18-30 BRIG Syntax for Miscellaneous Instructions	359

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 (page 376) . 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 Version 1.0* describes the HSA system architecture.
- *HSA Runtime Programmer's Reference Manual Version 1.0* describes the HSA runtime.
- *The OpenCL Specification Version 2.0* describes the OpenCL C language and runtime API: <http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf>
- *IEEE Standard for Floating-Point Arithmetic Version IEEE/ANSI Standard 754-2008* describes the IEEE/ANSI Standard 754-2008 floating-point standard.
- Jean-Michel Muller. *On the definition of $ulp(x)$* . RR-5504, 2005, pp.16.: <https://hal.inria.fr/inria-00070503>

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.
- An HSA runtime that is a library of services that supports the execution of HSAIL programs including a finalizer and a loader:
 - A *finalizer* translates HSAIL code into the appropriate native machine code if the hardware components cannot support HSAIL natively.
 - A *loader* loads native executable code onto hardware components.

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 kernel agent:

- 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 a kernel agent using memory instructions to construct and enqueue *Architected Queuing Language (AQL)* packets on *User Mode Queues* associated with the kernel agent. In some systems, a host CPU can also act as a kernel agent (with appropriate HSAIL finalizer and AQL mechanisms).
- A *kernel agent* is an agent that supports the HSAIL instruction set and includes a packet processor that supports the AQL packet format including the kernel dispatch packet. As an agent, a kernel agent can dispatch commands to any kernel agent (including itself) using memory instructions to construct and enqueue AQL packets on User Mode Queues associated with the kernel agent.

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

A kernel agent does not need to execute HSAIL code directly: it can execute machine code 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 kernel agents

Certain sections of code, called *kernels*, are executed in a data-parallel way by kernel agents. 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 \(page 39\)](#).

1.2 HSAIL Virtual Language

HSAIL is designed for parallel processing. The HSAIL virtual instruction set can be translated 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 \(page 288\)](#). HSAIL has no explicit parallel constructs; instead, each kernel contains instructions 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; the overhead of context switching among work-items is low.

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 \(page 46\)](#).

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

All mathematical instructions are register-to-register only. For example, to multiply two numbers, the values are loaded into registers and one of the multiply instructions (`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 kernel agents. Global memory (but not group memory or private memory) is coherent between all agents.

1.3 HSAIL Experimental Features

A few features in HSAIL are qualified as experimental. A future version of the specification may modify the feature in a non-backwards compatible way, may replace the feature with a different feature that serves similar goals, or may deprecate the feature completely. Experimental features are present as the functionality provided is considered potentially useful, although the exact form is not mature and still under development. A user should consider carefully whether to use these features as a future version of the specification may require changes to HSAIL source to continue executing correctly. Feedback on these features is solicited.

CHAPTER 2.

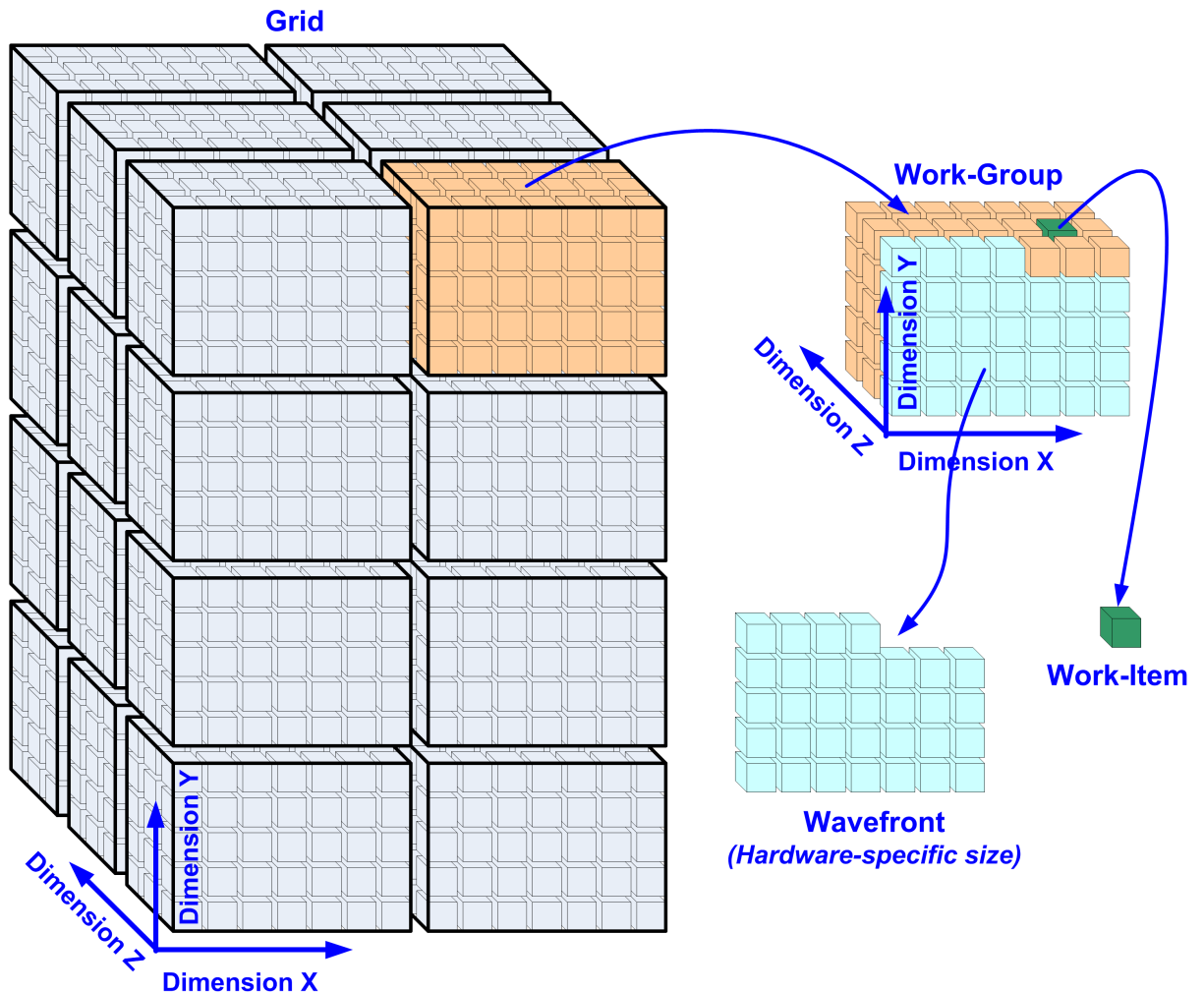
HSAIL Programming Model

This chapter describes the HSAIL programming model.

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

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

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



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

The kernel language runtime can be used to invoke the kernel language compiler that will produce HSAIL. The HSA runtime can then be used by the language runtime to execute the finalizer for the kernel agent that will execute the kernel. The finalizer takes the HSAIL represented in the binary BRIG format and produces an HSA code object that contains the kernel native machine code that will execute on that kernel agent. The finalizer can either be executed “online” as part of the application that will execute the kernel, or as part of an “offline” tool that saves the HSA code objects for later execution by other applications.

If the HSAIL requires more resources than are available on the kernel agent, the finalizer will return a failure result. For example, the kernel might require more group memory, or more fbarriers than are available on the kernel agent.

The kernel language runtime can use the HSA runtime loader to load HSA code objects onto kernel agents that have a matching native instruction set architecture. The loader can be used to obtain the information required to create AQL kernel dispatch packets used to execute the kernels contained in the loaded HSA code objects.

A kernel agent can have multiple User Mode Queues associated with it. Each User Mode Queue has a *queue ID*, which is unique across all the User Mode Queues created by the process executing the application.

A request to execute a kernel is made by appending an AQL kernel dispatch packet on a User Mode Queue associated with a kernel agent. Each AQL packet is assigned a *packet ID* that is unique for each User Mode Queue.

An HSA implementation ensures that all User Mode Queues are serviced and dispatches the kernel executable code associated with the queued kernel dispatch packets on the kernel agent with which the User Mode Queue is associated, causing the kernel to be executed.

If the kernel agent has insufficient resources to execute at least one work-group of a kernel dispatch, then the dispatch fails, and the HSA runtime transitions the User Mode Queue into the error state. No kernel execution occurs, and the kernel dispatch packet completion signal is not updated. For example, the dispatch might request more dynamic group memory than is available. A dispatch may, but is not required to, fail if the dispatch arguments are not compatible with any control directives specified when the kernel was finalized. For example, the dispatch work-group size might not match the values specified by a `requiredworkgroupsize` control directive.

The combination of the packet ID and the queue ID can be used to identify a kernel dispatch within the application. A kernel can access these IDs by means of the `packetid` special instruction and by using memory instructions to access the `id` field of the User Mode Queue memory structure. See [11.1. Kernel Dispatch Packet Instructions \(page 257\)](#) and the *HSA Platform System Architecture Specification Version 1.0* section 2.8 *Requirement: User Mode Queuing*.

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 \(page 26\)](#).

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 \(facing page\)](#).

A work-group is an instance of execution on the kernel agent. Execution is performed by a compute unit. A kernel agent 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 \(page 29\)](#).

The size of each grid dimension is not required to be an integral multiple of the size of the corresponding work-group dimension, 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 a kernel agent 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 \(page 29\)](#).

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 instructions 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 \(page 27\)](#).

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

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

Each work-group can access assorted predefined read-only values such as work-group ID, work-group size, and so forth through the use of dispatch packet instructions. See [11.1. Kernel Dispatch Packet Instructions \(page 257\)](#).

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

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

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

$$\text{work-group flattened ID} = \text{workgroupid}_0 + \text{workgroupid}_1 * \text{workgroupsize}_0 + \text{workgroupid}_2 * \text{workgroupsize}_0 * \text{workgroupsize}_1$$

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 instructions. See [Chapter 11. Special Instructions \(page 257\)](#).

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 \(page 31\)](#).

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 $[0, 1, 2, \dots, \text{max}_i - 1]$.

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

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

2.3.2 Work-Item Flattened ID and Current 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:

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

where:

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

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

Note that the set of values produced by work-item flattened ID for each work-item of a partial work-group (see 2.1. Overview of Grids, Work-Groups, and Work-Items (page 23)) is not dense since it is computed using `workgroupsize`, which applies only to non-partial work-groups.

However, the work-item ID can also be flattened into one dimension using `currentworkgroupsize`.

The current work-item flattened ID is defined as:

$$\text{current work-item flattened ID} = ID_0 + ID_1 * \text{current_max}_0 + ID_2 * \text{current_max}_0 * \text{current_max}_1$$

where:

ID_0 = workitemid (dimension 0)
 ID_1 = workitemid (dimension 1)
 ID_2 = workitemid (dimension 2)
 current_max_0 = currentworkgroupsize (dimension 0)
 current_max_1 = currentworkgroupsize (dimension 1)

Note that the set of values produced by current work-item flattened ID for each work-item of a work-group is always dense, even when it is a partial work-group.

The current work-item flattened ID can be accessed by means of the special instruction `currentworkitemflatid`. The value returned by this instruction will only be different from that returned by the `workitemflatid` instruction if the current work-item belongs to a partial work-group.

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 $[0, 1, 2, \dots, \max_i - 1]$.

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

The work-item absolute ID can be accessed by means of the special instruction `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:

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

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 instruction `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 instruction is one that executes the current instruction. For example:

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

The active work-items at this *instruction* 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 instruction 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 in a work-group are assigned to wavefronts consecutively in current work-item flattened ID order. This can be useful to expert programmers. See [2.3.2. Work-Item Flattened ID and Current Work-Item Flattened ID \(page 27\)](#)).

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 256 inclusive. Thus, a wavefront with a wavefront size of 64 has 64 lanes.

A lane has an identifier unique within the wavefront which can be accessed by means of the `laneid` instruction which is defined as:

```
current work-item flattened ID % wavefront size
```

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

Note that partial work-groups may have fewer wavefronts than non-partial work-groups. See [2.1. Overview of Grids, Work-Groups, and Work-Items \(page 23\)](#).

Two work-items in the same work-group will be in the same wavefront if the floor of `current 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 2-1 Wavefront 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

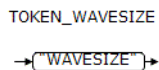
Wavefront 5						
Dimensions X, Y, Z	8-12, 0, 8	0-12, 1, 8	0-12, 2, 8	0-12, 0, 9	0-12, 1, 9	0-6, 2, 9
Work-Item Absolute Flattened IDs	320-324	325-337	338-350	351-363	364-376	377-383
Lane IDs	0-4	5-17	18-30	31-43	44-56	57-63

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 2-2 TOKEN_WAVESIZE Syntax Diagram



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 instruction operand where an integer, bit, or packed immediate value less than or equal to 64 bits 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 \(page 41\)](#).

`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 \(below\)](#).

- Registers

There are four register sizes:

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

Registers are untyped.

For more information, see [4.7. Registers \(page 79\)](#).

- 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 Instructions \(page 194\)](#).

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 kernel agents 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 instructions generate addresses that are outside the bounds of a segment.

No isolation guarantee between segments is provided. See [2.8.5. Memory Segment Isolation \(page 39\)](#).

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 [4.3.10. Declaration and Definition Qualifiers \(page 69\)](#).

- Global memory variables with program allocation have a single allocation for the variable which is visible to all agents, including all kernel agents 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 kernel agent.
- Global memory variables with agent allocation have multiple allocations, one for each kernel agent on which code is loaded that accesses the variable. Each allocation has a distinct global segment address and is only visible to the associated kernel agent. 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 kernel agent. In addition, the host CPU agent can access all allocations by using the HSA runtime and specifying the kernel agent.

The visibility of global memory is further constrained by the memory model (see [6.2. Memory Model \(page 169\)](#)). For a description of the visibility of variable initializers, see [4.10. Variable Initializers \(page 94\)](#).

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 \(page 69\)](#).

Standard page protections (for example, read-only, read-write, and protected) apply to global memory. See the *HSA Platform System Architecture Specification Version 1.0* section 2.1 *Requirement: Shared Virtual Memory*.

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 \(page 169\)](#).

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 \(page 35\)](#).

- 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 \(page 35\)](#).

- 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.

Kernel agents 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 kernel agent on which code is loaded that accesses the variable, each allocation with a distinct address. Each kernel agent can only access its associated allocation. The host CPU agent can access all allocations by using the HSA runtime and specifying the kernel agent. See the `alloc` qualifier description in section [4.3.10. Declaration and Definition Qualifiers \(page 69\)](#).

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, the behavior is undefined if a readonly variable allocation value for a kernel agent is changed while a kernel dispatch that uses that variable is executing on that kernel agent. See [4.10. Variable Initializers \(page 94\)](#).

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 \(page 69\)](#).

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 hardware 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` instruction. These restrictions make it easier for a finalizer to promote spill segment variables to hardware 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 hardware 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` instruction. This makes it easier for a finalizer to allocate arg segment variables to hardware 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 \(page 244\)](#).

Also see:

- [4.6.2. Scope \(page 78\)](#)
- [4.11. Storage Duration \(page 96\)](#)
- [4.3.10. Declaration and Definition Qualifiers \(page 69\)](#)

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 each other. 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 instructions 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.

The address of a location in the global segment is the same value as a flat address to the same global segment location. In addition, the same value is used for the null pointer value in both the global segment and in a flat address. Therefore, no conversion is required to or from a flat address that references the global segment and a global segment address.

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

For more information, see [6.1. Memory and Addressing \(page 166\)](#).

See also:

- [5.16. Segment Checking \(segmentp\) Instruction \(page 153\)](#)
- [5.17. Segment Conversion Instructions \(page 154\)](#)

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 2–2 \(below\)](#).

Table 2–2 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

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?
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
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> instruction.	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> instruction.	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 kernel agent, can be seen by any kernel dispatch executing on the same kernel agent in the same application execution. Note, the readonly segment variables for a kernel agent cannot be changed while a kernel dispatch that accesses the variables is executing on that kernel agent.
- **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 \(page 78\)](#) and [4.11. Storage Duration \(page 96\)](#).

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, the results are defined only if the flat address is converted back to a segment address of the original segment kind. This allows a `segmenttp` instruction 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 [5.16. Segment Checking \(segmenttp\) Instruction \(page 153\)](#).

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.

The results of converting a flat address to a segment address is defined only if the value accessed by the flat address is defined. For example, the results are not defined if a private segment address is converted into a flat address in one work-item, and then converted 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 \(page 69\)](#).
- Initializers, see [4.10. Variable Initializers \(page 94\)](#).

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:

- even if the variable is defined explicitly in HSAIL or is allocated dynamically by any agent including a host CPU;
- even if the variable is accessed using a segment address or a corresponding flat address; and
- even if 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 hardware registers.
 - Group addresses could be mapped to special scratch-pad memory allocated for each kernel agent 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 2–3 \(next page\)](#) shows the sizes used for the two models supported by HSAIL.

The machine model of the HSAIL code executed by a kernel agent must match the address space size of the process that owns the User Mode 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 `module` header. See [14.1. Syntax of the module Header \(page 284\)](#).

Table 2–3 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
Kernel code handle	64-bit	64-bit
Indirect function code handle	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 `module` header. See [Chapter 14. module Header \(page 284\)](#).

For more information, see [Chapter 16. Profiles \(page 288\)](#).

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 \(page 169\)](#).

2.12 Divergent Control Flow

On kernel agents with a wavefront size greater than 1, control flow instructions 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 Instructions \(page 227\)](#)), or a function call that can invoke multiple functions (namely a switch call `scall` or indirect call `icall`, see [Chapter 10. Function Instructions \(page 243\)](#)), 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` instruction will transfer control to the label for work-items where the source condition is true and to the instruction after the `cbr` for work-items where the source condition is false. Similarly, a `sbr` or `scall` instruction might transfer to different labels or functions respectively for work-items which have different values for the source index. Finally, an `icall` instruction 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 involving communication operations no later than the immediate post-dominator (see 2.12.3. (Post-)Dominator and Immediate (Post-)Dominator (page 45)). Communication operations comprise:

- atomic memory (see 6.5. Atomic Memory Instructions (page 180))
- `memfence` (see 6.9. Memory Fence (`memfence`) Instruction (page 192))
- signals (see 6.8. Notification (signal) Instructions (page 187))
- `imagefence` (see 7.6. Image Fence (`imagefence`) Instruction (page 225))
- cross-lane (see 9.4. Cross-Lane Instructions (page 240))
- `barrier` and `wavebarrier` (see 9.1. Barrier Instructions (page 229))
- `fbarriers` (see 9.2. Fine-Grain Barrier (`fbarrier`) Instructions (page 230))
- `clock` (see 11.4. Miscellaneous Instructions (page 264))
- `cleardetectexcept`, `getdetectexcept` and `setdetectexcept` (see 11.2. Exception Instructions (page 260))
- or calls to functions that contain any of these (see Chapter 10. Function Instructions (page 243))

In addition, control flow involving cross-lane instructions (see 9.4. Cross-Lane Instructions (page 240)) must diverge no later than the immediate dominator and reconverge no earlier than the immediate post-dominator.

These requirements can limit certain optimizations that involve code hoisting and cloning control flow (see 17.6. Control Flow Optimization (page 293)). 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 (page 169).

2.12.1 Uniform Instructions

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 instruction behaves the same for each work-item in the slice each time it is evaluated for a particular evaluation property, then the instruction is termed a *uniform instruction* with respect to the slice and evaluation property. Note that the instruction 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 instruction is evaluated. The instruction only has to behave the same for the work-items in a single slice for a single evaluation of the instruction.

Certain HSAIL memory, image, control flow, function and parallel synchronization, and communication instructions allow the uniformity of the operation to be specified by an optional width modifier. These instructions specify the evaluation property and default slice algorithm that will be used if the width modifier is omitted. In addition, some special instructions 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 instruction 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 instruction and so do not use the result of the instruction even if it is result uniform.

For example, a load instruction 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 instruction 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 instructions 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 Instructions \(next page\)](#).

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 instruction 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 instruction is executed.

For example, a cross-lane instruction is execution uniform if all work-items in the slice will execute it. This may allow a finalizer to use special machine 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 instructions (to both the global and group segments), memory fences, signal instructions, the `clock` instruction, cross-lane instructions, the DETECT exception special instructions, and the execution synchronization instructions (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 wavefront size that is greater than or equal to n , it is free to optimize the code 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 and Current Work-Item Flattened ID \(page 27\)](#)) 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 \(page 29\)](#)).

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 \(page 29\)](#).

If the number of work-items in a work-group is not a multiple of `WAVESIZE`, then the last wavefront of the work-group is termed a *partial wavefront*. Any lanes in a partial wavefront that do not correspond to work-items of the work-group are termed *partial lanes*, and are treated as inactive. For execution uniform, partial lanes are ignored, and only the non-partial lanes have to all be active or all be inactive.

If the slice size is larger than the work-group size, then it is treated the same as if `width (all)` was specified.

The default for the width modifier if it is omitted depends on the instruction, 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 Instructions

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 instructions, 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 kernel agent can be obtained by an HSA runtime query. Using `width(WAVESIZE)` may allow the finalizer to optimize.

2.12.3 (Post-)Dominator and Immediate (Post-)Dominator

The dominator of an instruction o is defined as a point p in the program such that every path from the start of the function or kernel that reaches o must go through p . No matter which path is taken from the start of the function or kernel to reach o , control will always pass through p . The immediate dominator is the unique point that does not dominate any other dominator of o .

The post-dominator of a branch instruction b is defined as a point p in the program such that every path from the instruction 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 of HSAIL instructions 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:

```
module &VectorAdd:1:0:$full:$small:$default;

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:
    ret;                                     // %if.end
@BB0_2:
    shl_u32           $s1, $s1, 2;          // %if.then
    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.

```

module &Transpose:1:0:$full:$small:$default;

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

```

CHAPTER 4. HSAIL Syntax and Semantics

This chapter describes the HSAIL syntax and semantics.

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 instructions show syntax for the text format.

For more information about BRIG, see [Chapter 18. BRIG: HSAIL Binary Format \(page 298\)](#).

The HSA runtime finalizer operates on the BRIG format.

4.2 Program, Code Object, and Executable

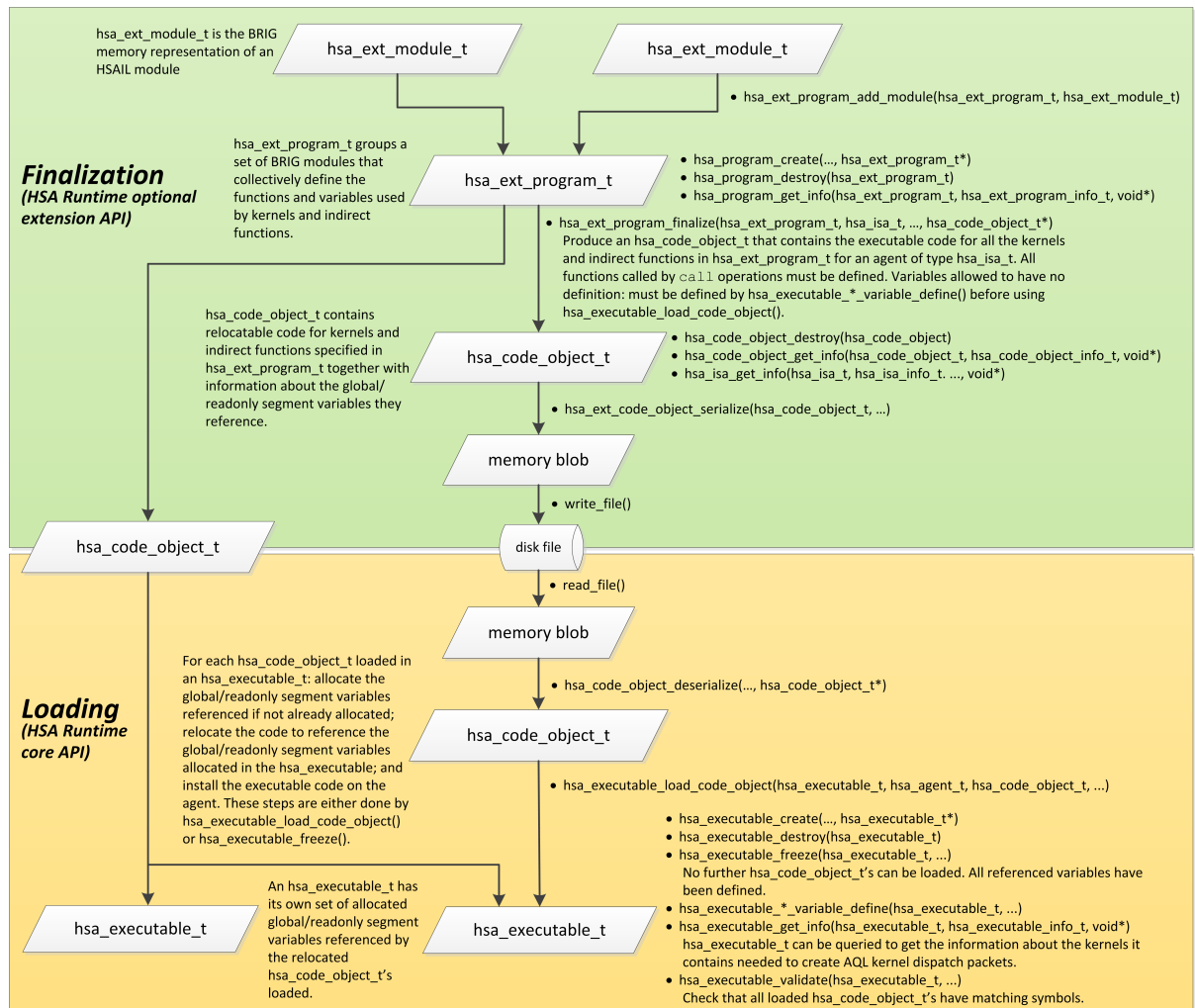
An application can use the HSA runtime to generate code from HSAIL that can be executed on kernel agents. The life cycle is split into three stages:

- Finalization: Creates code for a specific kernel agent instruction set architecture. (See [4.2.1. Finalization \(facing page\)](#).)
- Loading: Manages the allocation of global and readonly segment variables and installing of the finalized code onto specific kernel agents. (See [4.2.2. Loading \(page 51\)](#).)
- Execution: Creates dispatch packets that are executed on a kernel agent. (See [4.2.3. Execution \(page 52\)](#).)

The HSA runtime objects and operations on them that support the first two stages are illustrated in [Figure 4-1 \(facing page\)](#).

Finalization and loading can be performed within the same application, or can be done by independent applications by using the HSA runtime serialize and deserialize operations to save and restore the finalized code. This supports both online and offline finalization paths, and also provides the ability to implement application install time finalization and persistent disk caching to reduce online finalization.

Figure 4–1 HSA Runtime Support for HSAIL Life Cycle



4.2.1 Finalization

An application can use the HSA runtime to create zero or more HSAIL programs, to which it can add zero or more HSAIL modules.

When an HSAIL program is created the machine model (see 2.9. [Small and Large Machine Models](#) (page 39)), profile (see Chapter 16. [Profiles](#) (page 288)), and default floating-point rounding mode (see 4.19.2. [Floating-Point Rounding](#) (page 109)) must be specified. All HSAIL modules have a module header (see Chapter 14. [module Header](#) (page 284)) that specifies the HSAIL version, machine model, profile, and default floating-point rounding mode of the module. All HSAIL modules added to the program must:

- Must have an HSAIL version that the HSA runtime supports.
- Must have the same machine model as the HSAIL program.
- Must have the same profile as the HSAIL program.
- Must have either the default floating-point rounding mode or the same default floating-point rounding mode as the HSAIL program.

The HSAIL modules added to a program must not be destroyed until the program is destroyed.

The HSAIL module is the unit of HSAIL generation, and can contain multiple symbol declarations and definitions. A module can be added to zero or more programs. A module has a name (see [Chapter 14. module Header \(page 284\)](#)). Every module added to a program must have a unique name. Linking of symbol declarations to symbol definitions between modules is done within the context of the HSAIL program (see [4.12. Linkage \(page 97\)](#)).

The HSA runtime finalizer can be used to generate an HSA code object that contains the code for all the kernels and indirect functions defined in the modules added to a specific program for a specific instruction set architecture. All variables, barriers, and functions must be defined amongst the modules that have been added to the program if they are referenced by operations in the code block of:

- The kernels and indirect functions defined in the modules added to the program.
- The transitive closure of all functions specified by `call` or `scall` instructions starting with the kernels and indirect functions defined in the modules added to the program. See [Chapter 10. Function Instructions \(page 243\)](#).

The exception is that global and readonly segment variables with program linkage do not have to be defined. For example, this allows a host CPU allocated variable to act as the definition of an HSAIL variable.

It is allowed for a kernels and unreferenced indirect functions to have no definition in a program being finalized. Such kernels and indirect functions will not be part of the generated code object.

If the program specifies the default floating-point rounding mode as `default`, then the finalizer will use the default floating-point rounding mode of the kernel agent for which it is generating code. If the program specifies the default floating-point rounding mode as `zero` or `near` then finalizer will report an error if the kernel agent does not support that default floating-point rounding mode. Otherwise, the finalizer will use the default floating-point rounding mode of the program.

Note that:

- The finalizer uses the default floating-point rounding mode of the program, not that specified by the modules added to the program. In particular, if a module specifies a default floating-point rounding mode of `default` and the program it is added to has a default floating-point rounding mode of `zero` or `near`, then the module will behave as if its module header was specified with the same floating-point rounding mode as the program
- The code object produced by the finalizer always has a default floating-point rounding mode of `zero` or `near`, even if the default floating-point rounding mode of the program being finalized is `default`.
- In the case of a program with a default floating-point rounding mode of `default`, it is not until the program is finalized for a specific kernel agent that the actual floating-point rounding mode used as the default, by that kernel agent, can be determined and used in the generation of the code object.

Each instruction set architecture can support one or more call conventions. For example, different call conventions may use a different number of hardware registers to allow different numbers of wavefronts to execute on a compute unit. An HSA runtime query is available to determine the number of call conventions supported by an instruction set architecture, and to determine properties of each call convention such as maximum number of wavefronts per compute unit. When finalizing, the specific call convention required can be specified, or the finalizer can be requested to choose the best call convention based on the kernels. The call convention used is recorded in the code object produced by the finalizer.

An HSA program can be finalized for the same instruction set architecture with different call conventions.

Once the HSA code objects for the required instruction set architectures have been created, the HSA runtime can be used to destroy the HSAIL program. The code objects are independent of the program and modules.

The HSA runtime can be used to serialize an HSA code object to a memory blob so it can be saved to disk. This could be used to support offline finalization, or caching of finalized results.

4.2.2 Loading

A code object can either be created directly using the HSA runtime finalizer, or by deserializing a previously serialized code object.

To execute code in a code object it must be loaded. An application can use the HSA runtime to create zero or more HSA executables to which it can load zero or more code objects into specified kernel agents.

When an HSA executable is created, the profile must be specified.

The instruction set architecture of the code object must be compatible with the instruction set architecture of the kernel agent.

The version of the code object must be supported by the HSA runtime.

The machine model of the code object must match the address size used by the application (see [Table 2-3 \(page 40\)](#)).

The profile of the code object must match the profile of the executable.

The default floating-point rounding mode of the code object (which is either `zero` or `near`) must be supported by the kernel agent. Note that an executable may contain code objects that use different default floating-point rounding modes as different kernel agents may have different default floating-point rounding modes. Also note that the default floating-point rounding mode of the code object does not have to be the same as the default floating-point rounding mode of the kernel agent, it just has to be one of the floating-point rounding modes that the kernel agent supports.

Multiple code objects can be loaded into an executable for different kernel agents. The same code object can be loaded to multiple kernel agents that have the same instruction set architecture.

All code loaded into a single executable must have been finalized from the same program. Two programs are considered the same if they have the same modules added in the same order.

Once a code object has been loaded into an executable, it can be destroyed. The code object and executable are independent.

When all code objects have been added, the HSA runtime must be used to freeze the executable. Once frozen no further code objects can be loaded.

The executable manages allocating the global and readonly segment variables referenced by the code objects that are defined in the program according to the linkage (see [4.12. Linkage \(page 97\)](#)) and allocation (see [4.3.10. Declaration and Definition Qualifiers \(page 69\)](#) and [6.2.5. Agent Allocation \(page 171\)](#)) of the variable. For example:

- If multiple code objects are loaded in the same executable that reference a program allocation global segment variable, then they will share a single allocation.
- If multiple code objects are loaded in the same executable for different agents that reference an agent allocation global segment variable or readonly segment variable, then each agent will have a distinct allocation.

If the same code object is added to multiple executables, each executable will have its own distinct allocations.

In addition, the application can provide external global and readonly segment variable definitions to an executable for variables not defined by the program, and can obtain the address of global and readonly segment variables allocated by the executable, using the HSA runtime.

Once an executable is frozen:

- all global and readonly segment variables defined by the program and referenced by the loaded code objects have been allocated;
- the loaded code has been relocated to reference the variable allocations and external definitions;
- the code has been installed in the kernel agents, copying to agent local memory if necessary;
- and any instruction caches have been flushed.

At this point, the code of the executable is available to be executed on the kernel agents on which it has been loaded.

4.2.3 Execution

Once a code object has been loaded into an executable for a kernel agent, and the executable has been frozen, kernels in it can be executed by adding a kernel dispatch packet to a User Mode Queue associated with the kernel agent. The information reported by the finalizer that is required to create the kernel dispatch packet can be obtained using HSA runtime queries on the HSA executable or HSA code object for the specific kernel. This information includes:

- 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 instructions or fbarriers.

This does not include any dynamically allocated group segment space (see [4.20. Dynamic Group Memory Allocation \(page 112\)](#)).

- 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 hardware registers due to calls.
 - Any other finalizer introduced temporaries including spilled hardware registers and space for function call stack if statistically known.

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

If the kernel uses `alloca`, calls indirect functions using `icall` or has recursive function calls, then the finalizer may report that a dynamically sized call stack is required. The private segment size does not include the size of the dynamically sized call stack, only the size of the statically known private segment objects.

If the finalizer reports that a dynamic call stack is used, then the private segment size used in the dispatch packet must have the size of the call stack added to the reported static private segment size.

- The kernel code handle for the finalized code that includes the executable code for the kernel agent. It can be used for the kernel dispatch packet kernel object address field. A kernel code handle is an opaque 64-bit value for small and large machine model (see [2.9. Small and Large Machine Models \(page 39\)](#)).

Other information that may be useful to a high-level language runtime to invoke and manage the kernel's execution can also be queried. For example, the size and alignment of the kernarg segment and the call convention used by the code of the kernel.

Once a kernel dispatch packet has been added to the User Mode Queue, the kernel agent's packet processor will initiate execution of the kernel dispatch when it processes the packet.

HSA runtime queries on an HSA executable can also be used to obtain an indirect function code handle for an indirect function in a code object loaded on a kernel agent for a specific call convention. An indirect function code handle is an opaque 32-bit value in small machine model, and 64-bit value in large machine model (see [2.9. Small and Large Machine Models \(page 39\)](#)).

The application can pass indirect function code handles into kernel dispatches, or store them into global memory, for example, to use as a virtual function tables. A kernel can use them to call the indirect functions using the `icall` instruction (see [10.8. Indirect Call \(icall\) Instruction \(page 252\)](#)). The `icall` instruction is not supported by the Base profile (see [16.2.1. Base Profile Requirements \(page 289\)](#)).

The code for indirect functions is only made available to kernel dispatches launched after the indirect function has been loaded and the executable frozen. Therefore, prior to executing a kernel, all indirect functions that it will call must have been loaded for the kernel agent with the call convention used by the kernel code.

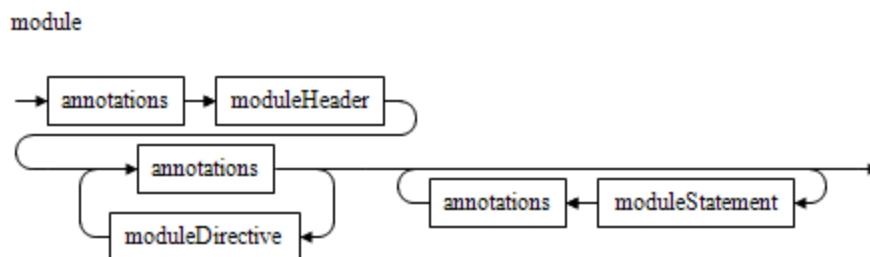
The code for kernels and indirect functions will remain available to execute until the HSA runtime is used to destroy the HSA executable in which the code is loaded. All HSA executables created by the application are implicitly destroyed when the application terminates.

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 module header, is followed by zero or more module directives, and ends with zero or more module statements.

Figure 4–2 module Syntax Diagram



The `module` header specifies the module name, HSAIL language version and the required profile, machine model, and default floating-point rounding mode. For more information, see [Chapter 14. module Header](#) (page 284).

Figure 4–3 moduleHeader Syntax Diagram

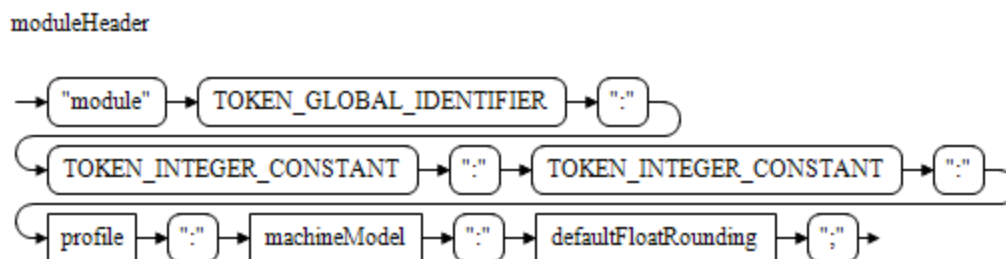


Figure 4–4 profile Syntax Diagram

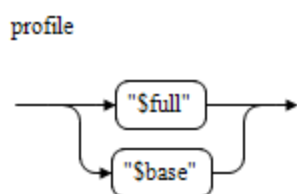


Figure 4–5 machineModel Syntax Diagram

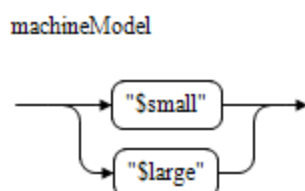
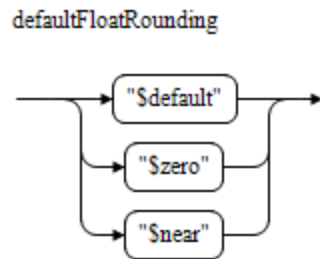
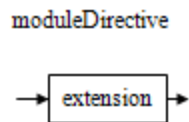


Figure 4–6 defaultFloatRounding Syntax Diagram



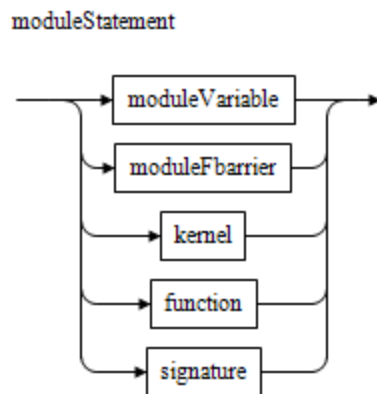
A module directive can be the extension directive which must precede other HSAIL statements and applies to the whole module. See [Chapter 13. Directives \(page 274\)](#).

Figure 4–7 moduleDirective Syntax Diagram



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

Figure 4–8 moduleStatement Syntax Diagram



4.3.1 Annotations

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

Figure 4–9 annotations Syntax Diagram

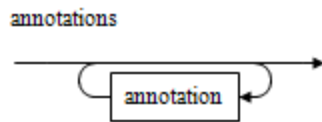
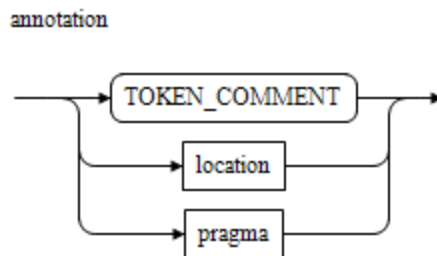


Figure 4–10 annotation Syntax Diagram

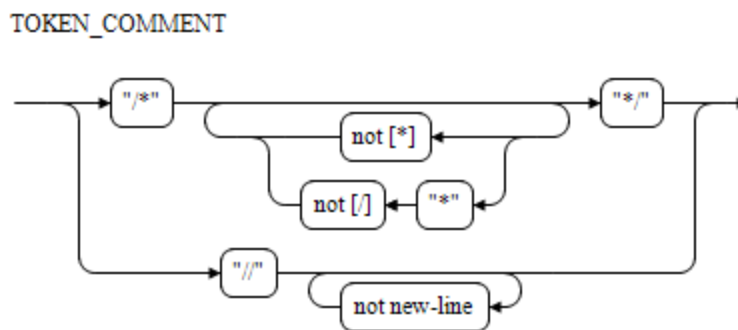


Comments that can span multiple lines use non-nested `/*` and `*/`. The comment starts at the `/*` and extends to the next `*/`, which might be on a different line.

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 4–11 `TOKEN_COMMENT` Syntax Diagram

For more information on location and pragma directives, see [Chapter 13. Directives \(page 274\)](#).

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 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. If a kernel is declared with module linkage, then it must have a definition in the same module. See [4.12. Linkage \(page 97\)](#).

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 none linkage as they are not referenced by any instructions.

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

Figure 4–12 kernel Syntax Diagram

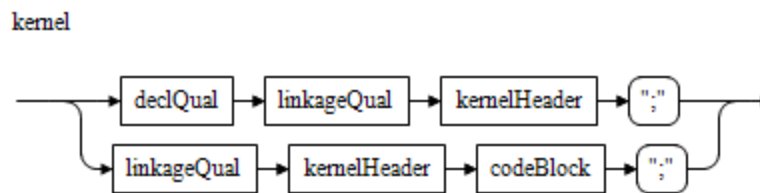


Figure 4–13 kernelHeader Syntax Diagram

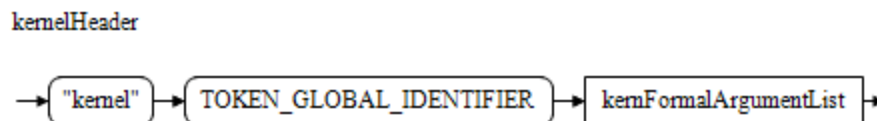
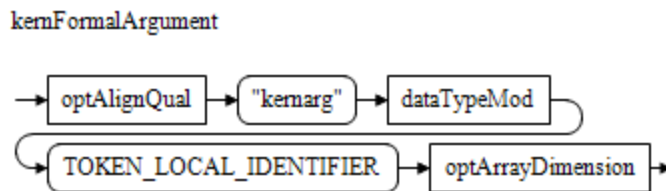


Figure 4–14 kernFormalArgumentList Syntax Diagram



Figure 4–15 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 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. If a function is declared with module linkage, then it must have a definition in the same module. See [4.12. Linkage \(page 97\)](#).

An indirect function has limitations on the symbols it can reference. See [10.8. Indirect Call \(icall\) Instruction \(page 252\)](#).

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 none 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 in which they are defined.

Figure 4–16 function Syntax Diagram

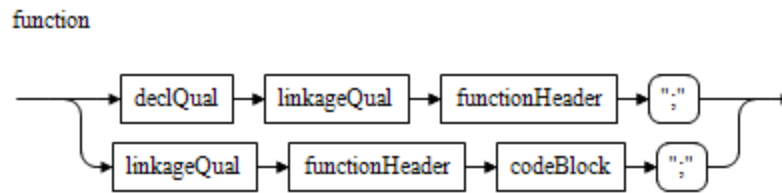


Figure 4–17 functionHeader Syntax Diagram

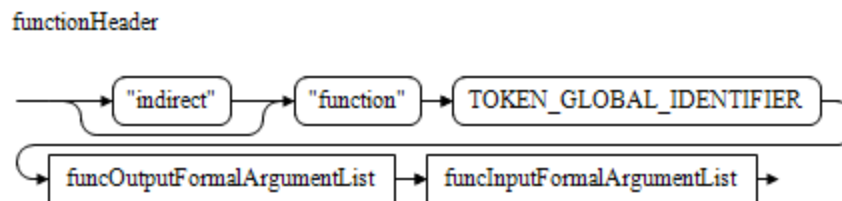


Figure 4–18 funcOutputFormalArgumentList Syntax Diagram

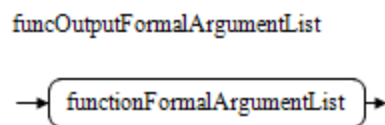


Figure 4–19 funcInputFormalArgumentList Syntax Diagram

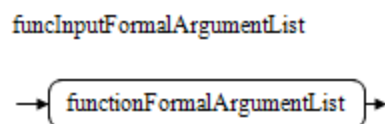


Figure 4–20 funcFormalArgumentList Syntax Diagram

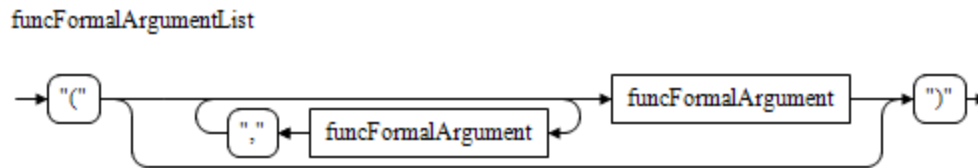
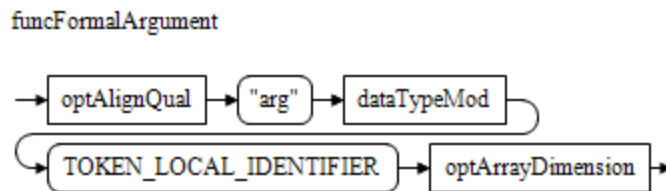


Figure 4–21 funcFormalArgument Syntax Diagram



For more information, see [Chapter 10. Function Instructions \(page 243\)](#).

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` instruction.

Syntactically, a signature is much like a function.

The arguments of a signature have none linkage as they are not referenced by any instructions.

Figure 4–22 signature Syntax Diagram

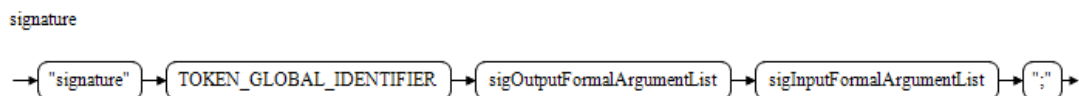


Figure 4–23 sigOutputFormalArgumentList

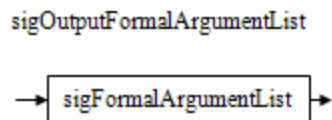


Figure 4–24 sigInputFormalArgumentList Syntax Diagram

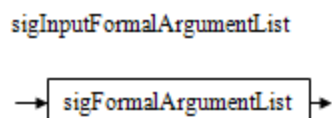


Figure 4–25 sigFormalArgumentList Syntax Diagram

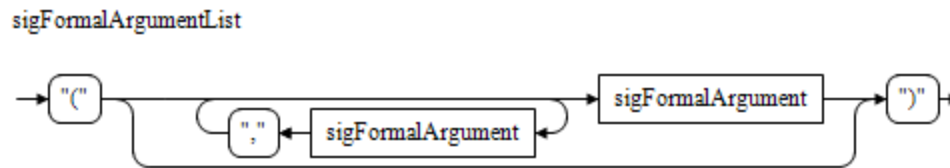
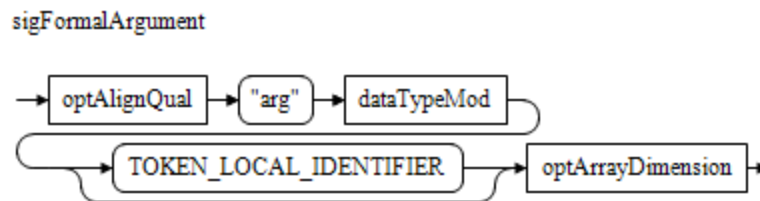


Figure 4–26 sigFormalArgument Syntax Diagram

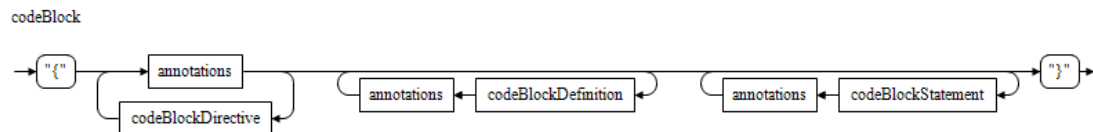


For more information, see [Chapter 10. Function Instructions](#) (page 243).

4.3.5 Code Block

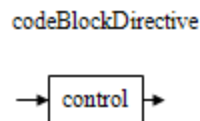
A code block consists of zero or more code block directives, followed by zero or more code block definitions, followed by zero or more code block statements, all surrounded by curly brackets ({ }).

Figure 4–27 codeBlock Syntax Diagram



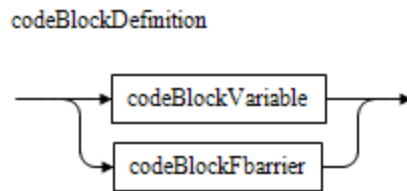
A code block directive can be a control directive which must precede other HSAIL statements in the code block and applies to the kernel or function with which the code block is associated.

Figure 4–28 codeBlockDirective Syntax Diagram



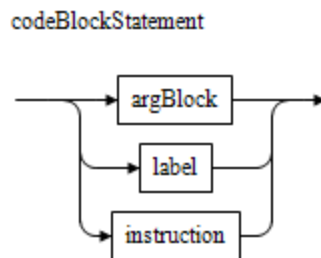
A code block definition can be a code block variable or code block fbarrier.

Figure 4–29 codeBlockDefinition



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

Figure 4–30 codeBlockStatement Syntax Diagram



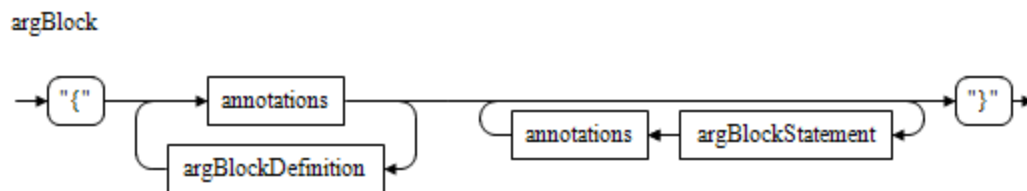
For more information on:

- Control directives, see [Chapter 13. Directives \(page 274\)](#).
- Labels, see [4.9. Labels \(page 94\)](#).

4.3.6 Arg Block

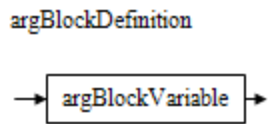
An arg block consists of zero or more arg block definitions, followed by one or more arg block statements, which must include exactly one call instruction, all surrounded by curly brackets ({ }). An arg block is used to pass argument values into and out of a call to a function. See [10.2. Function Call Argument Passing \(page 244\)](#).

Figure 4–31 argBlock Syntax Diagram



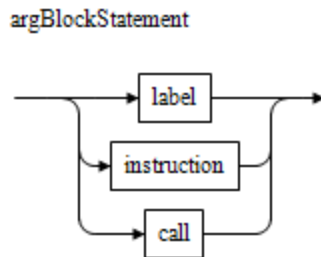
An arg block definition can be an arg block variable.

Figure 4–32 argBlockDefinition



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

Figure 4–33 argBlockStatement Syntax Diagram



For more information, see [10.2. Function Call Argument Passing \(page 244\)](#).

4.3.7 Instruction

An instruction is an executable HSAIL statement.

The example below shows four instructions:

```

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
  
```

Instructions 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 instructions 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 instructions also support lists of operands surrounded by parentheses (()) or square brackets ([]). The destination operand is first, followed by source operands. See [4.16. Operands \(page 104\)](#).

HSAIL allows a finalizer to support extensions that add additional features to HSAIL, for example, additional instructions and data types. A *finalizer extension* is enabled using the `extension` directive. Any instructions enabled by a finalizer extension are accessed like all other HSAIL instructions. For more information, see [13.1.3. How to Set Up Finalizer Extensions \(page 275\)](#).

For more information, see:

- [Chapter 5. Arithmetic Instructions \(page 116\)](#)
- [Chapter 6. Memory Instructions \(page 166\)](#)

- [Chapter 7. Image Instructions \(page 194\)](#)
- [Chapter 8. Branch Instructions \(page 227\)](#)
- [Chapter 9. Parallel Synchronization and Communication Instructions \(page 229\)](#)
- [Chapter 10. Function Instructions \(page 243\)](#)
- [Chapter 11. Special Instructions \(page 257\)](#)

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 a code object that references the variable is loaded into an executable. The memory is destroyed when the HSA runtime is used to destroy the executable. All HSAIL executables 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 byte size of the initializer divided by the byte size of the variable data type. It is an error if the initializer byte size is not an exact multiple of the variable data type byte size. (The `b1` bit type is not allowed for variables.)
 - 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 \(page 78\)](#).

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

An arg block variable has arg linkage and can only be referenced within the arg block in which 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. If a module variable is declared with module linkage, then it must have a definition in the same module. See [4.12. Linkage \(page 97\)](#).

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` instructions) 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 4–34 moduleVariable Syntax Diagram

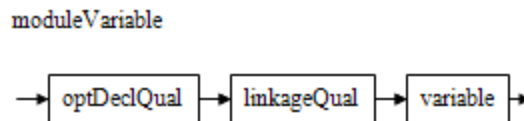


Figure 4–35 codeBlockVariable Syntax Diagram

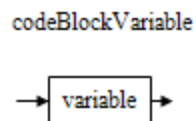


Figure 4–36 argBlockVariable Syntax Diagram

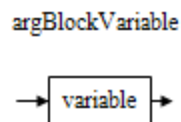
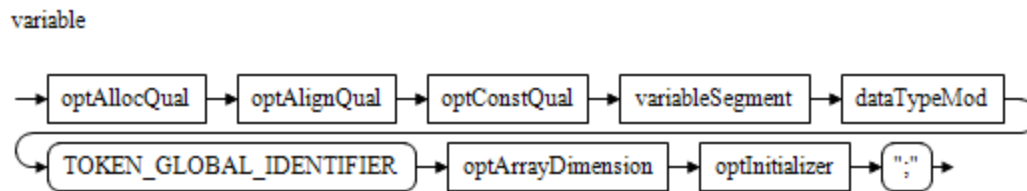


Figure 4–37 variable Syntax Diagram

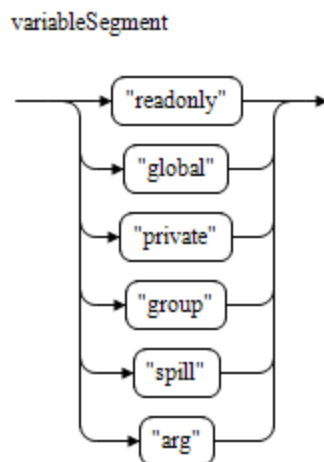


A variable segment can be one of the following:

- Readonly: Only allowed for module and code block variables.
- Global: Only allowed for module and code block variables.
- Group: Only allowed for kernel code block variables. In addition, allowed for module and function code block variables as an experimental feature (see [1.3. HSAIL Experimental Features \(page 22\)](#)).
- Private: Only allowed for code block variables. In addition, allowed for module variables as an experimental feature (see [1.3. HSAIL Experimental Features \(page 22\)](#)).
- Spill: Only allowed for code block variables.
- Arg: Only allowed for arg block variables.

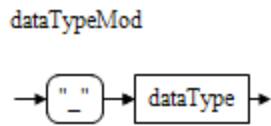
The syntax for kernarg and arg segment formal argument variables is defined in [4.3.2. Kernel \(page 56\)](#) and [4.3.3. Function \(page 58\)](#) respectively.

Figure 4–38 variableSegment Syntax Diagram



The variable data type can be one of the data types described in [4.13. Data Types \(page 99\)](#), except for b1.

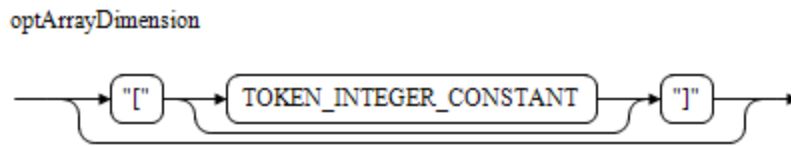
Figure 4–39 dataTypeMod Syntax Diagram



Variables that hold addresses of variables, kernel code handles or indirect function code handles should be of type `u` and of size 32 or 64 depending on the machine model (see [2.9. Small and Large Machine Models \(page 39\)](#)).

Array variables are provided to allow the high-level compiler to reserve a memory block of arbitrary size. 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. An integer constant with a value of 0 is not allowed. `WAVESIZE` is not allowed. Note that the array declaration is similar to the C++ language.

Figure 4–40 optArrayDimension Syntax Diagram



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

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

The last formal argument of a function or signature can be an array without a specified dimension. The size passed is determined by the size of the arg segment variable definition passed to the function by the call instruction. This is used to support variadic function calls. See [10.4. Variadic Functions \(page 248\)](#).

The array dimension of a global or readonly segment variable definition can be left empty in which case an initializer must be specified and is used to provide the array dimension size.

A variable can have an optional initializer. An initializer is only allowed for variable definitions for the following segments:

- Global
- Readonly

If there is no initializer, the value of the variable is undefined when it is allocated.

For more information on:

- Variable initializers, see [4.10. Variable Initializers \(page 94\)](#).
- Identifier scopes, see [4.6.2. Scope \(page 78\)](#).
- Variable storage duration, see [4.11. Storage Duration \(page 96\)](#).

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 and must be compatible.

fbarrier definition and declarations are compatible if they have the same linkage.

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. If a module fbarrier is declared with module linkage, then it must have a definition in the same module. See [4.12. Linkage \(page 97\)](#).

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 \(page 78\)](#).

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

At the time a kernel is finalized, there must be a definition for all the fine-grain barriers referenced by fbarrier instructions that are part of the kernel (including any indirect references from instructions in functions they call by `call` and `scall` instructions) 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 4–41 moduleFbarrier Syntax Diagram

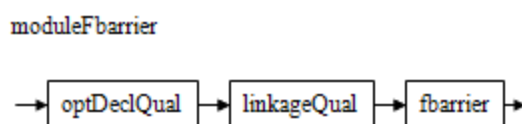


Figure 4–42 codeBlockFbarrier Syntax Diagram

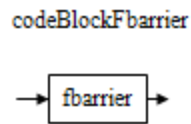
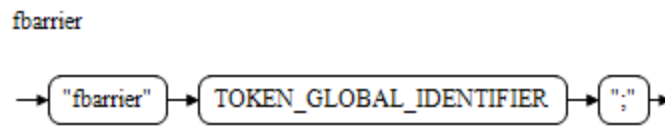


Figure 4–43 fbarrier Syntax Diagram



For more information, see [9.2. Fine-Grain Barrier \(fbarrier\) Instructions \(page 230\)](#).

4.3.10 Declaration and Definition Qualifiers

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

Figure 4–44 optDeclQual Syntax Diagram

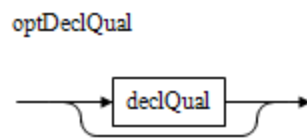


Figure 4–45 declQual Syntax Diagram

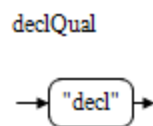


Figure 4–46 linkageQual Syntax Diagram

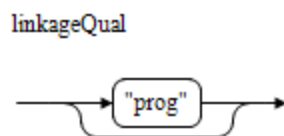


Figure 4–47 optAlignQual Syntax Diagram



Figure 4–48 optAllocQual Syntax Diagram

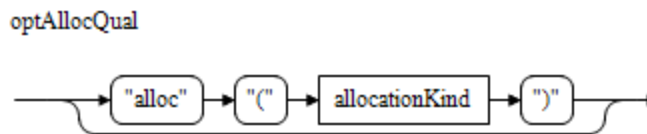
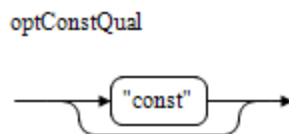


Figure 4–49 optConstQual Syntax Diagram



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.

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 \(page 97\)](#).

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, and automatic allocation for all other 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` instruction 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 \(page 211\)](#) and [7.1.8. Sampler Creation and Sampler Handles \(page 214\)](#)).

agent allocation

Causes the HSA runtime to perform a separate allocation for the variable for each kernel agent on which a code object that references the variable is loaded (see [4.2. Program, Code Object, and Executable \(page 48\)](#) and [6.2.5. Agent Allocation \(page 171\)](#)). Each separate allocation will have a unique global segment address. The results are undefined if the variable is accessed 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 a kernel agent will access the variable allocation that is associated with that agent. An `lda` instruction performed on the variable will obtain the distinct segment address for the allocation associated with the kernel agent on which it is executed, but the results are undefined if any other agent accesses that address. The variable's global segment address can be converted to a flat address, but the results are undefined if any other agent accesses 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 \(page 211\)](#) and [7.1.8. Sampler Creation and Sampler Handles \(page 214\)](#)).

automatic allocation

Causes variables to be automatically allocated at the start of the variable's storage duration. See [4.11. Storage Duration \(page 96\)](#).

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.

Array variables are naturally aligned to the size of the array element type (not the size of the entire array).

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` instruction 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 \(page 96\)](#).

The results are undefined if a store or atomic write or read-modify-write instruction is used 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 \(page 295\)](#).

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 \(page 78\)](#).

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 \(page 78\)](#).

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 \(page 78\)](#).

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 \(page 78\)](#).

The memory layout of variables in the kernarg segment is defined in [4.21. Kernarg Segment \(page 114\)](#).

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 \(page 78\)](#).

The memory layout of multiple variables in the spill segment is implementation defined, and a finalizer may promote them to hardware registers. The `lda` instruction 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 \(page 78\)](#).

The memory layout of multiple variables in the `arg` segment is implementation defined, and a finalizer may promote them to hardware registers. The `lda` instruction 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 \(page 96\)](#) for a description of when storage is allocated for variables.

Also see:

- [7.1.7. Image Creation and Image Handles \(page 211\)](#)
- [7.1.8. Sampler Creation and Sampler Handles \(page 214\)](#)

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.

The source text is broken into the following lexical tokens:

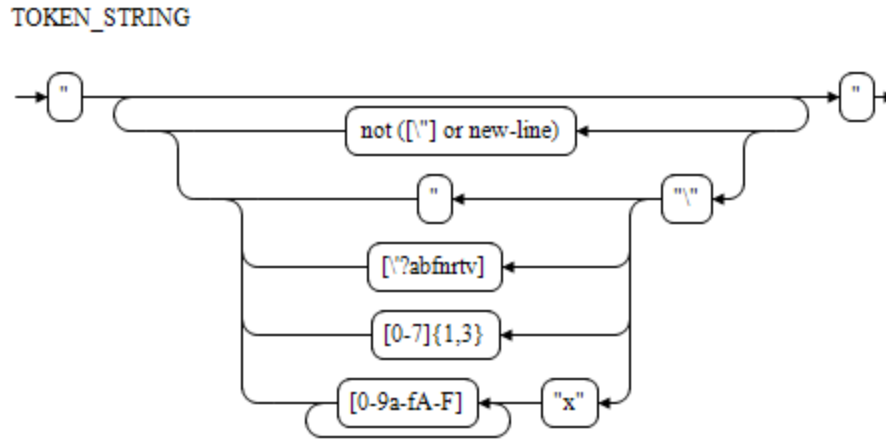
- `TOKEN_COMMENT` (see [4.3.1. Annotations \(page 55\)](#))
- `TOKEN_GLOBAL_IDENTIFIER` (see [4.6. Identifiers \(page 77\)](#))
- `TOKEN_LOCAL_IDENTIFIER` (see [4.6. Identifiers \(page 77\)](#))
- `TOKEN_LABEL_IDENTIFIER` (see [4.6. Identifiers \(page 77\)](#))
- `TOKEN_CREGISTER` (see [4.7. Registers \(page 79\)](#))
- `TOKEN_SREGISTER` (see [4.7. Registers \(page 79\)](#))
- `TOKEN_DREGISTER` (see [4.7. Registers \(page 79\)](#))
- `TOKEN_QREGISTER` (see [4.7. Registers \(page 79\)](#))
- `TOKEN_INTEGER_CONSTANT` (see [4.8.1. Integer Constants \(page 82\)](#))
- `TOKEN_HALF_CONSTANT` (see [4.8.2. Floating-Point Constants \(page 83\)](#))
- `TOKEN_SINGLE_CONSTANT` (see [4.8.2. Floating-Point Constants \(page 83\)](#))
- `TOKEN_DOUBLE_CONSTANT` (see [4.8.2. Floating-Point Constants \(page 83\)](#))
- `TOKEN_WAVESIZE` (see [2.6.2. Wavefront Size \(page 30\)](#))
- `TOKEN_STRING` (see [4.5. Strings \(next page\)](#))
- Tokens for all the terminal symbols of the HSAIL syntax (see [19.2. HSAIL Syntax Grammar in Extended Backus-Naur Form \(EBNF\) \(page 361\)](#))

Lexical tokens can be separated by zero or more white space characters: space, horizontal tab, new-line, vertical tab and form-feed. Whitespace is required between lexical tokens that can include alphabetic or numeric characters.

See [19.1. HSAIL Lexical Grammar in Extended Backus-Naur Form \(EBNF\) \(page 360\)](#).

4.5 Strings

Figure 4–50 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.

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 4–51 TOKEN_GLOBAL_IDENTIFIER Syntax Diagram

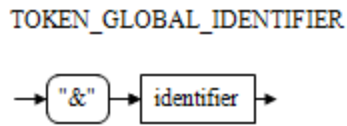


Figure 4–52 TOKEN_LOCAL_IDENTIFIER Syntax Diagram

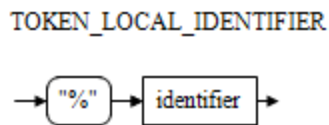


Figure 4–53 TOKEN_LABEL_IDENTIFIER Syntax Diagram

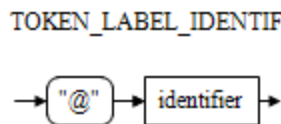
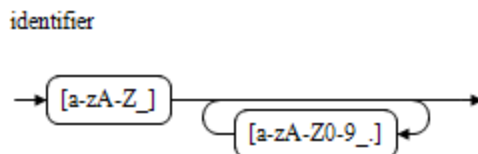


Figure 4–54 identifier Syntax Diagram



4.6.1 Syntax

Identifiers that are register names must start with a dollar sign (\$). See [4.7. Registers \(page 79\)](#).

Identifiers that are labels must start with an at sign (@). See [4.9. Labels \(page 94\)](#).

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 \(facing page\)](#).

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

The remaining characters of an identifier can be either letters, digits, underscore (`_`), or dot (`.`).

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 \(page 64\)](#).

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. The identifier in the module header has module scope.
- 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 \(page 244\)](#).
- 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 have argument scope and 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 4–55 TOKEN_CREGISTER Syntax Diagram

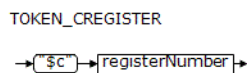


Figure 4–56 TOKEN_SREGISTER Syntax Diagram

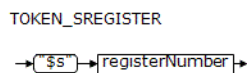


Figure 4–57 TOKEN_DREGISTER Syntax Diagram

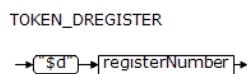


Figure 4–58 TOKEN_QREGISTER Syntax Diagram

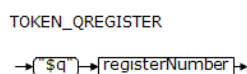
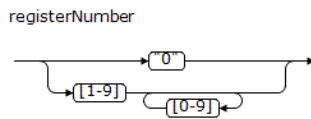


Figure 4–59 registerNumber Syntax Diagram



There are four types of registers:

- Control registers (c registers)

These hold a single bit value.

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

Control registers are similar to CPU condition codes.

These registers are named \$c0, \$c1, \$c2, and so on.

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.

These registers are named \$s0, \$s1, \$s2, and so on.

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.

These registers are named \$d0, \$d1, \$d2, and so on.

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.

These registers are named \$q0, \$q1, \$q2, and so on.

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 `c` registers in HSAIL are a single pool of resources per function scope. It is an error if the value $(c_{\max} + 1)$ exceeds 128 for any kernel or function definition, where c_{\max} is the highest `c` register number in the kernel or function code block, or -1 if no `c` registers are used. For example, if a function code block only uses registers `$c0` and `$c7`, then c_{\max} is 7 not 2.
- 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 2048 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 registers than the HSAIL register limits 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 four kinds of constants: integer constant, floating-point constant, typed constant, and aggregate constant. Constant values can be used to specify the initial value of variable definitions, and the value of immediate operands of instructions and directives.

Figure 4–60 initializerConstant Syntax Diagram

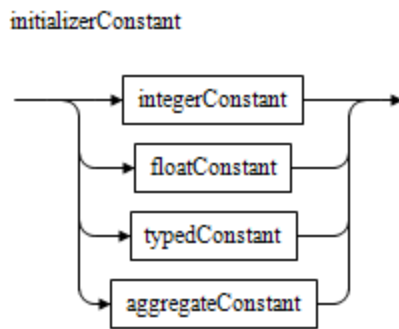
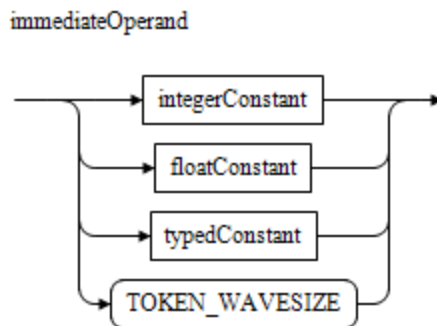


Figure 4–61 immediateOperand Syntax Diagram



All constants must be compatible with the data type of the expected value according to the rules in [Table 4–1 \(page 92\)](#). The data type of the expected value is determined by where the constant is used:

- Data initialization directives: the expected value type is the type of the variable being initialized.
- Instruction source operands: the expected value type is the type of the operand defined by the instruction.
- Instruction address expressions: the expected value type is an unsigned integer of the address size. See [Table 2–3 \(page 40\)](#). This is true if the integer constant specifies an absolute address or is an address offset for a base address specified by a symbol or register.
- Directive and module header operands: the expected value type of each operand is specified by the directive.
- Other usage: the expected value type used is `u64`. These include array dimensions, image size properties, alignment, equivalence class, the integer constant in the signal typed constant for the null signal handle, and so forth.

4.8.1 Integer Constants

Figure 4–62 TOKEN_INTEGER_CONSTANT Syntax Diagram

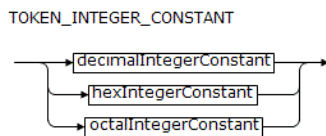


Figure 4–63 decimalIntegerConstant Syntax Diagram

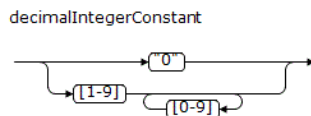


Figure 4–64 hexIntegerConstant Syntax Diagram

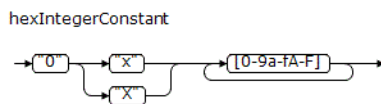
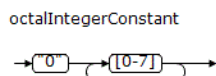


Figure 4–65 octalIntegerConstant Syntax Diagram

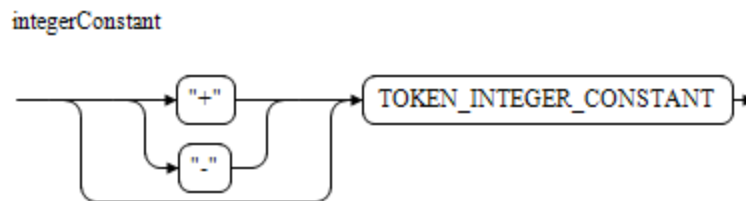


Integer constants are 64-bit unsigned values. In the Extended Backus-Naur Form syntax, an integer constant is referred to as a `TOKEN_INTEGER_CONSTANT`.

Integer constants are only valid for integer types, and for bit types less than or equal to 64 bits. See [4.13.1. Base Data Types \(page 99\)](#) and [4.8.5. How Text Format Constants Are Converted to Bit String Constants \(page 92\)](#). 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 64-bit signed value.

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 64-bit value and negated, regardless of whether the constant type is a signed integer type, and the resulting bits used as the value.

Figure 4–66 integerConstant Syntax Diagram



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 integer constant values that are bigger than needed. For example, in the following code, the 24 and 25 are 64-bit unsigned constant values, but the variable initializer and instruction expect 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;
add_s32 $s1, 24, 25;
```

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

- A decimal integer constant starts with a non-zero digit. See [Figure 4–63 \(previous page\)](#).
- A hexadecimal integer constant starts with 0x or 0X. See [Figure 4–64 \(previous page\)](#).
- An octal integer constant starts with 0. See [Figure 4–65 \(previous page\)](#).

4.8.2 Floating-Point Constants

Figure 4–67 TOKEN_HALF_CONSTANT Syntax Diagram

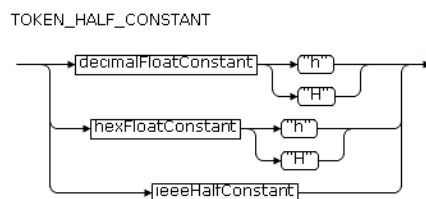


Figure 4–68 TOKEN_SINGLE_CONSTANT Syntax Diagram

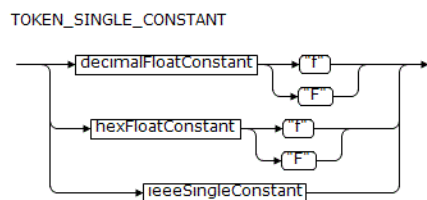


Figure 4–69 TOKEN_DOUBLE_CONSTANT Syntax Diagram

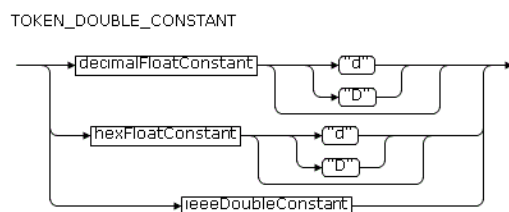


Figure 4–70 decimalFloatConstant Syntax Diagram

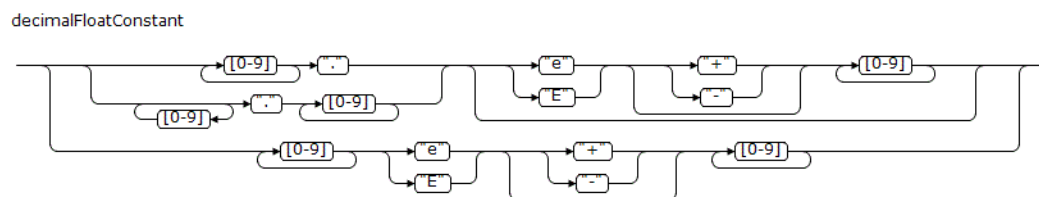


Figure 4–71 hexFloatConstant Syntax Diagram

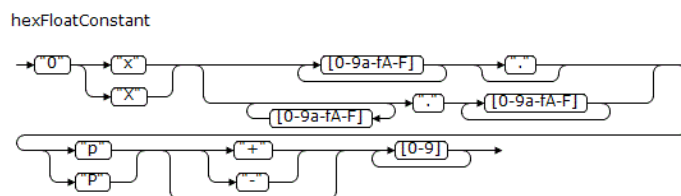


Figure 4–72 ieeeHalfConstant Syntax Diagram

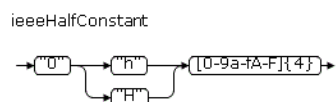


Figure 4–73 ieeeSingleConstant Syntax Diagram

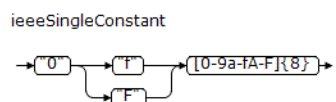
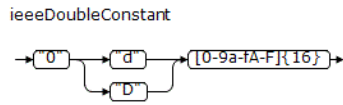


Figure 4–74 ieeeDoubleConstant Syntax Diagram



Floating-point constants are represented as either:

- 16-bit single-precision

It is an error to use a half-precision float constant unless the expected value type is `f16` or `b16`. See [4.8.5. How Text Format Constants Are Converted to Bit String Constants \(page 92\)](#). 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 expected value type is `f32` or `b32`. See [4.8.5. How Text Format Constants Are Converted to Bit String Constants \(page 92\)](#). 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 expected value type is `f64` or `b64`. See [4.8.5. How Text Format Constants Are Converted to Bit String Constants \(page 92\)](#). 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 \(page 289\)](#)).

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.

Figure 4–75 floatConstant Syntax Diagram

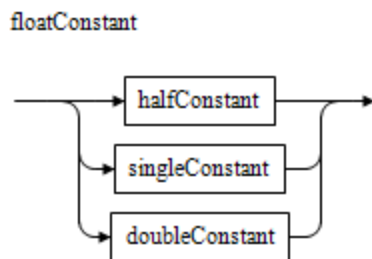


Figure 4–76 halfConstant Syntax Diagram

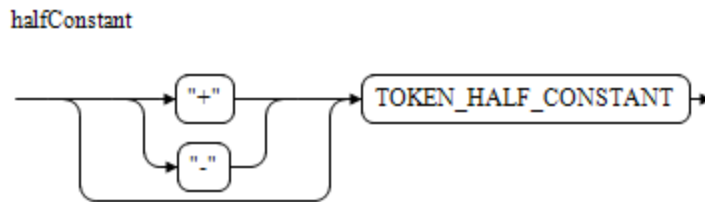


Figure 4–77 singleConstant Syntax Diagram

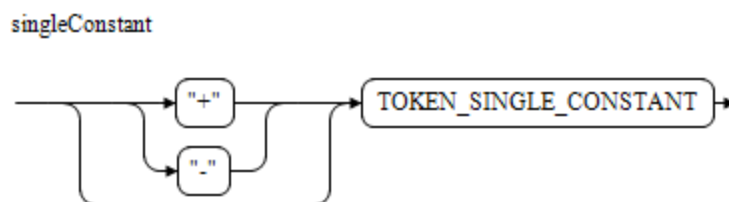
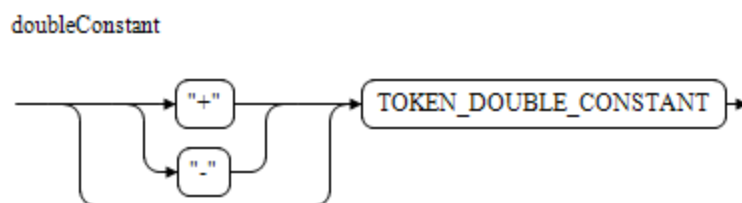


Figure 4–78 doubleConstant Syntax Diagram



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++ language 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. Floating-Point Rounding](#) (page 109)). See [Figure 4–70](#) (page 84).

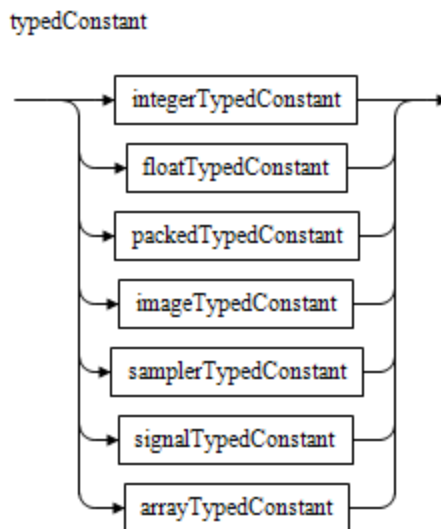
- A hexadecimal floating-point constant can be written using the C99 format. It consists of a hexadecimal prefix of 0x or 0X, a significand part, a binary exponent part, and a float size suffix. The significand part represents a rational number and consists of a sequence of hexadecimal digits (the whole 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 4–71 \(page 84\)](#).
- 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 Typed Constants

A non-array typed constant consists of a data type, followed by parenthesized arguments to provide a value of that type. The byte size of a non-array typed constant is the byte size of the data type.

Figure 4–79 typedConstant Syntax Diagram



Bit typed constants are not supported. Instead the value of a bit type can be specified using one of the other constant kinds such as an integer, floating-point, or packed constant.

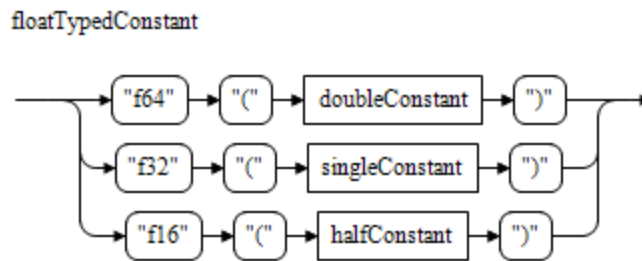
An integer typed constant requires the argument to be an integer constant which is truncated to the size of the integer type.

Figure 4–80 integerTypedConstant Syntax Diagram



A floating-point typed constant requires the argument to be a floating-point constant that is the same byte size as the floating-point type. The 64-bit floating-point type ($\mathbb{F}64$) is not supported by the Base profile (see [16.2.1. Base Profile Requirements \(page 289\)](#)).

Figure 4–81 floatTypedConstant Syntax Diagram



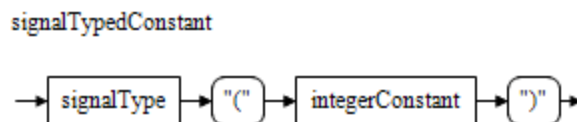
For information on packed type constants, see [4.14.2. Packed Type Constants \(page 103\)](#).

For information on image handle type constants, see [7.1.7. Image Creation and Image Handles \(page 211\)](#). They are allowed for constants used as a variable initializer and cause a corresponding image to be created with the properties specified. They are also allowed as the operand of a pragma directive (see [13.3. pragma Directive \(page 276\)](#)).

For information on sampler handle type constants, see [7.1.8. Sampler Creation and Sampler Handles \(page 214\)](#). They are allowed for constants used as a variable initializer and cause a corresponding sampler to be created with the properties specified. They are also allowed as the operand of a pragma directive (see [13.3. pragma Directive \(page 276\)](#)).

A signal handle typed constant requires the argument to be an integer constant with the value zero. This represents the null signal handle. The integer constant is treated as a `u64` type.

Figure 4–82 signalTypedConstant Syntax Diagram



An array typed constant consists of an array element data type, followed by “[]”, followed by parenthesized elements to provide a value of each array element. The array element data type can be any type except an array type or a bit type. Each array element must be a constant that is compatible with the array element data type according to the rules in [Table 4–1 \(page 92\)](#). The byte size of an array typed constant is the byte size of the array element data type multiplied by the number of array elements

Figure 4–83 arrayTypedConstant Syntax Diagram

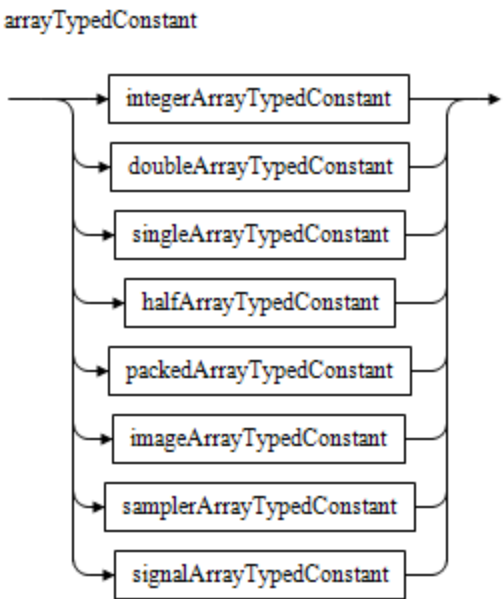


Figure 4–84 integerArrayTypedConstant Syntax Diagram

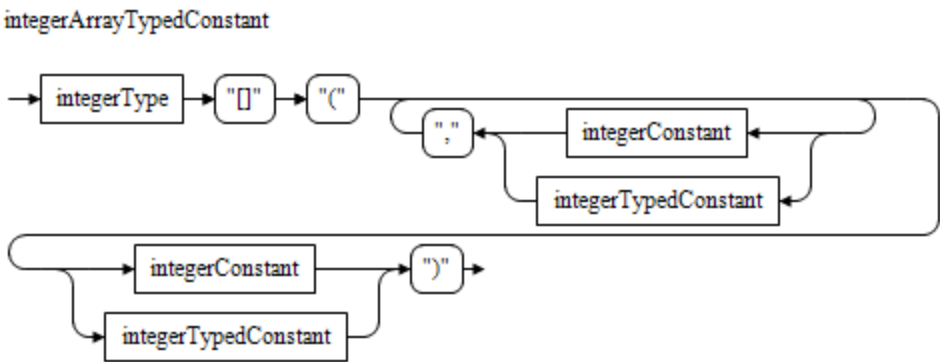


Figure 4–85 halfArrayTypedConstant Syntax Diagram

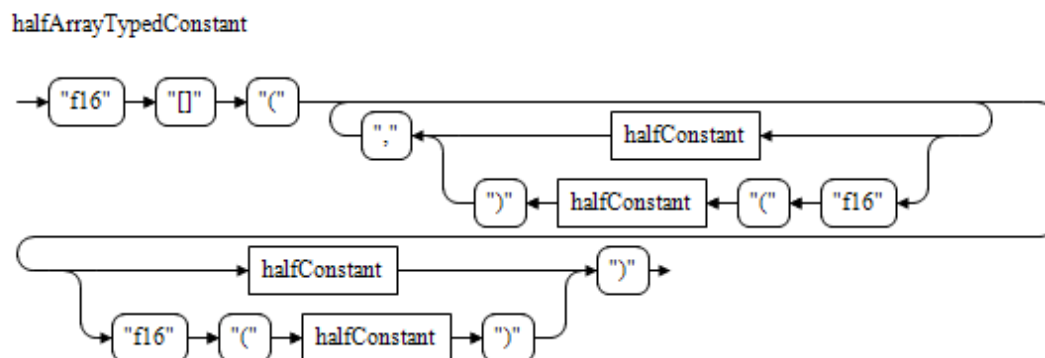


Figure 4–86 singleArrayTypedConstant Syntax Diagram

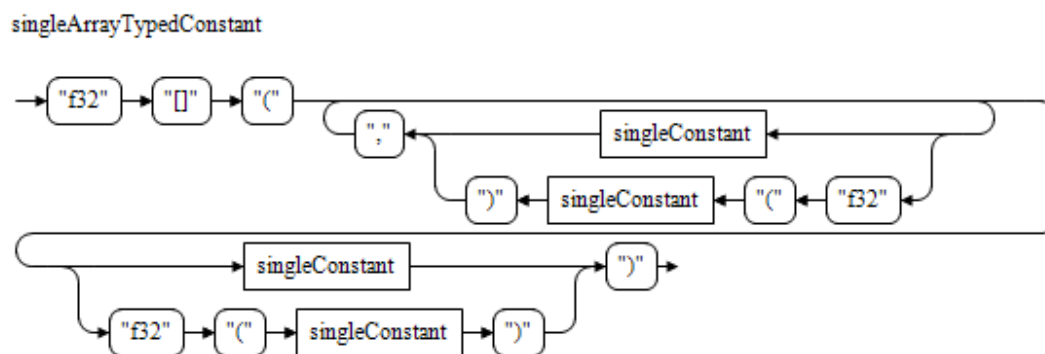


Figure 4–87 doubleArrayTypedConstant Syntax Diagram

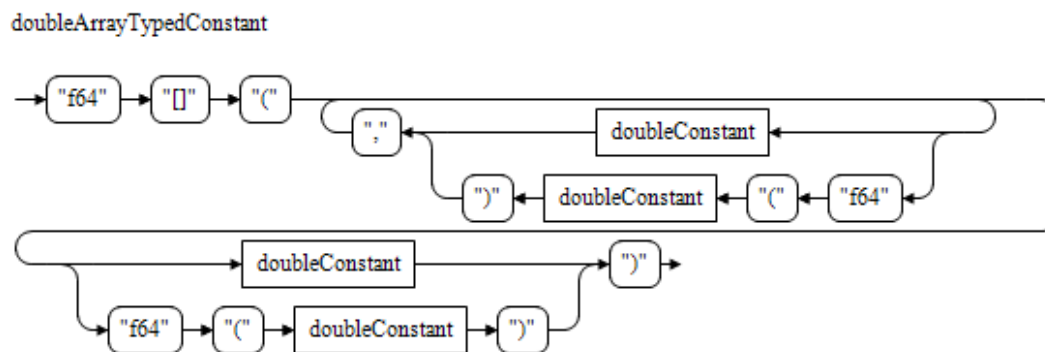


Figure 4–88 packedArrayTypedConstant Syntax Diagram

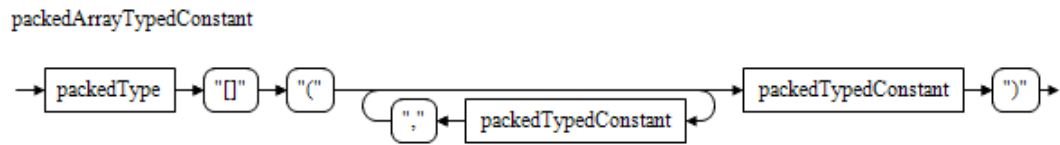


Figure 4–89 imageArrayTypedConstant Syntax Diagram



Figure 4–90 samplerArrayTypedConstant Syntax Diagram

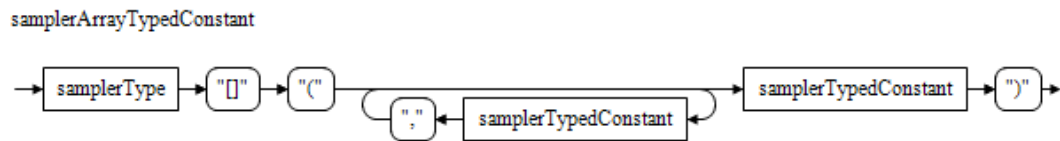
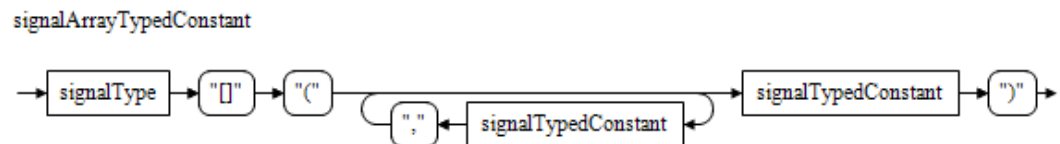


Figure 4–91 signalArrayTypedConstant Syntax Diagram



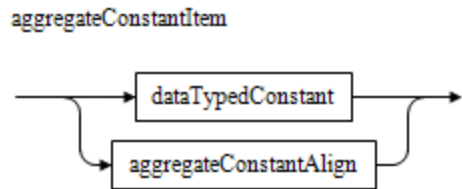
4.8.4 Aggregate Constants

An aggregate constant consists of a comma separated list of typed constants and alignment requests enclosed in curly brackets.

Figure 4–92 aggregateConstant Syntax Diagram



Figure 4–93 aggregateConstantItem Syntax Diagram

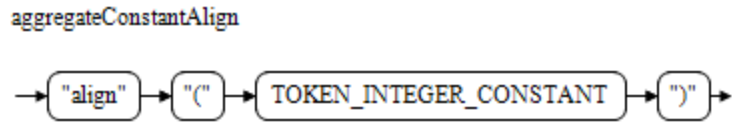


The bytes of the typed constant aggregate element values are ordered consecutively starting at the lowest addressed byte and do not have to be the same type. The byte size of each value is the byte size of the typed constant. There is no padding between values, therefore values need not be naturally aligned. This allows aggregate constants to provide a constant value for arbitrary structures which have different field types, as well as for arrays that have the same type for each element. The byte size of an aggregate constant is the sum of the sizes of its elements.

In addition, an aggregate constant element can be an alignment request: `align (n)`. This causes enough zero bytes to be generated to ensure the next element starts on the specified alignment relative to the start of the aggregate constant. If the alignment request appears as the last element, it causes zero bytes to be generated to make the aggregate constant byte size a multiple of the specified alignment. An aggregate constant cannot consist of only alignment request elements. Valid values of *n* are 1, 2, 4, 8, 16, 32, 64, 128, and 256.

An aggregate constant is used as the initializer of a bit type array variable. Any alignment requests the aggregate initializer contains do not influence the alignment of the variable it initializes.

Figure 4–94 aggregateConstantAlign Syntax Diagram



4.8.5 How Text Format Constants Are Converted to Bit String Constants

Tools can convert between the HSAIL text format and the BRIG binary format. See [Table 4–1 \(below\)](#), which describes how HSAIL text format constants are converted to bit string constants used in BRIG. What happens with the conversion depends on the data type expected by the operation.

Table 4–1 Text Constants and Results of the Conversion

Kind of text format constant provided (see 4.8. Constants (page 81))	Data type of expected value (see 4.13. Data Types (page 99))								
	Bit type	Signed/unsigned integer type	Floating-point type	Packed type	Image type	Sampler type	Signal type	Non-bit type	Bit type array
Integer constant	Truncate	Truncate	Error	Error	Error	Error	Error	Error	Error

Kind of text format constant provided (see 4.8. Constants (page 81))	Data type of expected value (see 4.13. Data Types (page 99))								
	Bit type	Signed/unsigned integer type	Floating-point type	Packed type	Image type	Sampler type	Signal type	Non-bit type	Bit type array
Floating-point constant	Length-only rule	Error	Type and length rule	Error	Error	Error	Error	Error	Error
Integer typed constant	Length-only rule	Type and length rule	Error	Error	Error	Error	Error	Error	Error
Float typed constant	Length-only rule	Error	Type and length rule	Error	Error	Error	Error	Error	Error
Packed typed constant	Length-only rule	Error	Error	Type and length rule	Error	Error	Error	Error	Error
Image typed constant	Error	Error	Error	Error	Type and length rule	Error	Error	Error	Error
Sampler typed constant	Error	Error	Error	Error	Error	Type and length rule	Error	Error	Error
Signal typed constant	Error	Error	Error	Error	Error	Error	Type and length rule	Error	Error
Array typed constant	Error	Error	Error	Error	Error	Error	Error	Type and length rule	Error
Aggregate constant	Error	Error	Error	Error	Error	Error	Error	Error	Length-only rule

Truncation for an integer value in the text is as follows: the value is input as 64 bits in 2s complement, then the length needed is compared to the size the instruction needs:

- If the instruction needs 64 bits or fewer, the 64-bit value is truncated if necessary.
- If the instruction needs more than 64 bits, it is an error.

For example:

```
add_s64 $d0, $d0, 0xffffffff; // Legal: 9 f's stored as 0x0000000fffffffff.
```

The 9 f's represents a 64-bit integer constant with 36 non-zero bits. The operation uses an integer type s64, so the number of bits match identically in BRIG. This would be stored as s64 0x0000000fffffffff.

```
add_s32 $s0, $s0, 0xffffffff; // Legal: 9 f's truncated and stored as 0xffffffff.
```

The `s32` is 32 bits, the constant would be truncated and stored as `s32 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 instruction requires 128 bits.
```

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

```
add_pp_u64x2 $q1, $q0, u64x2(1, 2); // Legal as packed types match.
add_pp_u64x2 $q1, $q0, 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 an immediate operand value, 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 \(page 30\)](#).

4.9 Labels

Label identifiers consist of an at sign (`@`) followed by the name of the identifier (see [4.6. Identifiers \(page 77\)](#)).

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 [8.1. Syntax \(page 227\)](#).

4.10 Variable Initializers

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 for the variable.

Figure 4–95 optInitializer Syntax Diagram



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 allocated by the HSA runtime (see [4.2. Program, Code Object, and Executable \(page 48\)](#)), an initial value is assigned if it has an initializer. The initialization is performed only once when the memory is allocated.

When a global segment variable is initialized by the HSA runtime, a release memory fence on the global segment at system memory scope is performed. The results are 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 (see [4.3.10. Declaration and Definition Qualifiers \(page 69\)](#)). When a readonly segment variable is defined and initialized, 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: this also makes the value visible to all subsequent dispatches on the corresponding agent. The results are undefined if the agent allocation is accessed by kernel dispatches that were executing before the variable's definition initialization, or HSA runtime update.

The initial value is specified as a constant (see [4.8. Constants \(page 81\)](#)):

- If the variable has no array dimension specified, then an integer constant, float constant, or non-array typed constant is allowed according to the rules in [Table 4-1 \(page 92\)](#) based on the type of the variable. `WAVESIZE` is not allowed.
- If the variable has an array dimension specified then an array typed constant or aggregate constant is allowed according to the rules in [Table 4-1 \(page 92\)](#) based on the array type of the variable. `WAVESIZE` is not allowed.

It is an error if the byte size of the constant is not the same as the byte size of the variable: smaller constants are not zero filled; larger constants are not truncated (except by the integer constant truncation rules). See [4.8.5. How Text Format Constants Are Converted to Bit String Constants \(page 92\)](#).

Image and sampler handle typed constants are allowed in variable initializers. When the HSA runtime allocates the variable, it initializes the handles to reference images and samplers that it also creates which have the specified properties. See [7.1.7. Image Creation and Image Handles \(page 211\)](#) and [7.1.8. Sampler Creation and Sampler Handles \(page 214\)](#).

For the initialization of signal handles, the initial value can be a signal typed constant with a value of 0 to indicate the null signal handle.

The array dimension of a variable definition can be left empty, in which case an initializer must be specified. In this case, the array dimension size is equal to the byte size of the constant initializer divided by the byte size of the variable element data type. It is an error if the initializer byte size is not an exact multiple of the variable element data type byte size. Note that the `b1` bit type is not allowed for a variable type or initializer typed constant value so all variables are an integral number of bytes.

If there is no initializer, the value of the variable is undefined when it is allocated.

Examples

```
global_u32 &loc1; // no initializer, value starts as undefined
global_f32 &size = 1.0f;
global_b32 &x = 3.0f; // initializes the identifier &x to the 32-bit value 0x40400000
global_u32 &c[4]; // no initializer, all array element values start as undefined
global_u8 &bg[4] = u8[(1, 2, 3, 4)];
global_b8 &bg[4] = {u8(0), u8(0), u8(0), u8(0)};
global_b8 &bh[4] = {u8(0), u16(1), u8(2)};
global_b8 &bi[16] = {f32(1.0f), u16(1), align(8), sig64(0)};
readonly_u8 &days1[] = u8[(31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)];
// Equivalent to specifying &days1[12]
readonly_b8 &days2[] = {u8(31), u8(28), u8(31), u8(30), u8(31), u8(30),
                        u8(31), u8(31), u8(30), u8(31), u8(30), u8(31)};
// Equivalent to specifying &days2[12]
global_f32 &bias[] = f32[(-1.0f, 1.0f)]; // Equivalent to specifying &bias[2]
align(8) const global_b8 &willholddouble[8] = {u8(0), u8(0), u8(0), u8(0),
                                                u8(0), u8(0), u8(0), u8(0)};
decl global_u32 &c[]; // Declarations do not require an array dimension size
global_sig64 &s1 = sig64(0); // Signal handles should only be initialized with 0.
global_sig64 &sa[2] = sig64[(sig64(0), sig64(0x00))];
global_sig64 &se[] = sig64[(sig64(0), sig64(0x00), sig64(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 HSA code objects that have been finalized from an HSAIL program that includes an HSAIL module containing the definition are loaded into an HSA executable and lasts until the HSA executable is destroyed (see [4.2. Program, Code Object, and Executable \(page 48\)](#)). This corresponds to the C++ language notion of static storage duration. (See the C++ language 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 \(page 112\)](#)).

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++ language notion of automatic storage duration. (See the C++ language 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, Code Object, and Executable \(page 48\)](#) and [4.3. Module \(page 53\)](#).

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 \(page 78\)](#).

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*.

Global and readonly segment variables with program linkage may also be linked to definitions outside the HSAIL program using the HSA executable. In this case the name must be marked as a declaration in all modules of the program. The HSA runtime must be used to define the name for an executable in which the code object produced by the finalizer from the program will be loaded (see [4.2. Program, Code Object, and Executable \(page 48\)](#)). For agent allocation variables, it is required to define the name for each agent onto which the code object is loaded (see [6.2.5. Agent Allocation \(page 171\)](#)).

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:

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

// program linkage definition: contains the body
prog function &foo() ()
{
    // ... 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 of 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 4–2 \(below\)](#).

Table 4–2 Base Data Types

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

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

The 64-bit floating-point type (`f64`) is not supported by the Base profile (see [16.2.1. Base Profile Requirements \(page 289\)](#)). This includes segment variable declarations, segment variable definitions, double-precision floating-point constants and instructions.

Most instructions specify a single compound type, used for both destinations and sources. However, the conversion instructions (`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 4-3 \(below\)](#).

Table 4-3 Packed Data Types and Possible Lengths

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

32-bit sizes are:

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

64-bit sizes are:

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

128-bit sizes are:

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

The 64-bit floating-point packed type (`f64x2`) is not supported by the Base profile (see [16.2.1. Base Profile Requirements \(page 289\)](#)). This includes segment variable declarations, segment variable definitions, packed constants and instructions.

4.13.3 Opaque Data Types

HSAIL also has the following opaque types:

Table 4–4 Opaque Data Types

Type	Description	Length in bits
roimg	Read-only image handle	64
woimg	Write-only image handle	64
rwimg	Read-write image handle	64
samp	Sampler handle	64
sig32	Signal handle for signal with 32-bit signal value	64
sig64	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 \(page 274\)](#)). This includes segment variable declarations, segment variable definitions and instructions.

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 \(page 39\)](#)). This includes segment variable declarations, segment variable definitions and instructions.

For more information see:

- [7.1.7. Image Creation and Image Handles \(page 211\)](#)
- [7.1.8. Sampler Creation and Sampler Handles \(page 214\)](#)
- [6.8. Notification \(signal\) Instructions \(page 187\)](#)

4.13.4 Array Data Types

HSAIL also has array types. An array has a fixed number of contiguous elements all of the same array element type. The array element type can be any type except an array type or `b1`. The size of the array type is the size of the array element type multiplied by the number of elements in the array.

4.14 Packing Controls for Packed Data

Certain HSAIL instructions 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.

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

See [4.13.2. Packed Data Types \(previous page\)](#).

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

See the tables below.

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

Control	Description
<code>p</code>	The single source is treated as packed. The instruction is applied to each element separately.

Control	Description
<code>p_sat</code>	Same as <code>p</code> , except that each result is saturated. (Cannot be used with floating-point values.)
<code>s</code>	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.
<code>s_sat</code>	Same as <code>s</code> , except that the result is saturated. (Cannot be used with floating-point values.)

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

Control	Description
<code>pp</code>	Both sources are treated as packed. The instruction is applied pairwise to corresponding elements independently.
<code>pp_sat</code>	Same as <code>pp</code> , except that each result is saturated. (Cannot be used with floating-point values.)
<code>ps</code>	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 instruction 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.
<code>ps_sat</code>	Same as <code>ps</code> , except that each result is saturated. (Cannot be used with floating-point values.)
<code>sp</code>	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 instruction 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.
<code>sp_sat</code>	Same as <code>sp</code> , except that each result is saturated. (Cannot be used with floating-point values.)
<code>ss</code>	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.
<code>ss_sat</code>	Same as <code>ss</code> , except that the result is saturated. (Cannot be used with floating-point values.)

4.14.1 Ranges

For all packing controls, the following applies:

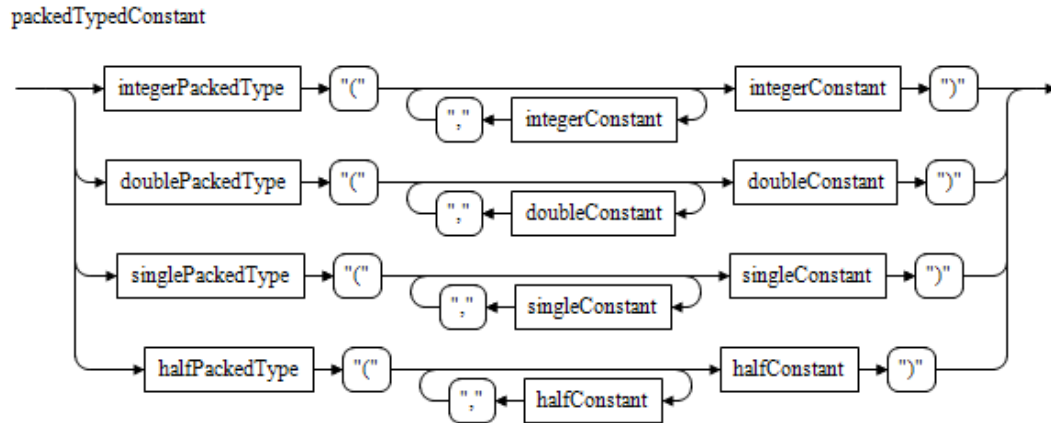
- For `u8x4`, `u8x8`, and `u8x16`, the range of an element is 0 to 255.
- For `s8x4`, `s8x8`, and `s8x16`, the range of an element is –128 to +127.
- For `u16x2`, `u16x4`, and `u16x8`, the range of an element is 0 to 65535.
- For `s16x2`, `s16x4`, and `s16x8`, the range of an element is –32768 to 32767.
- For `u32x2` and `u32x4`, the range of an element is 0 to 4294967295.
- For `s32x2` and `s32x4`, the range of an element is –2147483648 to 2147483647.
- For `u64x2`, the range of an element is 0 to 18446744073709551615.
- For `s64x2`, the range of an element is –9223372036854775808 to 9223372036854775807.

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 Type Constants

Figure 4–96 packedTypeConstant Syntax Diagram



HSAIL uses the typed constant notation for writing packed constant values: a packed type followed by a parenthesized list of constant values is converted to a single packed constant. The number of elements in the list must match the number of elements in the packed type. The packed element constants are ordered starting from most significant bit when loaded into a register. Therefore, the memory representation depends on the endianness of the platform.

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 \(page 83\)](#). The 64-bit packed floating-point type (*f64x2*) is not supported by the Base profile (see [16.2.1. Base Profile Requirements \(page 289\)](#)).

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 [4.8.5. How Text Format Constants Are Converted to Bit String Constants \(page 92\)](#).

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, 0x3f80000040000000;

```

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 instruction 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 value if the instruction accepts immediate operands. An immediate value can be either an integer constant, float constant, typed constant, or `WAVESIZE` according to the rules in [Table 4-1 \(page 92\)](#) based on the type of the operand. Note that image, sampler, and array typed constants are not allowed. See [4.8. Constants \(page 81\)](#) and [2.6.2. Wavefront Size \(page 30\)](#).
- Memory, image, segment checking, segment conversion, and `lda` instructions take an address expression as a source operand. See [4.18. Address Expressions \(page 106\)](#).
- Memory, image, and some copy (move) instructions allow vector operands as source and destination operands. These comprise a list of registers and, for source operands, immediate values. See [4.17. Vector Operands \(page 106\)](#).
- Branch instructions can take a label and list of labels as a source operand. See [8.1. Syntax \(page 227\)](#).
- Call instructions can take a function identifier, list of function identifiers, and signature identifier as a source operand. See [Chapter 10. Function Instructions \(page 243\)](#).

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

The destination operand of an instruction must be a register. It is denoted in the instruction 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 \(page 99\)](#). This defines the size and representation of the value provided by the source operand or stored in the destination operand.

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

For address expressions, the compound type refers to the value in memory, not the compound type of the address, which is always *u32* or *u64* according to the address size. See [Table 2-3 \(page 40\)](#).

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 [4.8.5. How Text Format Constants Are Converted to Bit String Constants \(page 92\)](#).

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* stores the value in the least significant 16 bits, and the most significant 16 bits are undefined. See [4.19.1. Floating-Point Numbers \(page 109\)](#).
- If the operand type is *u* or *s* with a size less than 32 bits then it must be an *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 instruction 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 *ld_u16* instruction 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` instruction 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` instruction 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 [5.19. Conversion \(cvt\) Instruction \(page 159\)](#).

4.17 Vector Operands

Several instructions 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 or immediate operand.

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 [4.8.5. How Text Format Constants Are Converted to Bit String Constants \(page 92\)](#).

In BRIG, the type of the vector operand is the type of each component operand. See [4.16. Operands \(page 104\)](#).

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 \(ld\) Instruction \(page 173\)](#).

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 instruction that uses an address expression operand specifies if it is a flat or segment address by the segment modifier on the instruction. 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 \(page 64\)](#).

Addresses are always in bytes. For information about how addresses are formed from an address expression, see [6.1.1. How Addresses Are Formed \(page 166\)](#).

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 \(ld\) Instruction \(page 173\)](#).

4.19 Floating Point

HSAIL provides a rich set of floating-point instructions. 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 \(page 289\)](#)) has been specified:
 - The 64-bit floating-point type ($\mathbb{F64}$) is not supported (see [16.2.1. Base Profile Requirements \(page 289\)](#)).
 - The DETECT and BREAK exception policies are not supported for the five floating point exceptions specified in [12.2. Hardware Exceptions \(page 269\)](#), therefore instructions do not have to generate them as they have no observable effect (see [4.19.5. Floating Point Exceptions \(page 112\)](#)).
- Floating-point values are stored in IEEE/ANSI Standard 754-2008 binary interchange format encoding. See [4.19.1. Floating-Point Numbers \(facing page\)](#).
- 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 \(page 269\)](#).
 - 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 Instructions \(page 260\)](#).
- Four IEEE/ANSI Standard 754-2008 rounding modes are supported for some floating-point instructions. See [4.19.2. Floating-Point Rounding \(facing page\)](#).
- The \mathbb{Ftz} (flush to zero) modifier, which forces subnormal values to zero, is supported on most instructions. See [4.19.3. Flush to Zero \(ftz\) \(page 110\)](#).
- Instructions that produce NaN results have certain requirements. See [4.19.4. Not A Number \(NaN\) \(page 111\)](#).
- Some instructions are fast approximations (the `nsqrt` instruction is an example). See [5.14. Native Floating-Point Instructions \(page 148\)](#).
- Many instructions that are not in the IEEE/ANSI Standard 754-2008 are provided.
- HSAIL supports saturating forms of floating-point to integer conversions. See [5.19.4. Description of Integer Rounding Modes \(page 162\)](#).
- HSAIL supports packed versions of some floating-point instructions.
 - The value for each element of the packed result is the same as would be produced by the non-packed version of the instruction, including handling of the \mathbb{Ftz} , rounding modifiers, and exceptions.
 - The exceptions generated by the packed instruction is the union of the exceptions generated for each element of the packed result.
- Some operations have a precision defined in terms of ULP rather than in terms of the correctly rounded result specified by IEEE/ANSI Standard 754-2008. See [4.19.6. Unit of Least Precision \(ULP\) \(page 112\)](#).

4.19.1 Floating-Point Numbers

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

- An `float` number is stored in memory and in an `s` register 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 `float` value stored in an `s` register occupies the least significant 16 bits of the register, and the most significant 16 bits are undefined.
- An `double` 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 `long double` 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 instructions.

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 Floating-Point Rounding

Four IEEE/ANSI Standard 754-2008 floating-point rounding modes are supported for some floating-point instructions:

- `up` specifies that result of the instruction should be rounded to positive infinity.
- `down` specifies that the result of the instruction should be rounded to negative infinity.
- `zero` specifies that result of the instruction should be rounded to zero.
- `near` specifies that result of the instruction 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 instruction supports a floating-point rounding mode, the default floating-point rounding mode specified by the module header is used. If the Base profile has been specified, the `round` modifier is not supported, and must always be omitted or it is an error. See [Chapter 14. module Header \(page 284\)](#) and [16.2.1. Base Profile Requirements \(page 289\)](#).

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 \(page 269\)](#)):

- As specified by IEEE/ANSI Standard 754-2008 Section 7.5, it is implementation defined if tininess (a tiny non-zero result) is detected before or after rounding, but an implementation must use the same method for all instructions.
- If the result is a NaN then the destination is set to a quiet NaN. See [4.19.4. Not A Number \(NaN\) \(facing page\)](#).
- 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 tininess 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 \(ftz\) \(below\)](#).
- 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 tininess 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 instructions that controls the flushing of source subnormal values and tiny results to zero.

If an instruction supports the `ftz` modifier then:

- If the Base profile has been specified then the `ftz` modifier must be specified. See [16.2.1. Base Profile Requirements \(page 289\)](#).
- Otherwise, the `ftz` modifier is optional.

If `ftz` is specified on an instruction that has floating-point source operands:

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

If `ftz` is specified on an instruction that has a floating-point destination operand:

- The instruction result before rounding is computed as defined by the IEEE/ANSI Standard 754-2008.
- If tininess is detected (see [4.19.2. Floating-Point Rounding \(page 109\)](#)), 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 instruction.
- Otherwise, the result is rounded according to the rounding modifier and stored in the destination operand. See [4.19.2. Floating-Point Rounding \(page 109\)](#).

4.19.4 Not A Number (NaN)

As required by IEEE/ANSI Standard 754-2008, for all floating-point instructions, except the floating-point bit instructions (see [5.13. Floating-Point Bit Instructions \(page 146\)](#)) and native floating-point instructions (see [5.14. Native Floating-Point Instructions \(page 148\)](#)):

- If one or more of the floating-point source operands is a signaling NaN, an invalid operation exception must be generated. Additionally, if the instruction is a signaling comparison form (see [5.18. Compare \(cmp\) Instruction \(page 155\)](#)) 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 \(page 269\)](#).
- If an instruction 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 instruction 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 instructions, 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 \(page 289\)](#).
- 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 signaling 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 \(page 290\)](#).
 - The `cvt` instruction is an exception to this rule when both the source and the destination are floating-point types. 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 signaling NaNs to quiet NaNs, except that the sign bits may differ.

The image instructions are an exception these rules, both when converting component values (see [7.1.4.2. Channel Type \(page 200\)](#)), and when using a sampler with normalized coordinates (see [7.1.6.1. Coordinate Normalization Mode \(page 206\)](#)) or a linear filter (see [7.1.6.3. Filter Mode \(page 209\)](#)). They must not generate an invalid operation exception for signaling 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 \(page 269\)](#)). It also provides a mechanism to control these exceptions by means of the DETECT and BREAK exception policies (see [12.3. Hardware Exception Policies \(page 271\)](#)). The exception policies are specified when a kernel is finalized and cannot be changed at runtime (see [13.4. Control Directives for Low-Level Performance Tuning \(page 278\)](#)). Whether either exception policy is supported by a kernel agent depends on the kernel agent and the profile specified (see [16.2. Profile-Specific Requirements \(page 289\)](#)).

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.

4.19.6 Unit of Least Precision (ULP)

Some operations have a precision defined in terms of ULP rather than in terms of the correctly rounded result specified by IEEE/ANSI Standard 754-2008. In addition, the precision of some operations varies according to the profile specified. See [16.2. Profile-Specific Requirements \(page 289\)](#).

The definition of *Units of least precision (ULP)* is the same as in *The OpenCL Specification Version 2.0*, which is based on Jean-Michel Muller's definition in *On the definition of ulp(x)*.

ulp(x) is defined as follows:

If x is a real number that lies between two finite consecutive floating-point numbers a and b , without being equal to one of them, then $ulp(x) = |b - a|$, otherwise $ulp(x)$ is the distance between the two non-equal finite floating-point numbers nearest x . Moreover, $ulp(\text{NaN})$ is NaN.

The maximum relative error of an operation can be expressed in terms of ULP. An operation is less than or equal to n ULP of the mathematically accurate result if, for all possible numeric source values of the operation:

$$|expected - actual| / ulp(expected) \leq n$$

where *expected* is the mathematically accurate result and *actual* is the result returned by the operation.

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

In order to provide equivalent functionality, 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 requirements of natural alignment and any `align` qualifier. For information about the `align` qualifier, see [4.3.10. Declaration and Definition Qualifiers \(page 69\)](#).

The base address of the kernarg segment variables for the currently executing kernel dispatch can be obtained by the `kernargbaseptr` instruction. 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 bytes and the maximum alignment of the kernel's kernarg segment variables.

HSA requires that the agent dispatching the kernel and the kernel agent 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 kernel agent 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 kernel agent, either by subdividing 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 [4.3.8. Variable \(page 64\)](#)).

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 kernel agent.

In the following code, the load (`ld`) instruction 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` instruction:

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

Such addresses must not be used in store instructions.

For more information, see [6.3. Load \(ld\) Instruction \(page 173\)](#) and [5.8. Copy \(Move\) Instructions \(page 130\)](#).

CHAPTER 5.

Arithmetic Instructions

This chapter describes the HSAIL arithmetic instructions.

5.1 Overview of Arithmetic Instructions

Unless stated otherwise, arithmetic instructions expect all inputs to be in registers or immediate values and to produce a single result in a register (see [4.16. Operands \(page 104\)](#)).

Consider this instruction:

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

In this case, the `max` instruction 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 instruction. For example, a `max` instruction could be signed integer, unsigned integer, or floating-point.

The length determines the size of the register used. In the descriptions of the instructions in this manual, a size `n` instruction expects all input registers to be of length `n` bits. For more information on the rules concerning operands, see [4.16. Operands \(page 104\)](#).

5.2 Integer Arithmetic Instructions

Integer arithmetic instructions 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 instructions.

Integer arithmetic instructions 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 instructions.

5.2.1 Syntax

Table 5–1 Syntax for Integer Arithmetic Instructions

Opcodes and Modifiers	Operands
<i>abs_sLength</i>	<i>dest, src0</i>
<i>add_TypeLength</i>	<i>dest, src0, src1</i>
<i>borrow_TypeLength</i>	<i>dest, src0, src1</i>
<i>carry_TypeLength</i>	<i>dest, src0, src1</i>
<i>div_TypeLength</i>	<i>dest, src0, src1</i>
<i>max_TypeLength</i>	<i>dest, src0, src1</i>
<i>min_TypeLength</i>	<i>dest, src0, src1</i>
<i>mul_TypeLength</i>	<i>dest, src0, src1</i>
<i>mulhi_TypeLength</i>	<i>dest, src0, src1</i>
<i>neg_sLength</i>	<i>dest, src0</i>
<i>rem_TypeLength</i>	<i>dest, src0, src1</i>
<i>sub_TypeLength</i>	<i>dest, src0, src1</i>

Explanation of Modifiers (see [Table 4–2 \(page 99\)](#))*Type*: s, u.*Length*: 32, 64.**Explanation of Operands (see [4.16. Operands \(page 104\)](#))***dest*: Destination register.*src0, src1*: Sources. Can be a register or immediate value.**Exceptions (see [Chapter 12. Exceptions \(page 269\)](#))**

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 5–2 Syntax for Packed Versions of Integer Arithmetic Instructions

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

Explanation of Modifiers (see [4.14. Packing Controls for Packed Data \(page 101\)](#))*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 \(page 100\)](#).

Explanation of Operands (see 4.16. Operands (page 104))
<i>dest</i> : Destination register.
<i>src0</i> , <i>src1</i> : Sources. Can be a register or immediate value.
Exceptions (see Chapter 12. Exceptions (page 269))
No exceptions are allowed.

For BRIG syntax, see [18.7.1.1. BRIG Syntax for Integer Arithmetic Instructions \(page 348\)](#).

5.2.2 Description

abs

The `abs` instruction 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_sLength` is valid.

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

add

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

borrow

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

carry

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

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

div

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

The `div` instruction 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 whether: 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 whether: no exception is generated; or an implementation defined exception is generated.

rem

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

The `rem` instruction 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 whether: no exception is generated; a divide by zero exception is generated; or some other implementation defined exception is generated.

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

max

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

min

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

mul

The `mul` instruction 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(-2^{31} , -1)` returns -2^{31} for 32-bit operands. `mul(-2^{63} , -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` instruction. See [5.8. Copy \(Move\) Instructions \(page 130\)](#).

neg

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

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

sub

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

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

Examples of Regular (Nonpacked) Instructions

```
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 Instructions

```
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 Instruction

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 Instructions \(next page\)](#).

5.3.1 Syntax

Table 5–3 Syntax for Integer Optimization Instruction

Opcode and Modifiers	Operands
<i>mad_TypeLength</i>	<i>dest, src0, src1, src2</i>
Explanation of Modifiers (see Table 4–2 (page 99))	
<i>Type</i> : s, u.	
<i>Length</i> : 32, 64.	
Explanation of Operands (see 4.16. Operands (page 104))	
<i>dest</i> : Destination register.	
<i>src0, src1, src2</i> : Sources. Can be a register or immediate value.	
Exceptions (see Chapter 12. Exceptions (page 269))	
No exceptions are allowed.	

For BRIG syntax, see [18.7.1.2. BRIG Syntax for Integer Optimization Instruction \(page 349\)](#).

Description

The integer *mad* (multiply add) instruction 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: $((s0 * s1) + s2) \& ((1 \ll \text{length}) - 1)$.

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 Instructions

Integer optimizations are intended to improve performance. High-level compilers should attempt to generate these whenever possible. These instructions 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 instructions if it does not support native 24-bit signed instructions.

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 instructions if it does not support native 24-bit unsigned instructions.

See also [5.3. Integer Optimization Instruction \(previous page\)](#).

5.4.1 Syntax

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

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

Explanation of Modifiers (see [Table 4–2 \(page 99\)](#))

Type: s, u

Length: 32

Explanation of Operands (see [4.16. Operands \(page 104\)](#))

dest: Destination register.

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

Exceptions (see [Chapter 12. Exceptions \(page 269\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.1.3. BRIG Syntax for 24-Bit Integer Optimization Instructions \(page 349\)](#).

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 Instructions

These instructions perform right or left shifts of bits.

These instructions have a packed form.

5.5.1 Syntax

Table 5–5 Syntax for Integer Shift Instructions

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 4–2 (page 99))

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 (page 104))

dest: Destination register.

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

Exceptions (see Chapter 12. Exceptions (page 269))

No exceptions are allowed.

For BRIG syntax, see 18.7.1.4. BRIG Syntax for Integer Shift Instructions (page 349).

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* instruction 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 Instructions

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

5.6.1 Syntax

Table 5–6 Syntax for Individual Bit Instructions

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

Explanation of Modifiers (see Table 4–2 (page 99))
<i>Type</i> : b
<i>Length</i> : 1, 32, 64; popcount does not support b1.

Explanation of Operands (see 4.16. Operands (page 104))
<i>dest</i> : Destination register.
<i>src0, src1</i> : Sources. Can be a register or immediate value.

Exceptions (see Chapter 12. Exceptions (page 269))
No exceptions are allowed.

For BRIG syntax, see [18.7.1.5. BRIG Syntax for Individual Bit Instructions \(page 349\)](#).

Description

The b1 form is used with control (c) register sources. It can only be used with the instructions `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` instruction 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` instruction 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` instruction 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 5-7 \(below\)](#).

Table 5-7 Inputs and Results for popcount Instruction

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

Examples

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

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

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

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

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

5.7 Bit String Instructions

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

5.7.1 Syntax

Table 5–8 Syntax for Bit String Instructions

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 4–2 \(page 99\)](#))

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 \(page 104\)](#))

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

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

Exceptions (see [Chapter 12. Exceptions \(page 269\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.1.6. BRIG Syntax for Bit String Instructions \(page 350\)](#).

Description

`bitextract`

Extracts a range of bits.

src0 and *dest* are treated as the *TypeLength* of the instruction. *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 instruction.type
}

```

bitinsert

Replaces a range of bits.

src0, *src1*, and *dest* are treated as the *TypeLength* of the instruction. *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* instruction 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 instruction. *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 instruction 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 `0x01ffffff` (seven 0's followed by 25 1's) returns 7

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

Length applies only to the source.

See this pseudocode:

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

See [Table 5-9 \(next page\)](#).

`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 5-9 \(next page\)](#).

`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` instruction 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 5–9 \(below\)](#).

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

Input	Result for firstbit	Result for lastbit
00000000	-1	-1
00ffffff	8	0
7ffffff	1	0
01ffffff	7	0
ffffffff	0	0
ffff0f00	0	8

Examples

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

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

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

bitmask_b32 $s0, $s1, $s2;

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

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

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

5.8 Copy (Move) Instructions

These instructions 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 \(page 289\)](#)).

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 \(page 39\)](#)).

5.8.1 Syntax

Table 5–10 Syntax for Copy (Move) Instructions

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

Explanation of Modifiers

segment: Optional segment: global, group, private, kernarg or readonly. If omitted, flat is used. See [2.8. Segments \(page 31\)](#).

Length: 1, 32, 64, 128 (see [Table 4–2 \(page 99\)](#)). For `lda` must match the address size (see [Table 2–3 \(page 40\)](#)).

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 \(page 39\)](#) and [16.2.1. Base Profile Requirements \(page 289\)](#).

Explanation of Operands (see [4.16. Operands \(page 104\)](#))

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

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

address: An address expression. See [4.18. Address Expressions \(page 106\)](#).

Exceptions (see [Chapter 12. Exceptions \(page 269\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.1.7. BRIG Syntax for Copy \(Move\) Instructions \(page 350\)](#).

Description**combine**

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

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

expand

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

This instruction has a 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 instruction 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 \(page 166\)](#). The size of *dest* must match the address size of the segment. See [Table 2–3 \(page 40\)](#).

The address of a kernel or function cannot be taken. The HSA runtime can be used to obtain kernel and indirect function code handles. The `scall` instruction can be used to achieve the equivalent of indirect calls.

The address of a label cannot be taken. The `sbr` instruction 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 instruction can also be used to take the byte address of a kernel's formal arguments in the kernarg segment.

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

mov

Copies a value of type *moveType* from source *src0* 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 Instructions \(page 216\)](#) and [6.8. Notification \(signal\) Instructions \(page 187\)](#)).

If *moveType* is `fl16`, the most significant 16 bits of the destination *s* register are undefined. If the source is also an *s* register, then it is not required that the most significant 16 bits of the destination match the most significant 16 bits of the source. See [4.19.1. Floating-Point Numbers \(page 109\)](#).

5.8.2 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 instruction. `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;

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

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 Instructions

These instructions perform shuffle, interleave, pack, and unpack operations on packed data. In addition, many of the integer and floating-point instructions support packed data as does the `cmp` instruction.

If the Base profile has been specified then the 64-bit packed floating-point type ($2 \times f64$) is not supported (see [16.2.1. Base Profile Requirements \(page 289\)](#)).

See also:

- [5.2. Integer Arithmetic Instructions \(page 116\)](#)
- [5.11. Floating-Point Arithmetic Instructions \(page 140\)](#)
- [5.18. Compare \(cmp\) Instruction \(page 155\)](#)

See [Table 5–11 \(below\)](#) and [Table 5–12 \(next page\)](#).

5.9.1 Syntax

Table 5–11 Syntax for Shuffle and Interleave Instructions

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 (page 100))
Type: s, u, f.
Length: 8x4, 8x8, 16x2, 16x4, 32x2

Explanation of Operands (see 4.16. Operands (page 104))
<code>dest</code> : Destination. See the Description below.
<code>src0, src1</code> : Sources. Must be a packed register an immediate value.
<code>src2</code> : Source. Must be a constant value used to select elements. <code>WAVESIZE</code> is not allowed. See Table 5–13 (page 136) .

Exceptions (see Chapter 12. Exceptions (page 269))
No exceptions are allowed.

For BRIG syntax, see [18.7.1.8. BRIG Syntax for Packed Data Instructions \(page 350\)](#).

Table 5–12 Syntax for Pack and Unpack Instructions

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 4–2 (page 99) , Table 4–3 (page 100) and 16.2.1. Base Profile Requirements (page 289) .

Explanation of Operands (see 4.16. Operands (page 104))
<i>dest</i> : Destination register.
<i>src0, src1, src2</i> : Sources. Can be a register or immediate value.

Exceptions (see Chapter 12. Exceptions (page 269))
No exceptions are allowed.

For BRIG syntax, see [18.7.1.8. BRIG Syntax for Packed Data Instructions \(page 350\)](#).

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 [5.9.2. Controls in src2 for shuffle Instruction \(page 136\)](#).

unpacklo

Copies and interleaves the lower half of the elements from each source into the destination. See [5.9.4. Examples of unpacklo and unpackhi Instructions \(page 139\)](#).

unpackhi

Copies and interleaves the upper half of the elements from each source into the destination. See [5.9.4. Examples of unpacklo and unpackhi Instructions \(page 139\)](#).

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 *srcType**srcLength*.

See [4.16. Operands \(page 104\)](#). 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, and the least significant 16 bits are used. See [4.19.1. Floating-Point Numbers \(page 109\)](#).

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 (page 104). 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 is stored in the least significant 16 bits and the most significant 16 bits are undefined. See 4.19.1. Floating-Point Numbers (page 109).

5.9.2 Controls in *src2* for shuffle Instruction

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 5–13 Bit Selectors for shuffle instruction

src2 Bits for Packed Data Types s8x4 and u8x4	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

5.9.3 Common Uses for shuffle Instruction

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

Broadcast

Broadcast the least significant data element into the destination:

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

src2 is the constant 00 00 00 00 in bits.

Broadcast the second data element into the destination:

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

src2 is the constant 01 01 01 01 in bits.

Broadcast the third data element into the destination:

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

src2 is the constant 10 10 10 10 in bits.

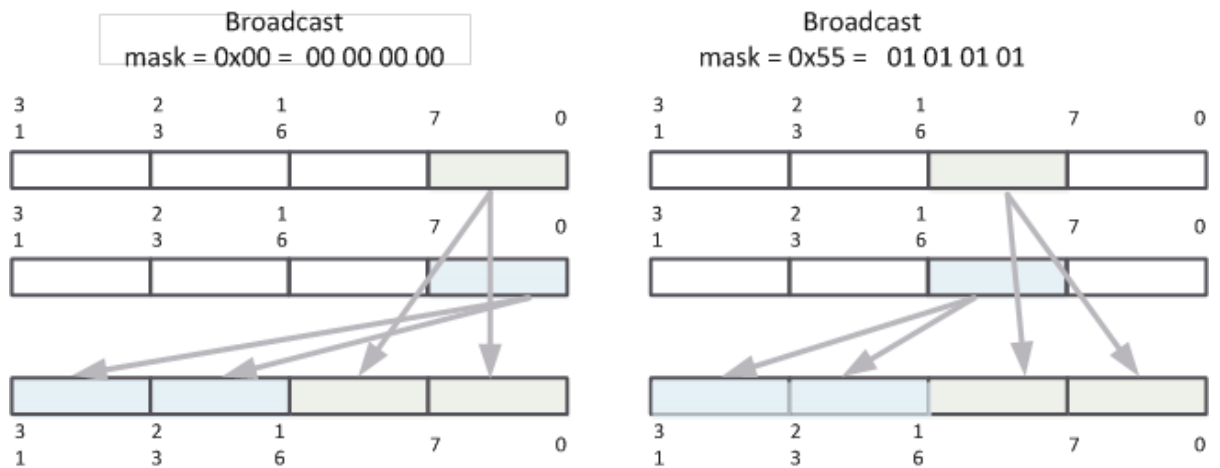
Broadcast the most significant data element into the destination:

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

src2 is the constant 11 11 11 11 in bits.

See the figure below.

Figure 5-1 Example of Broadcast

**Swap**

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

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

src2 is the constant 00 01 10 11 in bits.

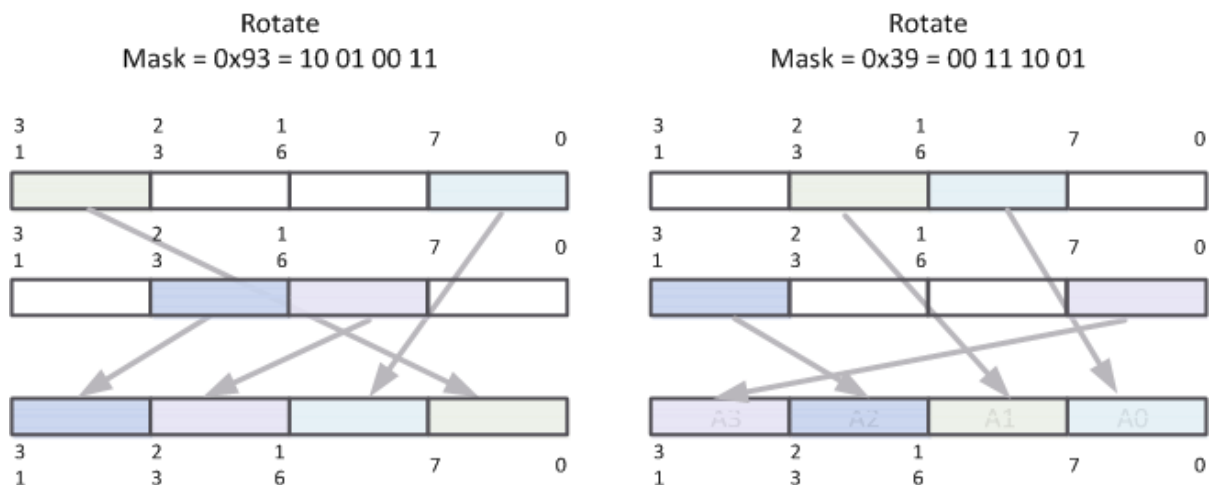
Rotate

To rotate:

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

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

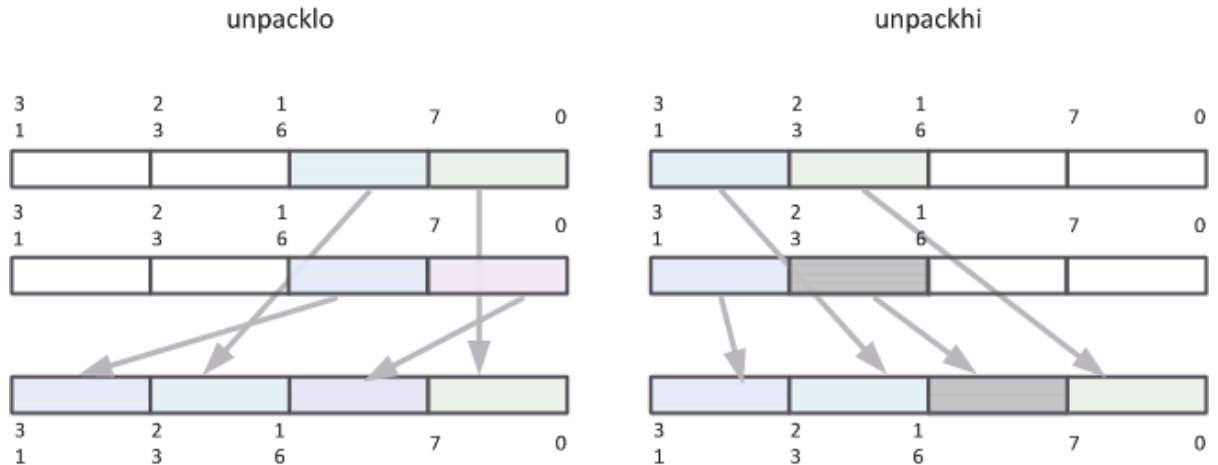
Figure 5-2 Example of Rotate



5.9.4 Examples of unpacklo and unpackhi Instructions

See the figure below.

Figure 5–3 Example of Unpack



Examples

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

```
// Packing with no conversions:
pack_f32x2_f32 $d1, $d1, $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_s16 $d1, $d1, $s2, 0;
pack_u32x2_u32 $d1, $d2, $s3, 0;
```

```
// Packing an f16:
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_u32_u32x4 $s1, $q1, 2;
unpack_s32_s32x2 $s1, $d2, 0;
```

```
// Unpacking an f16:
unpack_f16_f16x2 $s1, $s2, 1;
unpack_f16_f16x4 $s1, $d2, 3;
```

5.10 Bit Conditional Move (cmov) Instruction

The `cmov` instruction performs a bit conditional move.

There is a packed form of this instruction.

5.10.1 Syntax

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

Opcode and Modifiers	Operands
<code>cmov_TypeLength</code>	<code>dest, src0, src1, src2</code>
Explanation of Modifiers (see Table 4–2 (page 99))	
<i>Type</i> : For the regular form: b. For the packed form: s, u, f.	
<i>Length</i> : For the regular form, <i>Length</i> can be 1, 32, 64. Applies to <i>src1</i> , and <i>src2</i> . For the packed form, <i>Length</i> can be any packed type.	
Explanation of Operands (see 4.16. Operands (page 104))	
<i>dest</i> : Destination register. For the packed form, if the length is 32 bits, then <i>dest</i> must be an <i>s</i> register; if the length is 64 bits, then <i>dest</i> must be a <i>d</i> register; if the length is 128 bits, then <i>dest</i> must be a <i>q</i> register.	
<i>src0, src1, src2</i> : Sources. For the regular form, <i>src0</i> must be a control (<i>c</i>) register or an immediate value and is of type <i>b1</i> . For the packed form, if the <i>Length</i> is 32 bits, then <i>src0</i> must be an <i>s</i> register or immediate value of type <i>uLength</i> ; if the <i>Length</i> is 64 bits, then <i>src0</i> must be a <i>d</i> register or immediate value of type <i>uLength</i> ; if the <i>Length</i> is 128 bits, then <i>src0</i> must be a <i>q</i> register or immediate value of type <i>uLength</i> .	
Exceptions (see Chapter 12. Exceptions (page 269))	
No exceptions are allowed.	

For BRIG syntax, see [18.7.1.9. BRIG Syntax for Bit Conditional Move \(cmov\) Instruction \(page 351\)](#).

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 Instructions

These instructions perform floating-point arithmetic and follow the IEEE/ANSI Standard 754-2008. However, there are some important differences. See [4.19. Floating Point \(page 107\)](#).

5.11.1 Syntax

Table 5-15 Syntax for Floating-Point Arithmetic Instructions

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_round_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 \(ftz\) \(page 110\)](#).

round: Optional rounding mode. Possible values are *up*, *down*, *zero*, or *near*. If the Base profile has been specified, then must be omitted. If omitted, the default floating-point rounding mode specified by the module header is used. See [Chapter 14. module Header \(page 284\)](#).

Type: *f*. See [Table 4-2 \(page 99\)](#).

Length: 16, 32, and, if the Base profile has not been specified, 64. See [Table 4-2 \(page 99\)](#) and [16.2.1. Base Profile Requirements \(page 289\)](#).

Explanation of Operands (see [4.16. Operands \(page 104\)](#))

dest: Destination register.

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

Exceptions (see [Chapter 12. Exceptions \(page 269\)](#))

Floating-point exceptions are allowed.

Table 5-16 Syntax for Packed Versions of Floating-Point Arithmetic Instructions

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_round_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
<i>ftz</i> : 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) (page 110) .
<i>round</i> : Optional up, down, zero, or near rounding mode. If the Base profile has been specified, then must be omitted. If omitted, the default floating-point rounding mode specified by the module header is used. See Chapter 14. module Header (page 284) .
<i>Control</i> for ceil, floor, fract, rint, sqrt, and trunc: p or s. <i>Control</i> for add, div, max, min, mul, and sub: pp, ps, sp, or ss. See 4.14. Packing Controls for Packed Data (page 101) .
<i>TypeLength</i> : f16x2, f16x4, f16x8, f32x2, f32x4, and, if the Base profile has not been specified, f64x2. See 4.13.2. Packed Data Types (page 100) and 16.2.1. Base Profile Requirements (page 289) .

Explanation of Operands (see 4.16. Operands (page 104))
<i>dest</i> : Destination register.
<i>src0</i> , <i>src1</i> : Sources. Can be a register or immediate value.

Exceptions (see Chapter 12. Exceptions (page 269))
Floating-point exceptions are allowed.

For BRIG syntax, see [18.7.1.10. BRIG Syntax for Floating-Point Arithmetic Instructions \(page 351\)](#).

Description

add

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

ceil

Rounds the floating-point source *src0* toward positive infinity to produce a floating-point integral number that is assigned to the destination *dest*. If the source has an infinity value, the result will be the same infinity value. No exceptions are generated besides invalid operation for a signaling NaN source.

div

Performs the IEEE/ANSI Standard 754-2008 standard floating-point divide. Computes source *src0* divided by source *src1* and stores the result in the destination *dest*.

div must return a correctly rounded result in the Full profile and return a result less than or equal to 2.5 ULP (see [4.19.6. Unit of Least Precision \(ULP\) \(page 112\)](#)) of the mathematically accurate value in the Base profile. See [Chapter 16. Profiles \(page 288\)](#).

floor

Rounds the floating-point source *src0* toward negative infinity to produce a floating-point integral number that is assigned to the destination *dest*. If the source has an infinity value, the result will be the same infinity value. No exceptions are generated besides invalid operation for a signaling NaN source.

fma

The floating-point *fma* (fused multiply add) computes $src0 * src1 + src2$ with unbounded range and precision. The resulting value is then rounded once using the specified rounding mode.

No underflow, overflow, or inexact exception can be generated for the multiply. However, these exceptions can be generated by the addition. Thus, `fma` differs from a `mul` followed by an `add`.

`fma` is not supported as a packed operation, because it takes three source operands.

`fract`

Sets the destination `dest` to the fractional part of source `src0`.

```
src0' = ftz ? flush_subnormal_to_zero(src0) : src0
dest = (src0' == +0.0) ? +0.0
      : (src0' == -0.0) ? -0.0
      : (src0' == +inf) ? +0.0
      : (src0' == -inf) ? -0.0
      : (isNaN(src0')) ? NaNsrc0'
      : min(roundround_modifier, TypeLength(src0' - smallest_numericTypeLength), smallest_numericTypeLength)
```

where:

```
smallest_numericf16 = 0x1.ffcp-1h
smallest_numericf32 = 0x1.ffffep-1f
smallest_numericf64 = 0x1.fffffffffffp-1d
```

The `min` is used to ensure that the result of the `fract` operation of a small negative number is not 1.0 so that the result is in the half-open interval [0.0, 1.0).

NaN inputs are handled as described in [4.19.4. Not A Number \(NaN\) \(page 111\)](#).

`max`

Computes the maximum of source `src0` and source `src1` and stores the result in the destination `dest`.

`max` implements the `maxNum` operation as described in IEEE/ANSI Standard 754-2008. If one of the inputs is a quiet NaN and the other input is not a NaN, then the non-NaN input is returned; otherwise NaN inputs are handled as described in [4.19.4. Not A Number \(NaN\) \(page 111\)](#).

`min`

Computes the minimum of source `src0` and source `src1` and stores the result in the destination `dest`.

`min` implements the `minNum` operation as described in IEEE/ANSI Standard 754-2008. If one of the inputs is a quiet NaN and the other input is not a NaN, then the non-NaN input is returned; otherwise NaN inputs are handled as described in [4.19.4. Not A Number \(NaN\) \(page 111\)](#).

`mul`

Multiplies source `src0` by source `src1` (following IEEE/ANSI Standard 754-2008 rules) and stores the result in the destination `dest`.

`rint`

Rounds the floating-point source `src0` toward the nearest integral number, choosing the even integral value if there is a tie, to produce a floating-point integral number that is assigned to the destination `dest`. If the source has an infinity value, the result will be the same infinity value. No exceptions are generated besides invalid operation for a signaling NaN source.

sub

Subtracts source *src1* from source *src0* and places the result in the destination *dest*. The answer is computed according to IEEE/ANSI Standard 754-2008 rules.

sqr

Sets the destination *dest* to the square root of source *src0*.

If *src0* is negative, must return a quiet NaN and generate the invalid operation exception.

sqr returns the correctly rounded result for the Full profile and a result less than or equal to 1 ULP (see 4.19.6. Unit of Least Precision (ULP) (page 112)) of the mathematically accurate value for the Base profile. See Chapter 16. Profiles (page 288).

trunc

Rounds the floating-point source *src0* toward zero to produce a floating-point integral number that is assigned to the destination *dest*. If the source has an infinity value, the result will be the same infinity value. No exceptions are generated besides invalid operation for a signaling NaN source.

Examples of Regular (Nonpacked) Instructions

```
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 Instructions

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

5.12 Floating-Point Optimization Instruction

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

5.12.1 Syntax

Table 5–17 Syntax for Floating-Point Optimization Instruction

Opcode and Modifiers	Operands
mad_ftz_round_TypeLength	dest, src0, src1, src2

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) (page 110).

Explanation of Modifiers
<i>round</i> : Optional rounding mode. Possible values are <i>up</i> , <i>down</i> , <i>zero</i> , or <i>near</i> . If the Base profile has been specified, then must be omitted. If omitted, the default floating-point rounding mode specified by the module header is used. See Chapter 14. module Header (page 284) .
<i>Type</i> : <i>f</i> . See Table 4-2 (page 99) .
<i>Length</i> : 16, 32, and, if the Base profile has not been specified, 64. See Table 4-2 (page 99) and 16.2.1. Base Profile Requirements (page 289) .
Explanation of Operands (see 4.16. Operands (page 104))
<i>dest</i> : Destination register.
<i>src0</i> , <i>src1</i> , <i>src2</i> : Sources. Can be a register or immediate value.
Exceptions (see Chapter 12. Exceptions (page 269))
Floating-point exceptions are allowed.

For BRIG syntax, see [18.7.1.11. BRIG Syntax for Floating-Point Optimization Instruction \(page 352\)](#).

Description

The floating-point `mad` (multiply add) instruction multiplies source *src0* times source *src1* and then adds source *src2*. The result is stored in the destination *dest*. The computation must be performed using the semantic equivalent of one of the following methods:

- *Single Round Method*:

```
fma_ftz_round_fLength dest, src0, src1, src2;
```

- *Double Round Method*:

```
mul_ftz_round_fLength temp, src0, src1;
add_ftz_round_fLength dest, temp, src2;
```

Where each instruction uses the same modifiers and type as the `mad` instruction.

No alternative method is allowed.

The same method must be used for all floating-point `mad` instructions on a specific kernel agent. An HSA runtime query is available to determine the method used on a kernel agent.

The floating-point `mad` instruction enables high level compilers to generate a contracted multiply-addition without prescribing whether single or double rounding behavior should be used. This allows the finalizer for a kernel agent to generate either a separate multiply and addition with intermediate rounding, or an `fma` instruction without intermediate rounding, depending on which approach has better performance in terms of speed or power.

Floating-point `mad` is not supported as a packed instruction, because it takes three source operands.

Examples

```
mad_f16 $s1, $s2, $s3, $s5;
mad_f32 $s1, $s2, $s3, $s5;
mad_f64 $d1, $d2, $d3, $d2;
```

5.13 Floating-Point Bit Instructions

These instructions are performed as floating-point bit operations and follow the IEEE/ANSI Standard 754-2008. See [4.19. Floating Point \(page 107\)](#).

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.

5.13.1 Syntax

Syntax for Floating-Point Bit Instructions

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 \(page 104\)](#)) (see [Table 4–2 \(page 99\)](#))

Type: `f`. See [Table 4–2 \(page 99\)](#).

Length: 16, 32, and, if the Base profile has not been specified, 64. See [Table 4–2 \(page 99\)](#) and [16.2.1. Base Profile Requirements \(page 289\)](#).

Explanation of Operands (see [4.16. Operands \(page 104\)](#))

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 5–18 \(below\)](#).

Table 5–18 Class Instruction 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 \(page 269\)](#))

No exceptions are allowed.

Table 5–19 Syntax for Packed Versions of Floating-Point Bit Instructions

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
<i>Control</i> for <code>abs</code> , and <code>neg</code> : p or s. <i>Control</i> for <code>copysign</code> : pp, ps, sp, or ss. See 4.14. Packing Controls for Packed Data (page 101). <i>TypeLength</i> : f16x2, f16x4, f16x8, f32x2, f32x4, and, if the Base profile has not been specified, f64x2. See 4.13.2. Packed Data Types (page 100) and 16.2.1. Base Profile Requirements (page 289).

Explanation of Operands (see 4.16. Operands (page 104))
<i>dest</i> : Destination register.
<i>src0, src1</i> : Sources. Can be a register or immediate value.

Exceptions (see Chapter 12. Exceptions (page 269))
No exceptions are allowed.

For BRIG syntax, see 18.7.1.5. BRIG Syntax for Individual Bit Instructions (page 349).

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 5–18 (previous page) 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.

Note that `neg(x)` is not the same as `sub(+0.0, x)`. In addition to having no effects on the exception state, `neg(+0.0)` is -0.0 and `neg(-0.0)` is +0.0, while `sub(+0.0, x)` is always +0.0 when *x* is either +0.0 or -0.0.

Examples

```
abs_f32 $s1,$s2;
abs_f64 $d1,$d2;
class_bl_f32 $c1, $s1, 3;
class_bl_f32 $c1, $s1, $s2;
class_bl_f64 $c1, $d1, $s2;
class_bl_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 Instructions

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

5.14 Native Floating-Point Instructions

The floating-point native instructions produce fast approximate implementation dependent 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 instructions 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 instructions 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 \(page 107\)](#).

5.14.1 Syntax

Table 5–20 Syntax for Native Floating-Point Instructions

Opcode and Modifiers	Operands
ncos_f32	<i>dest, src</i>
nexp2_f32	<i>dest, src</i>
nfma_TypeLength	<i>dest, src0, src1, src2</i>
nlog2_f32	<i>dest, src</i>
nrCP_TypeLength	<i>dest, src</i>
nrsqrt_TypeLength	<i>dest, src</i>
nsin_f32	<i>dest, src</i>
nsqrt_TypeLength	<i>dest, src</i>

Explanation of Modifiers (see [Table 4–2 \(page 99\)](#))

Type: f.

Length: 16, 32, and, if the Base profile has not been specified, 64. See [16.2.1. Base Profile Requirements \(page 289\)](#).

Explanation of Operands (see 4.16. Operands (page 104))
<i>dest</i> : Destination register.
<i>src, src0, src1, src2</i> : Sources. Can be a register or immediate value.
Exceptions (see 12.2. Hardware Exceptions (page 269))
Standard floating-point exceptions are allowed.

For BRIG syntax, see 18.7.1.13. BRIG Syntax for Native Floating-Point Instructions (page 353).

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.

`nsqrt`

Computes the square root.

Examples

```
ncos_f32 $s1, $s0;

nexp2_f32 $s1, $s0;

nfma_f32 $s3, 1.0f, $s1, 23.0f;
nfma_f64 $d3, 1.0D, $d0, $d3;

nlog2_f32 $s1, $s0;

nrcp_f32 $s1, $s0;
```

```
nrsqrt_f32 $s1, $s0;
nsin_f32 $s1, $s0;
```

5.15 Multimedia Instructions

These instructions support fast multimedia operations. The instructions work on special packed formats that have up to four values packed into a single 32-bit register.

5.15.1 Syntax

Table 5–21 Syntax for Multimedia Instructions

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 (page 104))

`dest`: The destination must be an `s` register.

`src0, src1, src2, src3`: Sources. Can be a register or immediate value, except `src1` for `unpackcvt` must be a constant with value 0, 1, 2, or 3. (WAVESIZE is not allowed.)

Exceptions (see Chapter 12. Exceptions (page 269))

No exceptions are allowed.

For BRIG syntax, see 18.7.1.14. BRIG Syntax for Multimedia Instructions (page 353).

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_t 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) computes the unsigned 8-bit average of packed data. Useful in multimedia applications that use unsigned 8-bit packed data to represent pixels.

Treating the sources as four 8-bit packed unsigned values, this instruction 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_near_sat_u8_f32(src0))) << 0) |
        (((uint32_t)(cvt_near_sat_u8_f32(src1))) << 8) |
        (((uint32_t)(cvt_near_sat_u8_f32(src2))) << 16) |
        (((uint32_t)(cvt_near_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 a constant `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;

```

5.16 Segment Checking (*segmentp*) Instruction

The *segmentp* instruction tests whether or not a given flat address is within a specific memory segment.

See also [5.17. Segment Conversion Instructions \(next page\)](#).

5.16.1 Syntax

Table 5–22 Syntax for Segment Checking (*segmentp*) Instruction

Opcode and Modifiers	Operands
<code>segmentp_segment_nonull_b1_srcType</code> <i>srcLength</i>	<i>dest, src</i>
Explanation of Modifiers (see Table 4–2 (page 99))	
<i>segment</i> : Can be global, group or private. See 2.8. Segments (page 31) .	
<i>nonnull</i> : Optional. If present, indicates that the <i>src</i> operand will not be the <code>nullptr</code> address value for the <i>segment</i> . See the Description below.	
<i>srcType</i> : u.	
<i>srcLength</i> : 32, 64. The size of the source address. Must match the address size of flat addresses. See Table 2–3 (page 40) .	
Explanation of Operands (see 4.16. Operands (page 104))	
<i>dest</i> : Destination register. Must be a control (c) register.	
<i>src</i> : Source for the flat address that is being checked. Can be a register or immediate value. See Table 2–3 (page 40) .	
Exceptions (see Chapter 12. Exceptions (page 269))	
No exceptions are allowed.	

For BRIG syntax, see [18.7.1.15. BRIG Syntax for Segment Checking \(*segmentp*\) Instruction \(page 353\)](#).

Description

This instruction 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 \(page 39\)](#).

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 \(page 296\)](#).

See [2.8.4. Memory Segment Access Rules \(page 36\)](#).

Examples

```
segmentp_private_b1_u32 $c1, $s0;      // small machine model
segmentp_global_b1_u32 $c1, $s0;      // small machine model
segmentp_global_nonnull_b1_u32 $c1, $s0; // small machine model
segmentp_group_b1_u64 $c1, $d0;       // large machine model
```

5.17 Segment Conversion Instructions

The segment conversion instructions convert a flat address into a segment address, or a segment address into a flat address.

See also [5.16. Segment Checking \(segmentp\) Instruction \(previous page\)](#).

5.17.1 Syntax

Table 5–23 Syntax for Segment Conversion Instructions

Opcodes and Modifiers	Operands
ftos <i>segment_nonnull_destType</i> <i>destLength_srcType</i> <i>srcLength</i>	<i>dest</i> , <i>src</i>
stof <i>segment_nonnull_destType</i> <i>destLength_srcType</i> <i>srcLength</i>	<i>dest</i> , <i>src</i>

Explanation of Modifiers
<i>segment</i> : group or private. See 2.8. Segments (page 31) .
<i>nonnull</i> : Optional. If present, indicates that the <i>src</i> operand will not be the <code>nullptr</code> address value for the <i>segment</i> . See the Description below.
<i>destType</i> : u. See Table 4–2 (page 99) .
<i>destLength</i> : 32, 64. The size of the destination address. For <i>ftos</i> , must be the address size of <i>segment</i> ; for <i>stof</i> , must be the flat address size. See Table 2–3 (page 40) .
<i>srcType</i> : u. See Table 4–2 (page 99) .
<i>srcLength</i> : 32, 64. The size of the source address. For <i>ftos</i> , must be the flat address size; for <i>stof</i> , must be the address size of <i>segment</i> . See Table 2–3 (page 40) .

Explanation of Operands (see 4.16. Operands (page 104))
<i>dest</i> : Destination register.
<i>src</i> : Source to be converted. Can be a register or immediate value.

Exceptions (see Chapter 12. Exceptions (page 269))
No exceptions are allowed.

For BRIG syntax, see [18.7.1.16. BRIG Syntax for Segment Conversion Instructions \(page 354\)](#).

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 \(page 39\)](#).

The global segment is not supported as there is no conversion required from a flat address that references the global segment and a global segment address since the values are the same. See [2.8.3. Addressing for Segments \(page 35\)](#).

If the source is not in the specified segment, the result is undefined. See [2.8.4. Memory Segment Access Rules \(page 36\)](#).

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 \(page 296\)](#).

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 \(page 39\)](#).

The global segment is not supported as no conversion is required from a global segment address to a flat address since the values are the same. See [2.8.3. Addressing for Segments \(page 35\)](#).

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 \(page 296\)](#).

Examples

```
// large machine model conversions
stof_private_u64_u32 $d1, $s1;
stof_private_nonnull_u64_u32 $d1, $s1;
ftos_group_u32_u64 $s1, $d2;
ftos_group_nonnull_u32_u64 $s1, $d2;

// small machine model conversions
stof_private_u32_u32 $s1, $s2;
stof_private_nonnull_u32_u32 $s1, $s2;
ftos_group_u32_u32 $s1, $s2;
ftos_group_nonnull_u32_u32 $s1, $s2;
```

5.18 Compare (cmp) Instruction

The compare (cmp) instruction compares two numeric values. The value written depends on the type of destination *dest*.

cmp compares register-sized values, with one exception: for *f16* register operands, cmp uses the floating point value stored in the least significant 16 bits and ignores the most significant 16 bits. See [4.19.1. Floating-Point Numbers \(page 109\)](#).

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 \(page 107\)](#).

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 instruction is a signaling 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 \(page 269\)](#).

The `ftz` modifier is supported if the source operand type is floating-point. See [4.19.3. Flush to Zero \(ftz\) \(page 110\)](#).

See [Table 5-24 \(below\)](#) and [Table 5-25 \(below\)](#).

5.18.1 Syntax

Table 5-24 Syntax for Compare (cmp) Instruction

Opcode and Modifiers	Operands
<code>cmp_op_ftz_destType destLength_srcType srcLength</code>	<code>dest, src0, src1</code>
Explanation of Modifiers (see Table 4-2 (page 99))	
<i>op</i> for bit types: <code>eq</code> and <code>ne</code> .	
<i>op</i> for integer source types: <code>eq</code> , <code>ne</code> , <code>lt</code> , <code>le</code> , <code>gt</code> , <code>ge</code> .	
<i>op</i> 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 NaN forms <code>seq</code> , <code>sne</code> , <code>slt</code> , <code>sle</code> , <code>sgt</code> , <code>sge</code> , <code>sequ</code> , <code>sneu</code> , <code>sltu</code> , <code>sleu</code> , <code>sgtu</code> , <code>sgeu</code> , <code>snum</code> , <code>snan</code> .	
<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) (page 110) .	
<i>destType destLength</i> : Describes the destination.	
<i>destType</i> : <code>u</code> , <code>s</code> , <code>f</code> ; <code>b</code> if <i>destLength</i> is 1.	
<i>destLength</i> : 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 <i>destType</i> is <code>f</code> . See 16.2.1. Base Profile Requirements (page 289) .	
<i>srcType srcLength</i> : Describes the two sources.	
<i>srcType</i> : <code>b</code> , <code>u</code> , <code>s</code> , <code>f</code> .	
<i>srcLength</i> : 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 <i>srcType</i> is <code>f</code> . See 16.2.1. Base Profile Requirements (page 289) .	
Explanation of Operands (see 4.16. Operands (page 104))	
<i>dest</i> : Destination register.	
<i>src0, src1</i> : Sources. Can be a register or immediate value.	
Exceptions (see Chapter 12. Exceptions (page 269))	
Signaling NaN floating-point numbers generate the invalid operation exception. The <code>s</code> comparison forms also generate the invalid operation exception for quiet NaN floating-point numbers.	

Table 5-25 Syntax for Packed Version of Compare (cmp) Instruction

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 (page 100))	
<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) (page 110) .	
<i>Type</i> : <code>s</code> , <code>u</code> , <code>f</code> .	

Explanation of Modifiers (see 4.13.2. Packed Data Types (page 100))

Length: 8x4, 8x8, 8x16, 16x2, 16x4, 16x8, 32x2, 32x4, 64x2. If the Base profile has been specified, 64x2 is not supported if *Type* is *f*. See 16.2.1. Base Profile Requirements (page 289).

Explanation of Operands (see 4.16. Operands (page 104))

dest: Destination register. This instruction performs an element-by-element compare and puts the result in the destination. *dest* must be a packed register of equal dimension as the sources. Each element in the packed destination is written to either all 1's (for true) or all 0's (for false) based on the result of each element-wise compare.

src0, *src1*: Sources. Must be a packed register or an immediate value.

Exceptions (see Chapter 12. Exceptions (page 269))

Signaling NaN floating-point numbers generate the invalid operation exception. The *s* comparison forms also generate the invalid operation exception for quiet NaN floating-point numbers.

For BRIG syntax, see 18.7.1.17. BRIG Syntax for Compare (cmp) Instruction (page 354).

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, sltu, sleu, sgtu, sgeu, snum, snan

Signaling comparisons. There are signaling forms of the ordered, unordered, `num` and `nan` comparisons. For example, `sle` is the signaling form of `le`. Only supported for floating point source operand types. Returns the same result as the corresponding non-signaling 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 5-26 \(below\)](#):

- 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 5-26 Floating-Point Comparisons

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

HSA IL	IEEE/ANSI Standard 754-2008	
Comparison Operation	True Relations	Operation
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;

```

5.19 Conversion (cvt) Instruction

5.19.1 Overview

The conversion (cvt) instruction converts a value with a particular type and length to another value with a different type and/or length.

Conversion instructions 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` instruction 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 \(page 107\)](#).

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 bits in 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 5-27 \(below\)](#) shows how the first step of the conversion instruction does the transformation. The table uses the notation defined in [Table 5-28 \(below\)](#).

Table 5-27 Conversion Methods

	Source b1	Source u8	Source s8	Source u16	Source s16	Source f16	Source u32	Source s32	Source f32	Source u64	Source s64	Source f64
Destination b1	-	ztest	ztest	ztest	ztest	ztest	ztest	ztest	ztest	ztest	ztest	ztest
Destination u8	zext	-	-	chop	chop	h2u	chop	chop	f2u	chop	chop	d2u
Destination s8	zext	-	-	chop	chop	h2s	chop	chop	f2s	chop	chop	d2s
Destination u16	zext	zext	sext	-	-	h2u	chop	chop	f2u	chop	chop	d2u
Destination s16	zext	zext	sext	-	-	h2s	chop	chop	f2s	chop	chop	d2s
Destination f16	u2h	u2h	s2h	u2h	s2h	-	u2h	s2h	f2h	u2h	s2h	d2h
Destination u32	zext	zext	sext	zext	sext	h2u	-	-	f2u	chop	chop	d2u
Destination s32	b2s	zext	sext	zext	sext	h2s	-	-	f2s	chop	chop	d2s
Destination f32	u2f	u2f	s2f	u2f	s2f	h2f	u2f	s2f	-	u2f	s2f	d2f
Destination u64	zext	zext	sext	zext	sext	h2u	zext	sext	f2u	-	-	d2u
Destination s64	b2s	zext	sext	zext	sext	h2s	zext	sext	f2s	-	-	d2s
Destination f64	u2d	u2d	s2d	u2d	s2d	h2d	u2d	s2d	f2d	u2d	s2d	-

Table 5-28 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.
sxt	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).
f2d	Convert 32-bit floating-point to 64-bit floating-point (double).
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.

5.19.2 Syntax

Table 5–29 Syntax for Conversion (cvt) Instruction

Opcode and Modifiers	Operands
cvt_ftz_round_destType destLength_srcType srcLength	dest, src

Explanation of Modifiers (see Table 4–2 (page 99))

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) (page 110).

round: Optional rounding mode. Only valid if *destType* and/or *srcType* is floating-point, unless both are floating-point types and *destType* size is 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. Floating-Point Rounding (page 109), 16.2.1. Base Profile Requirements (page 289), 5.19.3. Rules for Rounding for Conversions (next page), 5.19.4. Description of Integer Rounding Modes (next page), and 5.19.5. Description of Floating-Point Rounding Modes (page 164).

destType: b, u, s, f.

Explanation of Modifiers (see Table 4-2 (page 99))
<i>destLength</i> : 1, 8, 16, 32, 64. 1 is only allowed for <i>destType</i> of b. 1 and 8 are not allowed for <i>destType</i> of f. If the Base profile has been specified, 64 is not supported if <i>destType</i> is f. See 16.2.1. Base Profile Requirements (page 289) .
<i>srcType</i> : b, u, s, f.
<i>srcLength</i> : 1, 8, 16, 32, 64. 1 is only allowed for <i>srcType</i> of b. 1 and 8 are not allowed for <i>srcType</i> of f. If the Base profile has been specified, 64 is not supported if <i>srcType</i> is f. See 16.2.1. Base Profile Requirements (page 289) .
Explanation of Operands (see 4.16. Operands (page 104))
<i>dest</i> : Destination register.
<i>src</i> : Source. Can be a register or immediate value.
Exceptions (see Chapter 12. Exceptions (page 269))
Floating-point exceptions are allowed.

For BRIG syntax, see [18.7.1.18. BRIG Syntax for Conversion \(cvt\) Instruction \(page 354\)](#).

5.19.3 Rules for Rounding for Conversions

Rounding for conversions follows the rules shown in [Table 5-30 \(below\)](#).

If the type of rounding is none, then no rounding mode must be specified.

Table 5-30 Rules for Rounding for Conversions

From	To	Type of rounding	Default rounding
f	f (smaller size)	floating-point	default rounding mode (specified by the module header)
f	f (larger size)	none (must not specify rounding)	none (no rounding performed)
s or u	f	floating-point	default rounding mode (specified by the module header)
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)

5.19.4 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 5-31 \(page 164\)](#).

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 \(NaN\) \(page 111\)](#).

The `ftz` modifier is supported. See [4.19.3. Flush to Zero \(ftz\) \(page 110\)](#).

- 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 of infinite precision according to the rounding mode. This rounded value is considered out of 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.
- There are both non-signaling and signaling forms of the regular and saturating integer rounding modes. For example, `supi` is the signaling 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-signaling forms do not generate an inexact exception and correspond to the IEEE/ANSI Standard 754-2008 inexact conversions. The signaling 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.
- For the regular integer rounding modes:
 - If the rounded value is out of range:
 - The result is undefined. An invalid operation exception must be generated.
 - If the rounded value is not out of range:
 - The result is the rounded value. For the signaling rounding modes, if the source value, after any flush to zero, is not an integral value, then the inexact exception must be generated. Otherwise, no exceptions must be generated.
- For the saturating integer rounding modes:
 - If the rounded result is a NaN:
 - The result is 0. If the source is a signaling NaN then an invalid operation exception must be generated. Otherwise, no exceptions must be generated.
 - If the destination integer type is unsigned and the rounded result is -inf or less than 0.0:
 - The result is 0. It is implementation defined what, if any, exceptions are generated. A future version of HSAIL may define what exceptions must be generated.
 - If the destination integer type is unsigned and the rounded result is +inf or greater than $2^{destLength-1}$:
 - The result is $2^{destLength-1}$. It is implementation defined what, if any, exceptions are generated. A future version of HSAIL may define what exceptions must be generated.
 - If the destination integer type is signed and the rounded result is -inf or less than $-2^{destLength-1}$:
 - The result is $-2^{destLength-1}$. It is implementation defined what, if any, exceptions are generated. A future version of HSAIL may define what exceptions must be generated.

- If the destination integer type is signed and the rounded result is $+\text{inf}$ or greater than $2^{\text{destLength}-1}-1$:
 - The result is $2^{\text{destLength}-1}-1$. It is implementation defined what, if any, exceptions are generated. A future version of HSAIL may define what exceptions must be generated.
- Otherwise:
 - The result is the rounded value. For the signaling rounding modes, if the source value, after any flush to zero, is not an integral value, then the inexact exception must be generated. Otherwise, no exceptions must be generated.

The regular integer rounding modes might execute faster than the saturating integer rounding modes.

Table 5–31 Integer Rounding Modes

Regular Integer Rounding Modes		Saturating Integer Rounding Modes		Regular Integer Rounding Mode Description
Non-Signaling Form	Signaling Form	Non-Signaling Form	Signaling Form	
upi	supi	upi_sat	supi_sat	Rounds up to the nearest integer greater than or equal to the exact result.
downi	sdowni	downi_sat	sdowni_sat	Rounds down to the nearest integer less than or equal to the exact result.
zeroi	szeroi	zeroi_sat	szeroi_sat	Rounds to the nearest integer toward zero.
neari	sneari	neari_sat	sneari_sat	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
```

5.19.5 Description of Floating-Point Rounding Modes

The floating-point rounding modes are (see [4.19.2. Floating-Point Rounding \(page 109\)](#)):

- **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 the default floating-point rounding mode specified by the module header (see [Chapter 14. module Header \(page 284\)](#)). If the Base profile has been specified, then it must be omitted. See [4.19.2. Floating-Point Rounding \(page 109\)](#).

The `ftz` modifier is supported. See [4.19.3. Flush to Zero \(ftz\) \(page 110\)](#).

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 \(NaN\) \(page 111\)](#).

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. Floating-Point Rounding \(page 109\)](#).

- 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 the default floating-point rounding mode specified by the module header (see [Chapter 14. module Header \(page 284\)](#)) If the Base profile has been specified, then it must be omitted.. See [4.19.2. Floating-Point Rounding \(page 109\)](#).

The `ftz` modifier is not supported. See [4.19.3. Flush to Zero \(ftz\) \(page 110\)](#).

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. Floating-Point Rounding \(page 109\)](#).

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 Instructions

This chapter describes the HSAIL memory instructions.

6.1 Memory and Addressing

Memory instructions transfer data between registers and memory and can define memory synchronization between work-items and other agents:

- The ordinary load and atomic load instructions move contents from memory to a register.
- The ordinary store and atomic store instructions move contents of a register into memory.
- The atomic read-modify-write memory instructions 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 instructions 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 instruction 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 \(page 39\)](#)). Each instruction indicates the type of address.

Memory instructions 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 \(page 35\)](#).

6.1.1 How Addresses Are Formed

The format of an address expression is described in [4.18. Address Expressions \(page 106\)](#).

Every address expression has one or both of the following:

- Name in square brackets.

If the instruction 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 instruction.

- 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 instruction. For example, an `s` register must be used for a group segment address, a `d` register must be used for a global segment address in the large machine model, and an `s` register must be used for a global address in the small machine model. See [Table 2–3 \(page 40\)](#).

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 instruction 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 \(page 82\)](#).

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 \(page 35\)](#).

If the resulting effective address value is outside the memory segment specified by the instruction, or is a flat address that is outside any segment, the result of the memory segment instruction is undefined.

For more information, see [4.18. Address Expressions \(page 106\)](#).

6.1.2 Memory Hierarchy

[Figure 6–1 \(next page\)](#) shows an example of the memory used by an agent executing a kernel dispatch grid.

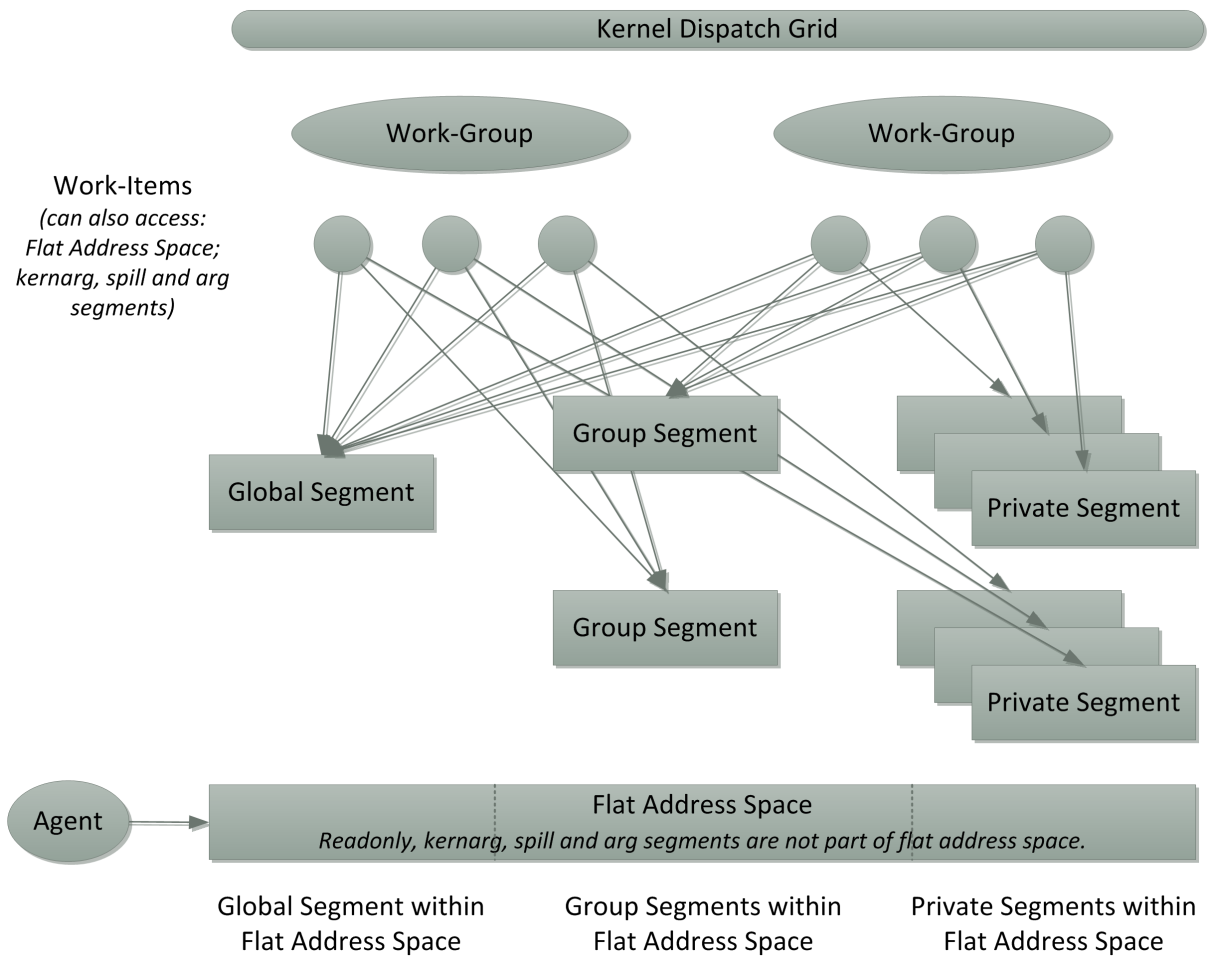
The addresses used to access memory do not need to be naturally aligned to a multiple of the access size.

The segment converting instructions (`ftos` and `stof`) convert addresses between flat address and segment address.

The segment checking instruction (`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.

Figure 6–1 Memory Hierarchy



6.1.3 Alignment

A memory instruction 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 instructions 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 aliasing information to the finalizer.

Equivalence classes are specified with the memory and image instructions.

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 instructions 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` instructions 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 instructions and modifiers to the HSA Memory Model defined in the *HSA Platform System Architecture Specification Version 1.0 Chapter 3 HSA Memory Consistency Model*.

Memory instructions are the load, store, atomic, signal, and memory fence instructions defined in this chapter. Read, write, and fence image instructions are the `rdimage`, `ldimage`, `stimage`, and `imagefence` instructions defined in [Chapter 7. Image Instructions \(page 194\)](#) which use a separate image memory model defined in [7.1.10. Image Memory Model \(page 218\)](#).

6.2.1 Memory Order

The memory synchronization of an instruction is specified by the memory order modifier which can have the following values which correspond to the memory orders with the same names defined in the *HSA Platform System Architecture Specification Chapter 3 HSA Memory Consistency Model*:

- `scacq` specifies the instruction is a sequentially consistent acquire memory instruction.
- `screl` specifies the instruction is a sequentially consistent release memory instruction.
- `scar` specifies the instruction is both a sequentially consistent acquire and sequentially consistent release memory instruction.
- `rlx` specifies the instruction is a relaxed memory instruction.

The memory model requires that every work-item and agent thread observes the same total ordering of synchronizing memory instructions for a data race free program. Therefore, if sequential consistency is required on synchronizing memory instructions, it is only necessary to ensure that the relaxed atomic memory instructions executed by a work-item are ordered with respect to the acquire and release atomic memory instructions executed by the same work-item. This can be achieved by:

- using `scar` on read-modify-write atomic memory instructions,
- preceding a load acquire atomic memory instruction with a release memory fence,
- and following a store release atomic memory instruction 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 instructions in a critical section bracketed by the acquire and release memory instructions can be moved out of the section. An acquire access of a global variable ensures that the subsequent memory instructions 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 memory instruction or memory fence is specified by the memory scope modifier which can have the following values which correspond to the memory scopes with the same names defined in the *HSA Platform System Architecture Specification Version 1.0* Chapter 3 *HSA Memory Consistency Model*:

- `wi` specifies work-item scope which includes only the executing work-item. Only supported by the image fence instruction on the image segment. See [7.6. Image Fence \(imagefence\) Instruction \(page 225\)](#).
- `wave` 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.
- `agent` specifies kernel agent scope which includes all work-items on the same kernel agent executing kernel dispatches for the same application process as the executing work-item. Only supported for the global segment.
- `system` specifies the entire HSA system scope which includes all work-items on all kernel agents 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 `system` scope on certain ranges of virtual addresses. If a memory instruction with `system` scope is performed on a location with a virtual address in a range that does not support `system` scope, then the memory instruction behaves as if `agent` scope was specified.

The Base profile requires that the HSA runtime is used to allocate all memory that is required to support `system` scope (see [16.2.1. Base Profile Requirements \(page 289\)](#)).

See [6.2.6. Course Grain Allocation \(page 172\)](#) for additional restrictions on the global segment

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.

6.2.3 Memory Synchronization Segments

The segment of an atomic memory instruction is specified by the segment modifier of the instruction and can have the following values:

- `group` specifies the group segment.
- `global` specifies the global segment.

- *image* specifies the image segment. It includes all the memory that holds image data, and is implicitly used by the image instructions. It is only supported if the "IMAGE" extension directive has been specified (see [13.1.2. extension IMAGE \(page 274\)](#)).

If the memory segment is omitted for an atomic or ordinary memory instruction, it specifies that a flat memory address is being used. See [6.2.9. Flat Addresses \(page 173\)](#).

A synchronizing memory instruction and memory fence affects memory operations to both the `group` and `global` segments regardless of the segment specified by the instruction.

See [2.8. Segments \(page 31\)](#).

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 \(previous page\)](#)). Only ordinary memory instructions are supported for these segments.

The `private`, `spill`, and `arg` segments can only be accessed by a single work-item.

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 instructions are allowed. The behavior is undefined if the locations are accessed other than by work-items that belong to the kernel dispatch.

`readonly` segment locations have agent allocation (see [6.2.5. Agent Allocation \(below\)](#)) and the behavior is undefined if the locations accessed by a kernel dispatch change value during its execution. Only load instructions 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 kernel agent. See [4.10. Variable Initializers \(page 94\)](#).

See [2.8. Segments \(page 31\)](#).

6.2.5 Agent Allocation

A segment variable with agent allocation results in distinct allocations of the variable for each kernel agent, each with a distinct segment address. If a location is accessed that is part of an agent allocation except from the kernel agent that the allocation is associated, or by using the HSA runtime copy operation, the results are undefined.

The `global` segment allows variables to have agent allocation. See [4.3.10. Declaration and Definition Qualifiers \(page 69\)](#). The HSA runtime memory allocator can be used to allocate global segment agent allocation memory by specifying a memory topology region that supports agent allocation for the required agent. The results are undefined if the returned address range is accessed except as a global segment address on the specified agent, or as the argument to the HSA runtime copy operation.

All `readonly` segment variables have agent allocation. The HSA runtime memory allocator can be used to allocate `readonly` segment memory by specifying a memory topology region that supports the `readonly` segment for the required agent. The results are undefined if the returned address range is accessed except as a `readonly` segment address on the specified agent, or as the argument to the HSA runtime copy operation.

An implementation may use memory that does not support `system` scope to allocate variables with agent allocation (see [6.2.2. Memory Scope \(previous page\)](#)).

6.2.6 Course Grain Allocation

The HSA runtime can be used to allocate a range of virtual addresses that have coarse grain synchronization. Such virtual address ranges are termed coarse grain allocations.

A course grain allocation does not support `system scope` (see [6.2.2. Memory Scope \(page 170\)](#)).

The HSA runtime can be used to specify ownership of coarse grain allocations. Only one agent can have ownership of a coarse grain allocation at any one time. The ownership can either be read-only or read-write.

A program is undefined if an agent:

- reads from a coarse grain allocation when it does not have read-only or read-write ownership
- writes to a coarse grain allocation when it does not have read-write ownership

6.2.7 Kernel Dispatch Memory Synchronization

Before a work-item starts executing, no implicit acquire memory fence is performed.

When a work-item completes execution, no implicit release memory fence is performed.

However, packet processor fences can be used to affect the work-items of kernel dispatch packets:

- An acquire packet fence can be used to perform an acquire that affects all work-items of kernel dispatch packets on any User Mode Queue of the same agent that have not yet entered the active phase.
- A release packet fence can be used to perform a release that affects all work-items of kernel dispatch packets on any User Mode Queue of the same agent that have completed the active phase.

The packet processor fences apply to the global and image segments, and can specify either `agent` or `system scope`.

Packet processor fences can be used with any packet. Note that packet processor fences do not just apply to the packet to which they belong.

For more information on the packet processor fence memory model, see the *HSA Platform System Architecture Specification Version 1.0*: section 2.9.1 *Packet Header* for the definition of when the packet fences are performed for each packet kind; section 2.9.2 *Packet Process Flow* for more details on the processing of the different packets; and section 3.4 *Packet Processor Fences*.

Because global memory update instructions 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 kernel agent, an implementation (both hardware and finalizer) cannot delete the update of the final value of global memory locations by the ordinary memory instructions 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 instructions, or atomic memory instructions with a memory scope of work-group or wavefront, does not give an implementation permission to delete a global memory update instruction 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 instruction, or an atomic memory instruction at a scope less than `system`, must be made visible by a release to `system` 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 instruction and a release at `system` scope. Similarly for locations used for kernel agent only coherent variables being released to `agent` and `system` scopes.

6.2.8 Execution Barrier

A barrier instruction is used to synchronize the execution of the work-items that participate in an associated execution barrier instance. In addition, an execution barrier instruction defines a memory ordering of synchronizing memory instructions executed by work-items participating in the execution barrier instance with respect to the synchronizing memory instructions executed by the other work-items participating in the same execution barrier instance. See [9.3. Execution Barrier \(page 238\)](#).

6.2.9 Flat Addresses

Synchronizing memory Instructions that use a flat address are defined as the equivalent segment address synchronizing memory Instruction using:

- A segment and segment address corresponding to the actual flat address when the flat synchronizing memory Instruction is executed at runtime.
- A memory scope that is the minimum of the memory scope specified by the flat synchronizing memory Instruction and the widest scope supported by the segment of the actual flat address when the flat synchronizing memory Instruction is executed at runtime.
- A memory order corresponding to that specified by the flat synchronizing memory Instruction.

6.3 Load (ld) Instruction

The load (`ld`) instruction loads from memory using a segment or flat address expression (see [4.18. Address Expressions \(page 106\)](#)) and places the result into one or more registers. It is an ordinary non-synchronizing memory instruction (see [6.2. Memory Model \(page 169\)](#)).

There are four variants of the `ld` instruction, depending on the number of destinations: one, two, three, or four.

The size of the value loaded is specified by the instruction's compound type. The value is stored into the destination register following the rules in [4.16. Operands \(page 104\)](#). Integer values are sign-extended or zero-extended to fit the destination register size. `£16` values are stored in the least significant 16 bits of the `s` register, and the most significant 16 bits are undefined (see [4.19.1. Floating-Point Numbers \(page 109\)](#)). No conversions are performed on other types. Use an explicit `cvt` instruction if floating-point conversion is required.

If the Base profile has been specified then the 64-bit floating-point type (`£64`) is not supported (see [16.2.1. Base Profile Requirements \(page 289\)](#)).

6.3.1 Syntax

Table 6–1 Syntax for Load (ld) Instruction

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
<code>v2</code> , <code>v3</code> , and <code>v4</code> : Optional vector element count. Used to specify that multiple contiguous memory locations, each of type <i>TypeLength</i> , are being loaded. See the Description below.
<i>segment</i> : Optional segment: <code>global</code> , <code>group</code> , <code>private</code> , <code>kernarg</code> , <code>readonly</code> , <code>spill</code> , or <code>arg</code> . If omitted, a flat address is used. See 2.8. Segments (page 31) .
<i>align(n)</i> : 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.
<i>const</i> : Optional. Used to indicate if the memory loaded is constant. Only allowed if <i>segment</i> specifies the global segment or flat address. See the Description below.
<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, which indicates that any memory location may be aliased. See 6.1.4. Equivalence Classes (page 168) .
<i>width</i> : Optional: <code>width(n)</code> , <code>width(WAVESIZE)</code> , or <code>width(all)</code> . 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 <code>width(1)</code> , indicating each active work-item can load different value(s). See the Description below.
<i>Type</i> : <code>u</code> , <code>s</code> , <code>f</code> . The <i>Type</i> specifies how the value is expanded to the size of the destination. See Table 4–2 (page 99) .
<i>Length</i> : 8, 16, 32, 64. If the Base profile has been specified, 64 is not supported if <i>Type</i> is <code>f</code> . The <i>Length</i> specifies the amount of data fetched from memory, and the amount to increment the address when the destination is a vector operand. See Table 4–2 (page 99) and 16.2.1. Base Profile Requirements (page 289) .
<i>TypeLength</i> can also be <code>b128</code> , in which case <i>destn</i> 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 <i>destn</i> must be a <code>d</code> register.

Explanation of Operands (see 4.16. Operands (page 104))
<code>dest0</code> , <code>dest1</code> , <code>dest2</code> , <code>dest3</code> : Destination registers.
<i>address</i> : Address to be loaded from. Must be an address expression for an address in <i>segment</i> (see 4.18. Address Expressions (page 106)).

Exceptions (see Chapter 12. Exceptions (page 269))
Invalid address exceptions are allowed. May generate a memory exception if address is unaligned and the <i>aligned</i> modifier has been specified.

For BRIG syntax, see [18.7.2. BRIG Syntax for Memory Instructions \(page 354\)](#).

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 by the instruction.

Front ends should generate vector forms whenever possible. The following forms are equivalent but the vector form is often faster.

Slow form:

```
ld_s32 $s0, [$s1];
ld_s32 $s1, [$s1+4];
```

Fast form using the vector:

```
ld_v2_s32 ($s0,$s1), [$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. The behavior is undefined if a memory load marked as aligned is in fact not aligned 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 \(page 295\)](#).

`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: Although the values loaded by `kernarg` and `non-const` readonly segment do not change during the execution of a single kernel dispatch, the values can be different for each kernel dispatch. Therefore, they are not considered constant memory accesses.

On some implementations, knowing a load is accessing constant memory might be more efficient. The behavior 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 \(page 295\)](#).

`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 \(page 41\)](#)). 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 instruction 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 \(page 166\)](#).

It is not valid to use a flat load instruction with an identifier. The following code is not valid:

```
ld_s64 $d1, [&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` instructions 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` instruction.

For `f16`, the destination must be an *s* register. The value is loaded into the least significant 16 bits of the *s* register, and the most significant 16 bits are undefined. If a conversion is required, then it should be done explicitly using a `cvt` instruction. See [4.19.1. Floating-Point Numbers \(page 109\)](#).

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 Instructions \(page 216\)](#) and [6.8. Notification \(signal\) Instructions \(page 187\)](#)).

The `ld` instruction is an ordinary non-synchronizing memory instruction. 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 synchronizing memory instructions or memory fences in conjunction with relaxed atomic memory instructions. For example, a `atomic_ld_acq` acts like a partial fence; no memory instruction after the `atomic_ld_acq` can be moved before the `atomic_ld_acq`. See [6.2. Memory Model \(page 169\)](#).

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) Instruction

The store (st) instruction stores a value from one or more registers, or an immediate value, (see [4.16. Operands \(page 104\)](#)) into memory using a segment or flat address expression (see [4.18. Address Expressions \(page 106\)](#)). It is an ordinary non-synchronizing memory instruction (see [6.2. Memory Model \(page 169\)](#)).

There are four variants of the store instruction, 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 \(page 289\)](#)).

6.4.1 Syntax

Table 6–2 Syntax for Store (st) Instruction

Opcode and Modifiers	Operands
st_segment_align(n)_equiv(n)_TypeLength	<i>src0, address</i>
st_v2_segment_align(n)_equiv(n)_TypeLength	<i>(src0,src1), address</i>
st_v3_segment_align(n)_equiv(n)_TypeLength	<i>(src0,src1,src2), address</i>
st_v4_segment_align(n)_equiv(n)_TypeLength	<i>(src0,src1,src2,src3), address</i>

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 stored. See the Description below.

segment: Optional segment: global, group, private, spill, or arg. If omitted, flat is used. See [2.8. Segments \(page 31\)](#).

Explanation of Modifiers
<i>align</i> (<i>n</i>): 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.
<i>equiv</i> (<i>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, which indicates that any memory location may be aliased. See 6.1.4. Equivalence Classes (page 168) .
<i>Type</i> : <i>u</i> , <i>s</i> , <i>f</i> . The <i>Type</i> specifies how the value is extracted from the source to match the size of the destination. See Table 4-2 (page 99) .
<i>Length</i> : 8, 16, 32, 64. If the Base profile has been specified, 64 is not supported if <i>Type</i> is <i>f</i> . The <i>Length</i> specifies the amount of data stored, and the amount to increment the address when the destination is a vector operand. See Table 4-2 (page 99) and 16.2.1. Base Profile Requirements (page 289) .
<i>TypeLength</i> can also be <i>b128</i> , in which case <i>srcn</i> must be a <i>q</i> register; or <i>roimg</i> , <i>woimg</i> , <i>rwimg</i> , <i>samp</i> , <i>sig32</i> , or <i>sig64</i> , in which case <i>srcn</i> must be a <i>d</i> register. If <i>roimg</i> , <i>woimg</i> , <i>rwimg</i> or <i>samp</i> then <i>segment</i> must be <i>arg</i> .
Explanation of Operands (see 4.16. Operands (page 104))
<i>src0</i> , <i>src1</i> , <i>src2</i> , <i>src3</i> : Sources. Can be a register or immediate value.
<i>address</i> : Address to be stored into. Must be an address expression for an address in <i>segment</i> (see 4.18. Address Expressions (page 106)).
Exceptions (see Chapter 12. Exceptions (page 269))
Invalid address exceptions are allowed. May generate a memory exception if address is unaligned and the <i>aligned</i> modifier has been specified.

For BRIG syntax, see [18.7.2. BRIG Syntax for Memory Instructions \(page 354\)](#).

6.4.2 Description

v2, *v3*, and *v4*

When *v2*, *v3*, or *v4* is used, HSAIL will store consecutive values from multiple registers or immediate values. The address is incremented by the size of the *TypeLength* specified the instruction.

Front ends should generate vector forms whenever possible. The following forms are equivalent but the vector form is often faster.

Slow form:

```
st_s32 $s0, [$s2];
st_s32 $s1, [$s2+4];
```

Fast form using the vector:

```
st_v2_s32 ($s0, $s1), [$s2];
```

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. The results are undefined if a memory store marked as aligned is in fact not aligned 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 \(page 295\)](#).

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 \(page 166\)](#).

It is not valid to use a flat store instruction 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. For example, 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` instruction.

For `f16`, if the source is a register, it must be an *s* register and the least significant 16 bits are stored. See [4.19.1. Floating-Point Numbers \(page 109\)](#).

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 Instructions \(page 216\)](#) and [6.8. Notification \(signal\) Instructions \(page 187\)](#)).

The `roimg`, `woimg`, `rwimg` and `samp` value types are only allowed if *segment* is `arg` (see [7.1.7. Image Creation and Image Handles \(page 211\)](#) and [7.1.8. Sampler Creation and Sampler Handles \(page 214\)](#)).

The `st` instruction is an ordinary non-synchronizing memory instruction. 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 synchronizing memory instructions or memory fences in conjunction with synchronizing memory instructions. For example, a `atomic_st_rel` acts like a partial fence; no memory instruction before the `atomic_st_rel` can be moved after the `atomic_st_rel`. See [6.2. Memory Model \(page 169\)](#).

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 Instructions

Atomic memory instructions 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 instruction on a memory location, no write to the same memory location using the same memory scope from outside the current atomic instruction by any work-item or agent can occur between the read and write performed by the instruction.

If multiple atomic memory instructions from different work-items or agents target the same memory location, the instructions 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 instructions 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 instructions 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 instructions have two forms:

- `atomic` instructions which return the value that was read before the modification. These instructions require the *dest* (destination) operand.
- `atomicnoret` instructions which do not return a value. These instructions do not have a destination operand.

An implementation may execute `atomicnoret` read-modify-write memory instructions faster than the corresponding `atomic` read-modify-write memory instructions. Therefore, compilers should identify cases where the result of read-modify-write memory instructions is not needed and whenever possible, should generate `atomicnoret` instructions.

Both `atomic` and `atomicnoreset` instructions can specify a memory order and memory scope.

For more information, see:

- [6.2. Memory Model \(page 169\)](#)
- [6.6. Atomic \(atomic\) Instructions \(below\)](#)
- [6.7. Atomic No Return \(atomicnoreset\) Instructions \(page 185\)](#)

6.6 Atomic (atomic) Instructions

The atomic memory (`atomic`) instructions 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` instructions are atomic memory instructions that can either be synchronizing or non-synchronizing, all except `atomic_ld` are read-modify-write instructions (see [6.2. Memory Model \(page 169\)](#)).

6.6.1 Syntax

Table 6–3 Syntax for Atomic Instructions

Opcode and Modifiers	Operands
<code>atomic_ld_segment_order_scope_equiv(n)_TypeLength</code>	<i>dest</i> , <i>address</i>
<code>atomic_and_segment_order_scope_equiv(n)_TypeLength</code>	<i>dest</i> , <i>address</i> , <i>src0</i>
<code>atomic_or_segment_order_scope_equiv(n)_TypeLength</code>	<i>dest</i> , <i>address</i> , <i>src0</i>
<code>atomic_xor_segment_order_scope_equiv(n)_TypeLength</code>	<i>dest</i> , <i>address</i> , <i>src0</i>
<code>atomic_exch_segment_order_scope_equiv(n)_TypeLength</code>	<i>dest</i> , <i>address</i> , <i>src0</i>
<code>atomic_add_segment_order_scope_equiv(n)_TypeLength</code>	<i>dest</i> , <i>address</i> , <i>src0</i>
<code>atomic_sub_segment_order_scope_equiv(n)_TypeLength</code>	<i>dest</i> , <i>address</i> , <i>src0</i>
<code>atomic_wrapinc_segment_order_scope_equiv(n)_TypeLength</code>	<i>dest</i> , <i>address</i> , <i>src0</i>
<code>atomic_wrapdec_segment_order_scope_equiv(n)_TypeLength</code>	<i>dest</i> , <i>address</i> , <i>src0</i>
<code>atomic_max_segment_order_scope_equiv(n)_TypeLength</code>	<i>dest</i> , <i>address</i> , <i>src0</i>
<code>atomic_min_segment_order_scope_equiv(n)_TypeLength</code>	<i>dest</i> , <i>address</i> , <i>src0</i>
<code>atomic_cas_segment_order_scope_equiv(n)_TypeLength</code>	<i>dest</i> , <i>address</i> , <i>src0</i> , <i>src1</i>

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 (page 31) .
<i>order</i> : Memory order used to specify synchronization. Can be <code>rlx</code> (relaxed) and <code>scacq</code> (sequentially consistent acquire) for all instructions, and for all instructions except <code>ld</code> can also be <code>screl</code> (sequentially consistent release) or <code>scar</code> (sequentially consistent acquire and release). See 6.2.1. Memory Order (page 169) .
<i>scope</i> : Memory scope used to specify synchronization. Can be <code>wave</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>agent</code> (kernel agent) or <code>system</code> (system). For a <code>flat</code> address, can be <code>wave</code> , <code>wg</code> , <code>agent</code> , and <code>system</code> , but if the address references the <code>group</code> segment, <code>agent</code> and <code>system</code> behave as if <code>wg</code> was specified. See 6.2.2. Memory Scope (page 170) .
<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, which indicates that any memory location may be aliased. See 6.1.4. Equivalence Classes (page 168) .
<i>Type</i> : <code>b</code> for <code>ld</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>max</code> , <code>min</code> ; <code>u</code> for <code>wrapinc</code> , <code>wrapdec</code> . See Table 4–2 (page 99) .

Explanation of Modifiers
<i>Length</i> : 32, 64. See Table 4–2 (page 99) . 64 is not allowed for small machine model. See 2.9. Small and Large Machine Models (page 39) .
Explanation of Operands (see 4.16. Operands (page 104))
<i>dest</i> : Destination register.
<i>address</i> : Source location in the specified segment. Must be an address expression for an address in <i>segment</i> (see 4.18. Address Expressions (page 106)).
<i>src0</i> , <i>src1</i> : Sources. Can be a register or immediate value.
Exceptions (see Chapter 12. Exceptions (page 269))
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 Instructions \(page 354\)](#).

6.6.2 Description of Atomic and Atomic No Return Instructions

ld

Loads the contents of the *address* into *dest*.

```
dest = [address];
```

NOTE: There is no `atomicnoret` version of this instruction.

st

Stores the value in *src0* to *address*.

```
[address] = src0;
```

NOTE: There is only an `atomicnoret` version of this instruction.

and

ANDs the contents of the *address* with the value in *src0*.

For the `atomic` instruction, sets *dest* to the original contents of the *address*.

```
original = [address];
[address] = original & src0;
dest = original; // Only if atomic instruction
```

or

ORs the contents of the *address* with the value in *src0*.

For the `atomic` instruction, sets *dest* to the original contents of the *address*.

```
original = [address];
[address] = original | src0;
dest = original; // Only if atomic instruction
```

xor

XORs the contents of the *address* with the value in *src0*.

For the `atomic` instruction, sets *dest* to the original contents of the *address*.

```
original = [address];
[address] = original ^ src0;
dest = original; // Only if atOMIC instruction
```

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 instruction.

add

Adds (using integer arithmetic) the value in *src0* to the contents of the memory location with address *address*. For the `atomic` instruction, sets *dest* to the original contents of the *address*.

```
original = [address];
[address] = original + src0;
dest = original; // Only if atOMIC instruction
```

sub

Subtracts (using integer arithmetic) the value in *src0* from the contents of the memory location with address *address*. For the `atomic` instruction, sets *dest* to the original contents of the *address*.

```
original = [address];
[address] = original - src0;
dest = original; // Only for atOMIC instruction
```

min,max

Sets the memory location with *address* to the minimum/maximum of the original value and *src0*. For the `atomic` instructions, sets *dest* to the original contents of the *address*.

```
original = [address];
[address] = min/max(original, src0);
dest = original; // Only if atOMIC instruction
```

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 instruction
```

After the instruction, the contents of the *address* will be in the range $[0, src0]$ inclusive. For the `atomic` instruction, 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 value of 1. On some implementations this may perform significantly better than a `wrapinc`.

wrapdec

Decrements, 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 instruction
```

After the instruction, the contents of the *address* will be in the range $[0, src0]$ inclusive. For the atomic instruction, 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 value 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 instruction, 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 instruction
```

NOTE: There is no `atomicnoret` version of this instruction.

Examples

```
atomic_ld_global_rlx_system_equiv(49)_b32 $s1, [&x];
atomic_ld_global_scacq_agent_b32 $s1, [&x];
atomic_ld_group_scacq_wg_b32 $s1, [&x];
atomic_ld_scacq_system_b64 $d1, [$d0];

atomic_and_global_scar_wg_b32 $s1, [&x], 23;
atomic_and_global_rlx_wave_b32 $s1, [&x], 23;
atomic_and_group_rlx_wg_b32 $s1, [&x], 23;
atomic_and_rlx_system_b32 $s1, [$d0], 23;

atomic_or_global_scar_system_b64 $d1, [&x], 23;
atomic_or_global_screl_system_b64 $d1, [&x], 23;
atomic_or_group_scacq_wave_b64 $d1, [&x], 23;
atomic_or_rlx_system_b64 $d1, [$d0], 23;

atomic_xor_global_scar_system_b64 $d1, [&x], 23;
atomic_xor_global_rlx_system_b64 $d1, [&x], 23;
atomic_xor_group_rlx_wg_b64 $d1, [&x], 23;
atomic_xor_screl_agent_b64 $d1, [$d0], 23;

atomic_cas_global_scar_system_b64 $d1, [&x], 23, 12;
atomic_cas_global_rlx_system_b64 $d1, [&x], 23, 1;
atomic_cas_group_rlx_wg_b64 $d1, [&x], 23, 9;
atomic_cas_rlx_system_b64 $d1, [$d0], 23, 12;

atomic_exch_global_scar_system_b64 $d1, [&x], 23;
atomic_exch_global_rlx_system_b64 $d1, [&x], 23;
atomic_exch_group_rlx_wg_b64 $d1, [&x], 23;
atomic_exch_rlx_system_b64 $d1, [$d0], 23;

atomic_add_global_scar_system_u64 $d1, [&x], 23;
atomic_add_global_rlx_system_s64 $d1, [&x], 23;
atomic_add_group_rlx_wg_u64 $d1, [&x], 23;
atomic_add_screl_system_s64 $d1, [$d0], 23;

atomic_sub_global_scar_system_u64 $d1, [&x], 23;
atomic_sub_global_rlx_system_s64 $d1, [&x], 23;
atomic_sub_group_rlx_wg_u64 $d1, [&x], 23;
atomic_sub_rlx_agent_s64 $d1, [$d0], 23;
```

```

atomic_wrapinc_global_scar_system_u64 $d1, [&x], 23;
atomic_wrapinc_global_rlx_system_u64 $d1, [&x], 23;
atomic_wrapinc_group_rlx_wg_u64 $d1, [&x], 23;
atomic_wrapinc_rlx_system_u64 $d1, [$d0], 23;

atomic_wrapdec_global_scar_system_u64 $d1, [&x], 23;
atomic_wrapdec_global_rlx_system_u64 $d1, [&x], 23;
atomic_wrapdec_group_rlx_wg_u64 $d1, [&x], 23;
atomic_wrapdec_rlx_system_u64 $d1, [$d0], 23;

atomic_max_global_scar_system_s64 $d1, [&x], 23;
atomic_max_global_rlx_system_s64 $d1, [&x], 23;
atomic_max_group_rlx_wg_u64 $d1, [&x], 23;
atomic_max_rlx_system_u64 $d1, [$d0], 23;

atomic_min_global_scar_system_s64 $d1, [&x], 23;
atomic_min_global_rlx_system_s64 $d1, [&x], 23;
atomic_min_group_rlx_wg_u64 $d1, [&x], 23;
atomic_min_rlx_system_u64 $d1, [$d0], 23;

```

6.7 Atomic No Return (atomicnoret) Instructions

The atomic no return memory (atomicnoret) instructions, 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*. The `atomicnoret_st` instruction atomically stores the value in *src0* at *address*. The `atomicnoret` instructions do not have a destination, are atomic memory instructions that can either be synchronizing or non-synchronizing, and all except `atomicnoret_st` are read-modify-write instructions (see 6.2. Memory Model (page 169)).

6.7.1 Syntax

Table 6–4 Syntax for Atomic No Return Instructions

Opcodes and Modifiers	Operands
<code>atomicnoret_st_segment_order_scope_equiv(n)_TypeLength</code>	<i>address</i> , <i>src0</i>
<code>atomicnoret_and_segment_order_scope_equiv(n)_TypeLength</code>	<i>address</i> , <i>src0</i>
<code>atomicnoret_or_segment_order_scope_equiv(n)_TypeLength</code>	<i>address</i> , <i>src0</i>
<code>atomicnoret_xor_segment_order_scope_equiv(n)_TypeLength</code>	<i>address</i> , <i>src0</i>
<code>atomicnoret_add_segment_order_scope_equiv(n)_TypeLength</code>	<i>address</i> , <i>src0</i>
<code>atomicnoret_sub_segment_order_scope_equiv(n)_TypeLength</code>	<i>address</i> , <i>src0</i>
<code>atomicnoret_wrapinc_segment_order_scope_equiv(n)_TypeLength</code>	<i>address</i> , <i>src0</i>
<code>atomicnoret_wrapdec_segment_order_scope_equiv(n)_TypeLength</code>	<i>address</i> , <i>src0</i>
<code>atomicnoret_max_segment_order_scope_equiv(n)_TypeLength</code>	<i>address</i> , <i>src0</i>
<code>atomicnoret_min_segment_order_scope_equiv(n)_TypeLength</code>	<i>address</i> , <i>src0</i>

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 (page 31).

order: Memory order used to specify synchronization. Can be `rlx` (relaxed) and `screl` (sequentially consistent release) for all instructions, and for all instructions except `st` can also be `scacq` (sequentially consistent acquire) or `scar` (sequentially consistent acquire and release). See 6.2.1. Memory Order (page 169).

Explanation of Modifiers
<i>scope</i> : Memory scope used to specify synchronization. Can be <i>wave</i> (wavefront) and <i>wg</i> (work-group) for <i>global</i> or <i>group</i> segments, and for <i>global</i> segment can also be <i>agent</i> (kernel agent) or <i>system</i> (system). For a flat address, any value can be used, but if the address references the <i>group</i> segment, <i>agent</i> and <i>system</i> behave as if <i>wg</i> was specified. See 6.2.2. Memory Scope (page 170) .
<i>equiv</i> (<i>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, which indicates that any memory location may be aliased. See 6.1.4. Equivalence Classes (page 168) .
<i>Type</i> : <i>b</i> for <i>st</i> , <i>and</i> , <i>or</i> , <i>xor</i> ; <i>u</i> and <i>s</i> for <i>add</i> , <i>sub</i> , <i>max</i> , <i>min</i> ; <i>u</i> for <i>wrapinc</i> , <i>wrapdec</i> . See Table 4–2 (page 99) .
<i>Length</i> : 32, 64. See Table 4–2 (page 99) . 64 is not allowed for small machine model. See 2.9. Small and Large Machine Models (page 39) .
Explanation of Operands (see 4.16. Operands (page 104))
<i>address</i> : Source location in the specified segment. Must be an address expression for an address in <i>segment</i> (see 4.18. Address Expressions (page 106)).
<i>src0</i> : Source. Can be a register or immediate value.
Exceptions (see Chapter 12. Exceptions (page 269))
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 Instructions \(page 354\)](#).

6.7.2 Description

See [6.6.2. Description of Atomic and Atomic No Return Instructions \(page 182\)](#).

The *atomicnoret* instructions change memory in the same way as the *atomic* instructions but do not have a destination.

Examples

```
atomicnoret_st_global_rlx_system_equiv(49)_b32 [&x], $s1;
atomicnoret_st_global_screl_agent_b32 [&x], $s1;
atomicnoret_st_group_screl_wg_b32 [&x], $s1;
atomicnoret_st_screl_system_b64 [$d0], $d1;

atomicnoret_and_global_scar_wg_b32 [&x], 23;
atomicnoret_and_global_rlx_wave_b32 [&x], 23;
atomicnoret_and_group_rlx_wg_b32 [&x], 23;
atomicnoret_and_rlx_system_b32 [$d0], 23;

atomicnoret_or_global_scar_system_b64 [&x], 23;
atomicnoret_or_global_screl_system_b64 [&x], 23;
atomicnoret_or_group_scarq_wave_b64 [&x], 23;
atomicnoret_or_rlx_system_b64 [$d0], 23;

atomicnoret_xor_global_scar_system_b64 [&x], 23;
atomicnoret_xor_global_rlx_system_b64 [&x], 23;
atomicnoret_xor_group_rlx_wg_b64 [&x], 23;
atomicnoret_xor_screl_agent_b64 [$d0], 23;

atomicnoret_add_global_scar_system_u64 [&x], 23;
atomicnoret_add_global_rlx_system_s64 [&x], 23;
atomicnoret_add_group_rlx_wg_u64 [&x], 23;
atomicnoret_add_screl_system_s64 [$d0], 23;

atomicnoret_sub_global_scar_system_u64 [&x], 23;
```

```

atomicnoret_sub_global_rlx_system_s64 [&x], 23;
atomicnoret_sub_group_rlx_wg_u64 [&x], 23;
atomicnoret_sub_rlx_agent_s64 [$d0], 23;

atomicnoret_wrapinc_global_scar_system_u64 [&x], 23;
atomicnoret_wrapinc_global_rlx_system_u64 [&x], 23;
atomicnoret_wrapinc_group_rlx_wg_u64 [&x], 23;
atomicnoret_wrapinc_rlx_system_u64 [$d0], 23;

atomicnoret_wrapdec_global_scar_system_u64 [&x], 23;
atomicnoret_wrapdec_global_rlx_system_u64 [&x], 23;
atomicnoret_wrapdec_group_rlx_wg_u64 [&x], 23;
atomicnoret_wrapdec_rlx_system_u64 [$d0], 23;

atomicnoret_max_global_scar_system_u64 [&x], 23;
atomicnoret_max_global_rlx_system_s64 [&x], 23;
atomicnoret_max_group_rlx_wg_u64 [&x], 23;
atomicnoret_max_rlx_system_s64 [$d0], 23;

atomicnoret_min_global_scar_system_u64 [&x], 23;
atomicnoret_min_global_rlx_system_s64 [&x], 23;
atomicnoret_min_group_rlx_wg_u64 [&x], 23;
atomicnoret_min_rlx_wg_s64 [$d0], 23;

```

6.8 Notification (signal) Instructions

Signal instructions 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 instructions, the HSA platform architecture signals allow implementations to optimize for power and performance during signal instructions. For example, spin loops involving atomic memory instructions can be replaced with signal wait instructions 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 Version 1.0* section 2.8 *Requirement: 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 instructions.

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 instructions.

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 \(page 39\)](#)). 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 instructions provided by the HSA runtime and by the HSAIL signal operations described in this section. The results are undefined if signal value is accessed or updated by any other operation, including both ordinary and atomic memory instructions. A signal instruction specifies the size of the signal value. A signal instruction is undefined if the signal handle provided does not reference a signal with the same size of signal value as specified by the signal instruction.

Signals are generally intended for notification between agents. Therefore, signal instructions interact with the memory model (see [6.2. Memory Model \(page 169\)](#)) as if the signal value resides in global segment memory, is naturally aligned (see [6.1.3. Alignment \(page 168\)](#)) and is accessed using atomic memory instructions at system scope. However, an implementation is permitted to allocate the signal value in any memory, provided all instructions interact with the memory model as if it was allocated in global segment memory.

Signal instructions allow a memory ordering to be specified which is used by the atomic memory instruction that accesses the signal value. The memory ordering affects how other memory instructions 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` instructions. Note that these instructions 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` instruction, but again this is the address of the signal handle, not the signal.

A signal handle defined as a global or readonly segment variable can have an initializer. A signal handle type constant uses the typed constant notation (see [4.8.3. Typed Constants \(page 87\)](#)): a signal handle type, followed by an integer constant in parentheses. Only an integer constant with the value 0 is allowed, which represents the null signal handle. The rules for using signal handle typed constants are the same as other typed constants (see [4.8.5. How Text Format Constants Are Converted to Bit String Constants \(page 92\)](#)):

- When initializing a signal handle type variable without an array dimension, a signal handle typed constant of the same type as the variable must be used.
- When initializing a signal handle type variable with an array dimension, an array typed constant must be used with the same array element type as the variable, the same number of array elements as the variable, and each array element the same signal type as the variable array element type.
- An aggregate constant that includes signal typed constants can be used to initialize bit type array variables. The aggregate constant must have the same byte size as the array variable.

The following is an example of signal handle variable initializations:

```
global_sig32 &name0 = sig32(0);
global_sig32 &namedsig32WithInit[2] = { sig32(0),
                                         sig32(0)
                                         };
global_b8 &namedStructInit[16] = { u32(4),
                                   align(8),
                                   sig32(0)
                                   };

```

6.8.1 Syntax

Table 6–5 Syntax for Signal Instructions

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>

Explanation of Modifiers

order: Memory order used to specify synchronization. Can be `rlx` (relaxed) for all instructions; `scacq` (sequentially consistent acquire) for all instructions except `st`; `screl` (sequentially consistent release) for all instructions except `ld`, `wait` and `waittimeout`; or `scar` (sequentially consistent acquire and release) for all instructions except `st`, `ld`, `wait` and `waittimeout`. See [6.2.1. Memory Order \(page 169\)](#).

waitOp: The comparison operation to perform. Can be `eq` (equal) `ne` (not equal), `lt` (less than) and `gte` (greater than or equal).

Type: `b` for `ld`, `st`, `and`, `or`, `xor`, `exch`, `cas`; `u` and `s` for `add`, `sub`; `s` for `wait`, `waittimeout`. See [Table 4–2 \(page 99\)](#).

Length: 32, 64. See [Table 4–2 \(page 99\)](#). Must match the signal value size of *signalType*. See [2.9. Small and Large Machine Models \(page 39\)](#).

signalType: `sig32`, `sig64`. See [Table 4–4 \(page 101\)](#). Must be `sig32` for small machine model and `sig64` for large machine model. See [2.9. Small and Large Machine Models \(page 39\)](#).

Explanation of Operands (see [4.16. Operands \(page 104\)](#))

dest: Destination register of type *TypeLength*.

signalHandle: A source operand `d` register that contains a value of a signal handle of type *signalType*. The results are 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 or immediate value.

timeout: Timeout value of type `u64`. Specified in same units as the system timestamp. Can be a register or immediate value.

Exceptions (see [Chapter 12. Exceptions \(page 269\)](#))

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 Instructions \(page 354\)](#).

6.8.2 Description of Signal Instructions

`ld, st, and, or, xor, exch, add, sub, cas`

The `signal` instructions have the same definition as the corresponding `atomic` instructions, 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 \(atomic\) Instructions \(page 181\)](#).

The `signalnoret` instructions have the same definition as the corresponding `atomicnoret` instructions in a similar manner. See [6.7. Atomic No Return \(atomicnoret\) Instructions \(page 185\)](#)

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` instruction 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` instruction, 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 instruction resumes, the last signal value read is returned in `dest` operand.

A wait instruction 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 a function to obtain this value. Additionally, a wait instruction 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 instruction 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 instruction may suspend thread or work-item execution, and resume it in response to another signal instruction that changes the value of a signal value.

Since the wait instruction can return spuriously, it is necessary to test the returned value to see if the condition was met. For this reason a wait instruction 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 instruction 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 wavefront, or a work-item later in the flattened work-item order, as deadlock may occur.

The signal values seen by a wait instruction are guaranteed to make forward progress in the modification order of the signal value memory location. However, it is not guaranteed that the wait instruction 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 instruction, but the wait instruction does not observe it before it is changed to a value that does not satisfy its condition, and therefore the wait instruction 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 instruction is not required to resume immediately that the signal value satisfies the condition, even if the wait instruction 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 waitOpsrc0) && !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_b64_sig64 $d2, $d0, 23;
signal_and_rlx_b32_sig32 $s2, $d1, 23;

signal_or_scar_b64_sig64 $d2, $d0, 23;
signal_or_srel_b32_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_u64_sig64 $d2, $d0, 23;
signal_add_rlx_s32_sig32 $s2, $d1, 23;

signal_sub_scar_u64_sig64 $d2, $d0, 23;
signal_sub_rlx_s32_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_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) Instruction

The memory fence (`memfence`) instruction can either be a release memory fence, an acquire memory fence, or both an acquire and a release memory fence. `memfence` instructions are synchronizing memory operations. See [6.2. Memory Model \(page 169\)](#).

6.9.1 Syntax

Table 6–6 Syntax for `memfence` Instruction

Opcode and Modifier
<code>memfence_order_scope</code>
Explanation of Modifier
<i>order</i> : 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). See 6.2.1. Memory Order (page 169) .
<i>scope</i> : Memory scope used to specify synchronization. Can be <code>wave</code> (wavefront), <code>wg</code> (work-group), <code>agent</code> (kernel agent) or <code>system</code> (system). See 6.2.2. Memory Scope (page 170) .
Exceptions (see Chapter 12. Exceptions (page 269))
No exceptions are allowed.

For BRIG syntax, see [18.7.2. BRIG Syntax for Memory Instructions \(page 354\)](#).

6.9.2 Description

The `memfence` instruction allows memory access and updates to be synchronized between work-items and other agents for the global and group segments. See [6.2. Memory Model \(page 169\)](#).

For example:

```

st_global_u32 1, [&x];
memfence_screl_system; // Will ensure 1 is visible to work-items that
                        // subsequently perform an acquire at system scope.

```

The `memfence` instruction can be used in conditional code.

Examples

```
memfence_scacq_system;  
memfence_screl_wg;  
memfence_scacq_agent;  
memfence_scar_wave;
```

CHAPTER 7.

Image Instructions

This chapter describes how images and samplers are used in HSAIL and also describes the associated read, load, store, memory fence and query instructions.

The image operations defined in this chapter are only allowed if the "IMAGE" extension directive has been specified. See [13.1.2. extension IMAGE \(page 274\)](#).

The minimum limits with respect to images are specified in [Appendix A. Limits \(page 374\)](#).

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
 - 5. Image Addressing and Filtering
 - <http://www.khronos.org/registry/cl/specs/opencl-2.0-opencl.c.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*. See [7.1.3. Image Geometry \(next page\)](#).

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 \(page 198\)](#).

There can be implementation dependent restrictions on how an image can be accessed and there is a minimum set of required access permissions for different image formats and geometries. See [7.1.5. Image Access Permission \(page 204\)](#).

Images are accessed using image coordinates. See [7.1.6. Image Coordinate \(page 206\)](#).

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 instructions using an opaque image handle. See [7.1.7. Image Creation and Image Handles \(page 211\)](#).

The `rdimage` image instruction 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 instructions using an opaque sampler handle. See [7.1.8. Sampler Creation and Sampler Handles \(page 214\)](#).

There are a set of image instructions 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 Instructions \(page 216\)](#).

The image memory model defines the interaction of image operations between different work-items and other agents. See [7.1.10. Image Memory Model \(page 218\)](#).

7.1.3 Image Geometry

Each image has an associated geometry. See [Table 7–1 \(below\)](#) for a list of the image geometries supported.

Table 7–1 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	rdimage_1d, ldimage_1d, stimage_1d	one-dimensional image
2d	width	height	unused		rdimage_2d, ldimage_2d, stimage_2d	two-dimensional image
3d	width	height	depth		rdimage_3d, ldimage_3d, stimage_3d	three-dimensional image
1da	width	array index	unused		rdimage_1da, ldimage_1da, stimage_1da	one-dimensional image array
2da	width	height	array index		rdimage_2da, ldimage_2da, stimage_2da	two-dimensional image array
1db	width	unused	unused		ldimage_1db, stimage_1db	one-dimensional image buffer
2ddepth	width	height	unused	depth, depth_stencil	rdimage_2ddepth, ldimage_2ddepth, stimage_2ddepth	two-dimensional depth image
2dadepth	width	height	array index		rdimage_2dadepth, ldimage_2dadepth, stimage_2dadepth	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 indices. It can be addressed by two coordinates (x , y) corresponding to the width and array indices, 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 indices, 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. Consequently the `rdimage` image instruction does not support 1DB images.

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 instructions is undefined unless the image segment, which is used when image instructions access image data, is made coherent with the global segment by appropriate acquire and release memory fences. See [7.1.10. Image Memory Model \(page 218\)](#).

2DDEPTH

Same as the 2D geometry except the image instructions 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 instructions 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. For example, all the core features of *The OpenCL Specification Version 2.0* are supported. The BRIG enumeration for geometry includes additional geometry values that can be used by extensions. See [18.3.13. BrigImageGeometry \(page 305\)](#).

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 instructions 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.

Each channel order has an associated border color that is used as the access value by some coordinate addressing modes when an image is accessed by out of range coordinates. For the `depth` and `depth_stencil` channel orders it is implementation defined if the border color is (0) or (1). (See [7.1.6.2. Addressing Mode \(page 207\)](#)).

NOTE: The *OpenCL Extension Specification Version 1.2* specifies that the border color of depth images is (0) while the core *OpenCL Specification Version 2.0* defines it as (1). A future version of HSAIL may define the value that must be used when this inconsistency has been resolved.

See [Table 7-2 \(facing page\)](#) for a list of the channel orders supported and their associated border colors.

Table 7–2 Channel Order Properties

Channel Order	Memory Components	Access Components	Border Color	Channel Types	Image Geometries
a	(a)	(0,0,0,a)	(0,0,0,0)	snorm_int8, snorm_int16, unorm_int8, unorm_int16, signed_int8, signed_int16, signed_int32, unsigned_int8, unsigned_int16, unsigned_int32, half_float, float	1D, 2D, 3D, 1DA, 2DA, 1DB
r	(r)	(r,0,0,1)	(0,0,0,1)		
rx	(r)	(r,0,0,1)	(0,0,0,0)		
rg	(r,g)	(r,g,0,1)	(0,0,0,1)		
rgx	(r,g)	(r,g,0,1)	(0,0,0,0)		
ra	(r,a)	(r,0,0,a)	(0,0,0,0)		
rgba	(r,g,b,a)	(r,g,b,a)	(0,0,0,0)		
rgb	(r,g,b)	(r,g,b,1)	(0,0,0,1)	unorm_short_565, unorm_short_555, unorm_int_101010	
rgbx	(r,g,b)	(r,g,b,1)	(0,0,0,0)		
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. The conversion is done before computing the weighted average when a sampler with linear filtering is used.</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)		
intensity	(i)	(i,i,i,i)	(0,0,0,0)	unorm_int8, unorm_int16, snorm_int8, snorm_int16, half_float, float	2DDEPTH, 2DADEPTH
luminance	(L)	(L,L,L,1)	(0,0,0,1)		
depth	(d)	(d)	<i>implementation defined if (0) or (1)</i>	unorm_int16, unorm_int24, float	
depth_stencil	(d,s)	(d)	<i>defined if (0) or (1)</i>	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 an `rgba` 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`, `sbggra` 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 (below)), and then the resulting sRGB access values are converted to 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 less than or equal to 0.5 ULP (see 4.19.6. Unit of Least Precision (ULP) (page 112)) 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_componentis 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 signaling 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 accesses 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 convert from the component memory type to the component access type by image read instructions, and from the component access type to the component memory type by image write instructions. See [Table 7-3 \(below\)](#) for a list of the channel types supported together with their properties.

Table 7-3 Channel Type Properties

Channel Type	Memory Type		Access Type	Conversion Method
	Bit Size	Encoding		
<code>snorm_int8</code>	8	signed integer	<code>f32</code>	<code>SignedNormalize(2⁷-1)</code>
<code>snorm_int16</code>	16	signed integer		<code>SignedNormalize(2¹⁵-1)</code>
<code>unorm_int8</code>	8	unsigned integer		<code>UnsignedNormalize(2⁸-1)</code>
<code>unorm_int16</code>	16	unsigned integer		<code>UnsignedNormalize(2¹⁶-1)</code>
<code>unorm_int24</code>	24	unsigned integer		<code>UnsignedNormalize(2²⁴-1)</code>
<code>unorm_short_565</code>	<code>r=5 bits[15:11]</code>	unsigned integer		<code>UnsignedNormalize(2⁵-1)</code>
	<code>g=6 bits[10:05]</code>			<code>UnsignedNormalize(2⁶-1)</code>
	<code>b=5 bits[04:00]</code>			<code>UnsignedNormalize(2⁵-1)</code>
<code>unorm_short_555</code>	<code>r=5 bits[14:10]</code>	unsigned integer		
	<code>g=5 bits[09:05]</code>			
	<code>b=5 bits[04:00]</code>			
	<code>ignored bit[15]</code>			
<code>unorm_int_101010</code>	<code>r=10 bits[29:20]</code>	unsigned integer		<code>UnsignedNormalize(2¹⁰-1)</code>
	<code>g=10 bits[19:10]</code>			
	<code>b=10 bits[09:00]</code>			
	<code>ignored bits[31:30]</code>			
<code>signed_int8</code>	8	signed integer	<code>s32</code>	<code>SignedClamp(-2⁷, 2⁷-1)</code>
<code>signed_int16</code>	16	signed integer		<code>SignedClamp(-2¹⁵, 2¹⁵-1)</code>
<code>signed_int32</code>	32	signed integer		<code>Identity()</code>
<code>unsigned_int8</code>	8	unsigned integer	<code>u32</code>	<code>UnsignedClamp(2⁸-1)</code>
<code>unsigned_int16</code>	16	unsigned integer		<code>UnsignedClamp(2¹⁶-1)</code>
<code>unsigned_int32</code>	32	unsigned integer		<code>Identity()</code>
<code>half_float</code>	16	float	<code>f32</code>	<code>HalfFloat()</code>
<code>float</code>	21	float		<code>Float()</code>

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 \(page 109\)](#)). For the packed representations of `unorm_short_555`, `unorm_short_565`, and `unorm_int_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 instructions that specify the image component (see [Table 4-2 \(page 99\)](#)).

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 instruction, it is implementation defined if subnormal values are flushed to zero, if NaN values are propagated or payloads preserved (regardless of the profile specified) or if signaling NaNs are converted to quiet NaNs (see [4.19.4. Not A Number \(NaN\) \(page 111\)](#)). All other values are preserved. No invalid operation exception is generated if the value is a signaling NaN. No inexact exception is generated.

HalfFloat()

On a read image instruction, 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 or payloads preserved (regardless of the profile specified) or if signaling NaNs are converted to quiet NaNs (see [4.19.4. Not A Number \(NaN\) \(page 111\)](#)). No invalid operation exception is generated if the value is a signaling NaN.

On write image instructions, 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. Floating-Point Rounding \(page 109\)](#)). 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 or payloads preserved (regardless of the profile specified) or if signaling NaNs are converted to quiet NaNs (see [4.19.4. Not A Number \(NaN\) \(page 111\)](#)). No invalid operation exception is generated if the value is a signaling 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 instruction, the access component is set to the memory component value zero extended to `u32`.

On write image instructions, 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 instruction, the access component is set to the memory component value sign extended to `s32`.

On write image instructions, 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 interval [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 instruction, 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 (see [4.19.6. Unit of Least Precision \(ULP\)](#) (page 112)), 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 instructions, the memory component value is set to:

```
memory_component = min(max(intneari(access_component * float(scale)), 0), scale);
```

The conversion to integer uses *neari* rounding mode (see [5.19.4. Description of Integer Rounding Modes](#) (page 162)). 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 signaling 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 instruction, 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 instructions, the memory component value is set to:

```
memory_component = min(max(intneari(access_component * float(scale)), -scale - 1), scale);
```

The conversion to integer uses *neari* rounding mode (see [5.19.4. Description of Integer Rounding Modes](#) (page 162)). 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 signaling 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 7-4 \(facing page\)](#).

7.1.5 Image Access Permission

The image access permissions refer to how an image can be accessed using image instructions. If the access permissions of a specific image include:

- read-only, then image read instructions are allowed
- write-only, then write instructions are allowed
- read-write, then both read and write instructions 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 kernel agent. However, for every kernel agent 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 kernel agent. It is undefined if an image instruction requires an access permission not supported by the kernel agent for a specific image. See [Table 7-4 \(facing page\)](#) 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 a kernel agent 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 a kernel agent 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 a kernel agent that supports images is required to support the combination for the read-write access permission. Otherwise, it may optionally support it if legal.

Table 7–4 Channel Order, Channel Type, and Image Geometry Combination

Channel Order	Channel Type						Image Geometry	bpp
	Bits	unorm	snorm	uint	sint	float		
r	8	Y (ro,wo,rw)	Y (ro,wo)	Y (ro,wo,rw)	Y (ro,wo,rw)		1D, 2D, 3D, 1DA, 2DA, 1DB	8
	16	Y (ro,wo)	Y (ro,wo)	Y (ro,wo,rw)	Y (ro,wo,rw)	Y (ro,wo,rw)		16
	32			Y (ro,wo,rw)	Y (ro,wo,rw)	Y (ro,wo,rw)		32
rx, a	8	Y	Y	Y	Y			8
	16	Y	Y	Y	Y	Y		16
	32			Y	Y	Y		32
rg	8, 8	Y (ro,wo)	Y (ro,wo)	Y (ro,wo)	Y (ro,wo)			16
	16, 16	Y (ro,wo)	Y (ro,wo)	Y (ro,wo)	Y (ro,wo)	Y (ro,wo)		32
	32, 32			Y (ro,wo)	Y (ro,wo)	Y (ro,wo)		64
rgx, ra	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	Y						16
	5, 5, 5, 1	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		2DDEPTH, 2DADEPTH	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)						16
	24	Y						24
	32					Y (ro,wo)		32
depth_stencil	24, 8	Y						32
	32					Y		64

7.1.6 Image Coordinate

Image instructions 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 7-1 \(page 196\)](#).

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 instruction (see [7.1.8. Sampler Creation and Sampler Handles \(page 214\)](#)). For the `ldimage` and `stimage` image instructions, fixed modes are used (see [7.1.6.3. Filter Mode \(page 209\)](#)). The 1DB image geometry does not support samplers and so cannot be used with the `rdimage` image instruction.

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 \(page 209\)](#)). 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 \(facing page\)](#)).

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 instructions which only allow `nearest` filter mode and `unnormalized` coordinate normalization mode.

`s32`

This can be used when the sampler for the `rdimage` image instruction 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 instruction specifies an `unnormalized` coordinate normalization mode. It is also used for the array index coordinate for the `rdimage` image instruction 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 $\text{coord}_{\text{dim}}$ is the size of the corresponding dimension. It is specified as an f32 coordinate data type value. The value is multiplied by $\text{coord}_{\text{dim}}$ to determine the image element index. This is used for non-array index coordinates when the sampler for the `rdimage` image instruction 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 (coordis_array_index ? unnormalized : normalization_mode) {
  case unnormalized:
    switch (coorddata_type) {
    case u32:
      switch (filter_mode) {
      case nearest: return float(coord);
      }
    case s32:
      switch (coordis_array_index ? nearest : filter_mode) {
      case nearest: return float(coord);
      case linear: return float(coord) - 0.5;
      }
    case f32:
      if (coordis_nan or coordis_infinity) return is_undefined;
      switch (coordis_array_index ? nearest : filter_mode) {
      case nearest: return coord;
      case linear: return coord - 0.5;
      }
    }
  }
  case normalized:
    switch (coorddata_type) {
    case f32:
      if (coordis_nan or coordis_infinity) return is_undefined;
      switch (filter_mode) {
      case nearest: return coord * coorddim;
      case linear: return (coord * coorddim) - 0.5;
      }
    }
  }
}
```

7.1.6.2 Addressing Mode

The coordinate addressing mode controls how out of range coordinates are processed:

undefined

The image instruction is undefined if the coordinate value is out of range.

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 instruction 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.

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 7-2 \(page 199\)](#)).

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. The results are undefined if `mirrored_repeat` addressing mode is specified unless the normalization mode is `normalized`.

The `undefined` mode is always used for all coordinates of the `stimage`, and for non-array index coordinates of the `ldimage` image instructions.

The `clamp_to_edge` mode is always used by the `rdimage` image instruction for an array index coordinate regardless of the addressing mode specified by the sampler.

It is implementation defined whether the `ldimage` image instruction always uses the `undefined` or `clamp_to_edge` mode for an array index coordinate.

NOTE: A future version of HSAIL may define the mode that must be used when the ambiguity in the *OpenCL Specification Version 2.0* has been resolved.

The conversions to an integer image element index for non-array index coordinates uses `downi`, whereas `neari` is used for array index coordinates (see [5.19.4. Description of Integer Rounding Modes \(page 162\)](#)).

The addressing mode is computed as follows:

```
addressing(coord) {
    if (coordis_undefined) return is_undefined;
    out_of_range = (intdowni(coord) < 0) or (intdowni(coord) > coorddim - 1);
    if (coordis_array_index) {
        if (out_of_range and
            ((operation == stimage) or
             ((operation == ldimage) and implementation_definedis_ldimage_array_index_out_of_range_undefined))
            return is_undefined;
        if ((operation == ldimage) and out_of_range and return is_undefined;
        return max(0, min(intneari(coord), coorddim - 1));
    }
    if ((normalization_mode == unnormalized) and
        ((addressing_mode == repeat) or (addressing_mode == mirrored_repeat))
        ) return is_undefined;
    if (not out_of_range) return intdowni(coord);
    switch(addressing_mode) {
    case undefined:      return is_undefined;
    case clamp_to_edge:  return intdowni(max(0, min(coord, coorddim - 1)));
    case clamp_to_border: return is_border;
    case repeat:
        tile = intdowni(coord / coorddim);
        return intdowni(coord - (tile * coorddim));
    case mirrored_repeat:
        mirrored_coord = (coord < 0) ? (-coord - 1) : coord;
        tile = intdowni(mirrored_coord / coorddim);
        mirrored_coord = intdowni(mirrored_coord - (tile * coorddim));
        if (tile & 1) {
            mirrored_coord = (coorddim - 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 instructions ignore any coordinate indices that are unused by the image geometry (see [Table 7-1 \(page 196\)](#)). Of the used coordinate indices, if any are *is_undefined*, then the image instruction 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 7-2 \(page 199\)](#)). 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 orders). 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 orders) before storing in the image element. See [Table 7-3 \(page 201\)](#) and [7.1.4.1.2. Standard RGB \(s-Form\) Channel Orders \(page 200\)](#).

`load_texel` and `store_texel` map between memory components and access components as shown in [Table 7-2 \(page 199\)](#). If the image channel order has fewer than four memory components:

- `load_texel` returns the fixed value from [Table 7-2 \(page 199\)](#) for any missing memory components
- `store_texel` ignores any access components that have no corresponding memory component

The coordinate properties used by each image instruction 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 (page 214)).

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_frac   = normalization(x) - floor(normalization(x));
y0_index = addressing(normalization(y));
y1_index = addressing(normalization(y) + 1);
y_frac   = normalization(y) - floor(normalization(y));
z0_index = addressing(normalization(z));
z1_index = addressing(normalization(z) + 1);
z_frac   = normalization(z) - floor(normalization(z));
switch (geometry) {
case 1d:
case 1da:
    return (1 - x_frac) * load_texel(x0_index, y0_index, z0_index)
           + x_frac    * load_texel(x1_index, y0_index, z0_index);
case 2d:
case 2da:
case 2ddepth:
case 2dadepth:
    return (1 - x_frac) * (1 - y_frac) * load_texel(x0_index, y0_index, z0_index)
           + x_frac    * (1 - y_frac) * load_texel(x1_index, y0_index, z0_index)
           + (1 - x_frac) * y_frac    * load_texel(x0_index, y1_index, z0_index)
           + x_frac    * y_frac    * load_texel(x1_index, y1_index, z0_index);
case 3d:
    return (1 - x_frac) * (1 - y_frac) * (1 - z_frac)
           * load_texel(x0_index, y0_index, z0_index)
           + x_frac    * (1 - y_frac) * (1 - z_frac)
           * load_texel(x1_index, y0_index, z0_index)
           + (1 - x_frac) * y_frac    * (1 - z_frac)
           * load_texel(x0_index, y1_index, z0_index)
           + x_frac    * y_frac    * (1 - z_frac)
           * load_texel(x1_index, y1_index, z0_index)
           + (1 - x_frac) * (1 - y_frac) * z_frac
           * load_texel(x0_index, y0_index, z1_index)
           + x_frac    * (1 - y_frac) * z_frac
           * load_texel(x1_index, y0_index, z1_index)
           + (1 - x_frac) * y_frac    * z_frac
           * load_texel(x0_index, y1_index, z1_index)
           + x_frac    * y_frac    * z_frac
           * 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 instructions can be used to convert to unnormalized coordinates, and to perform the equivalent of any linear filter mode using component values accessed by image instructions 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 7-4 \(page 205\)](#)). The maximum image size supported for an image geometry, and the maximum number of image handles that can exist at any one time for a specific access permission, is implementation defined for each agent, but there are minimum limits that all agents must support (see [Appendix A. Limits \(page 374\)](#)). An HSA runtime query is available to obtain the maximum limits supported by an agent.

The HSA runtime can be used to destroy an image handle which reduces the number of created handles. The results are undefined if an image handle is used after it has been destroyed.

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.

The results are undefined if multiple image handles are created 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 \(page 200\)](#)).
- 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 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.

In HSAIL there are three opaque image handle types, `roimg`, `woimg` and `rwimg` (see [Table 4–4 \(page 101\)](#)). These correspond to the three image access permissions (see [7.1.5. Image Access Permission \(page 204\)](#)). See [Table 7–5 \(below\)](#).

- 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 7–5 Image Handle Properties

Image Handle Type	Image Access Permission	Image Instructions
<code>roimg</code>	ro	<code>rdimage</code> , <code>ldimage</code>
<code>woimg</code>	wo	<code>stimage</code>
<code>rwimg</code>	rw	<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 instructions `rdimage`, `ldimage` and `stimage`, not through the memory instructions `ld`, `st`, `atomic`, or `atomicnoret`. The results are undefined if an image handle is used 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 instructions on a kernel agent for which the image handle was not created, or with an image handle that has an access permission that is not supported by the kernel agent for the image's properties. Different kernel agents may use different representations for image handles, and their image instructions may not be able to access each other's image data allocations. Also see [7.1.10. Image Memory Model \(page 218\)](#).

An image handle variable 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.

A variable definition in the global or readonly segment can have an initializer that defines the properties of the image. For 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 \(page 69\)](#).

An image handle type constant uses the typed constant notation (see [4.8.3. Typed Constants \(page 87\)](#)): an image handle type, followed by a parenthesized list containing pairs of `keyword = value`. 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.

An image handle typed constant can be used in a variable initializer, but cannot be used in an immediate source operand. The rules for using image handle typed constants are the same as other typed constants (see [4.8.5. How Text Format Constants Are Converted to Bit String Constants \(page 92\)](#)):

- When initializing an image handle type variable without an array dimension, an image handle typed constant of the same type as the variable must be used.
- When initializing an image handle type variable with an array dimension, an array typed constant must be used which has the same array element type as the variable, the same number of array elements as the variable, and each array element the same image type as the variable array element type.
- An aggregate constant that includes image typed constants can be used to initialize bit type array variables. The aggregate constant must have the same byte size as the array variable.

The following is an example of image handle variable initializations:

```
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_rwing &namedrwing12;
decl prog global_rwing &namedrwing2;
decl prog global_rwing &namedrwing3;
decl prog global_rwing &ArrayOfrwings[10];
alloc(agent) global_rwing &namedrwingWithInit[2] =
    rwing[( rwing(geometry = 3d,
                  width = 5, height = 4, depth = 6,
                  channel_type = unorm_int_101010,
                  channel_order = rgbx),
            rwing(geometry = 2d,
                  width = 5, height = 4,
                  channel_type = unorm_short_555,
```

```

                                channel_order = rgb)
                                );
alloc(agent) global_b8 &namedStructInit[16] =
{ u32(4),
  align(8),
  rwimg(geometry = 2d,
         width = 5, height = 4,
         channel_type = unorm_short_555,
         channel_order = rgb)
};

```

When a code object, that references a variable definition that has an initializer which includes any image handle typed constants, is loaded into an executable for a kernel agent, images with the specified properties are created for that kernel agent if it supports images. The kernel agent's agent allocation variable is allocated and the image handles initialized to reference the corresponding images. The associated image data is not initialized. When the executable is destroyed, the images, image data and image handles are destroyed. See [4.2. Program, Code Object, and Executable \(page 48\)](#).

The `queryimage` instruction can be used to query the properties of an image. See [7.5. Query Image and Query Sampler Instructions \(page 224\)](#).

7.1.8 Sampler Creation and Sampler Handles

Samplers are used to specify how to process image coordinates by the `rdimage` image instruction (see [7.1.6. Image Coordinate \(page 206\)](#)).

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 maximum number of sampler handles that can exist at any one time is implementation defined for each agent, but there are minimum limits that all agents must support (see [Appendix A. Limits \(page 374\)](#)). 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 4-4 \(page 101\)](#)). It is undefined to use HSAIL sampler operations on a kernel agent for which the sampler handle was not created. Different kernel agents may use different representations for sampler handles.

A sampler handle variable 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 an initializer that defines the properties of the sampler. For 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 \(page 69\)](#).

A sampler handle type constant uses the typed constant notation (see [4.8.3. Typed Constants \(page 87\)](#)): `samp`, followed by a parenthesized list containing pairs of `keyword = value`. 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`.

A sampler handle typed constant can be used in a variable initializer, but cannot be used in an immediate source operand. The rules for using sampler handle typed constants are the same as other typed constants (see [4.8.5. How Text Format Constants Are Converted to Bit String Constants \(page 92\)](#)):

- When initializing a sampler handle type variable without an array dimension, a sampler handle typed constant must be used.
- When initializing a sampler handle type variable with an array dimension, an array typed constant must be used which has a sampler handle array element type, the same number of array elements as the variable, and each array element a sampler handle typed constant.
- An aggregate constant that includes sampler typed constants can be used to initialize bit type array variables. The aggregate constant must have the same byte size as the array variable.

The following is an example of sampler handle variable initializations:

```
alloc(agent) global_samp &y1 = samp(coord = normalized,
                                   filter = nearest,
                                   addressing = clamp_to_edge);

alloc(agent) global_samp &y2[2] =
    samp[( samp(coord = unnormalized,
               filter = nearest,
               addressing = clamp_to_border),
          samp(coord = normalized,
               filter = linear,
               addressing = mirrored_repeat)
        );

alloc(agent) global_b8 &namedStructInit[16] =
    { u32(4),
      align(8),
      samp(coord = unnormalized,
           filter = nearest,
           addressing = clamp_to_border)
    };
};
```

When a code object, that references a variable definition that has an initializer which includes any sampler handle typed constants, is loaded into an executable for a kernel agent, samplers with the specified properties are created for that kernel agent if it supports images. The kernel agent's agent allocation variable is allocated and the sampler handles initialized to reference the corresponding samplers. When the executable is destroyed, the samplers and sampler handles are destroyed. See [4.2. Program, Code Object, and Executable \(page 48\)](#).

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 (page 209)).

Samplers cannot be used with 1DB images which are not supported by the `rdimage` image instruction.

The `query_sampler` instruction can be used to query the properties of a sampler. See 7.5. Query Image and Query Sampler Instructions (page 224).

7.1.9 Using Image Instructions

The image instructions are listed in Table 7-6 (below).

- It is undefined to use an image instruction with an image geometry modifier that does not match the geometry of the image. See Table 7-1 (page 196).
- It is undefined to use the image instructions with a combination of image handle type, coordinate type, access type, image geometry and sampler properties not listed in Table 7-6 (below).
- It is undefined to use the image instructions on an image with a channel order, channel type and image geometry not specified in Table 7-4 (page 205).
- It is undefined if the access type of the image instruction does not match the access type required by the image's channel type specified in Table 7-4 (page 205).

Table 7-6 Image Instruction Combinations

Image Instruction	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, rwing	u32	u32, s32, f32	Sampler not allowed (undefined if coordinate not in range 0 to dimension size - 1)			1D, 2D, 3D, 1DA, 2DA, 1DB, 2DDEPTH, 2DADEPTH
stimage	woimg, rwing						

To access the data in an image, an image handle is loaded into a `d` register using a `load (ld)` instruction with a source type of `roimg`, `woimg` or `rwing`. 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)` instructions.

The differences between the `rdimage` instruction and the `ldimage` instruction 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 instruction by using an opaque sampler handle which is loaded into a `d` register with a source type of `samp`.

An image handler or sampler handle in a `d` register can be:

- Moved to another `d` register using the move (`mov`) instruction with the corresponding `roimg`, `woimg`, `rwimg` or `samp` type.
- Stored to an `arg` segment variable using a store (`st`) instruction with the corresponding `roimg`, `woimg`, `rwimg` or `samp` type. The `arg` segment variable must be:
 - An input actual argument of a call instruction 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 instruction is not allowed on any other segment. This restriction ensures that the actual image or sampler used by an image instruction can be statically determined if function calls are inlined. Note, true bindless textures are not supported.

The results are undefined if the `d` register used in an image instruction does not contain a value that ultimately originated from a global, readonly, or kernel argument variable. For image handles, the original value type (`roimg`, `woimg`, or `rwimg`) must match the type of all instructions that use the value. For sampler handles, the original variable and all instructions that use the value must specify the sampler handle type (`samp`). These instructions include load (`ld`), store (`st`), move (`mov`), the image instructions (`rdimage`, `ldimage` and `stimage`), and the image and sampler query instructions (`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`) instructions with the type of the corresponding function argument.

The results are undefined if an image instruction (`rdimage`, `ldimage`, and `stimage`) or `queryimage` instruction with an image handle value that is not compatible. The image handle is compatible if:

- It is currently created by the HSA runtime for the agent executing the kernel dispatch.
- The image handle was created with an image access permission that corresponds to the image handle type (`roimg`, `woimg`, or `rwimg`) of the image instruction. See [Table 7-5 \(page 212\)](#).

It is undefined to use an `rdimage` instruction or `samplerquery` instruction with a sampler handle value that is not currently created by the HSA runtime for the agent. See [7.1.8. Sampler Creation and Sampler Handles \(page 214\)](#).

The address of an image or sampler handle variable can be taken using the `lda` instruction. This allows them to be passed by reference. The results are undefined if the address returned is used by a load or store instruction that does not specify the same type as the original image handle or sampler handle. Note that this is the address of a handle variable: in the case of an image handle, it is neither the address of the image nor the address of the image data; and in the case of a sampler handle, it is not the address of the sampler.

7.1.10 Image Memory Model

This section maps the HSAIL image instructions to the HSA Image Memory Model defined in the *HSA Platform System Architecture Specification Version 1.0* section 2.15 *Requirement: Images*. It also provides an overall informal definition of the memory model.

1. It is undefined to use an image or sampler handle 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 kernel agent 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 7-5 \(page 212\)](#).
2. The image elements accessed by an `rdimage` instruction with a sampler with a `linear` filter mode includes all locations accessed to perform the weighted average (see [7.1.6.3. Filter Mode \(page 209\)](#)).
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.
 - b. It is undefined to access the same image element using both image instructions (which use the image segment) and memory instructions using the global segment, unless all accesses 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 `imagefence` instruction (see [6.9. Memory Fence \(memfence\) Instruction \(page 192\)](#)).
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 an `imagefence` instruction after the write, followed by a `barrier` or `wavebarrier` that both A and B participate, followed by A executing an `imagefence` before the read (see [7.6. Image Fence \(imagefence\) Instruction \(page 225\)](#)).
 - b. An `imagefence` instruction cannot be reordered across a `barrier` or `wavebarrier` in either direction.
 - c. An `imagefence` executed by work-item A that is ordered before a `barrier` or `wavebarrier` will be ordered before any `acquire memfence` that is ordered after the `barrier` or `wavebarrier` that both A and B participate, in work-item B, provided A is a member of the scope instance of the `memfence`.
 - d. A `release memfence` executed by work-item A that is ordered before a `barrier` or `wavebarrier` will be ordered before any `imagefence` that is ordered after the `barrier` or `wavebarrier` that both A and B participate, in work-item B, provided B is a member of the scope instance of the `memfence`.

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 Version 1.0* section 2.9.1 *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 instructions 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 \(page 169\)](#).

7.2 Read Image (rdimage) Instruction

The read image (`rdimage`) instruction uses image coordinates together with a sampler to perform an image memory lookup.

7.2.1 Syntax

Table 7–7 Syntax for Read Image Instruction

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_2ddepth_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 instruction returns 4 components, otherwise only 1 component is returned.	

Explanation of Modifiers
<i>1d</i> , <i>2d</i> , <i>3d</i> , <i>1da</i> , <i>2da</i> , <i>2ddepth</i> , <i>2dadepth</i> : Image geometry. Specifies the number and meaning of coordinates required to access an image element. Can be <i>1d</i> (width); <i>2d</i> or <i>2ddepth</i> (width and height); <i>3d</i> (width, height, and depth); <i>1da</i> (width and array index); or <i>2da</i> or <i>2dadepth</i> (width, height and array index). <i>1db</i> is not supported. See 7.1.3. Image Geometry (page 196) .
<i>equiv</i> (<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, which indicates that any memory location may be aliased. See 6.1.4. Equivalence Classes (page 168) .
<i>destType</i> : Destination type: <i>u32</i> , <i>s32</i> , or <i>f32</i> . See Table 4-2 (page 99) .
<i>imageType</i> : Image object type: <i>roimg</i> . See Table 4-4 (page 101) .
<i>coordType</i> : Source coordinate element type: <i>s32</i> or <i>f32</i> . See Table 4-2 (page 99) .
Explanation of Operands (see 4.16. Operands (page 104))
<i>destR</i> , <i>destG</i> , <i>destB</i> , <i>destA</i> : Destination. Must be an <i>s</i> register.
<i>image</i> : A source operand <i>d</i> register that contains a value of an image object of type <i>imageType</i> . See 7.1.7. Image Creation and Image Handles (page 211) and 7.1.9. Using Image Instructions (page 216) .
<i>sampler</i> : A source operand <i>d</i> register that contains a value of a sampler object. It is always of type <i>samp</i> . See 7.1.8. Sampler Creation and Sampler Handles (page 214) and 7.1.9. Using Image Instructions (page 216) .
<i>coordWidth</i> , <i>coordHeight</i> , <i>coordDepth</i> , <i>coordArrayIndex</i> : A source <i>s</i> register or immediate value of type <i>coordType</i> that specifies the coordinates being read.
Exceptions (see Chapter 12. Exceptions (page 269))
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 Instructions \(page 355\)](#).

Description

The read image (*rdimage*) instruction performs an image memory lookup using image coordinates. The instruction 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 $d3, [&samp1];
rdimage_v4_ld_equiv(12)_s32_roimg_f32 ($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, $s2);
rdimage_v4_lda_s32_roimg_f32 ($s0, $s1, $s2, $s3), $d1, $d3, ($s6, $s7);
rdimage_v4_2da_s32_roimg_f32 ($s0, $s1, $s3, $s4), $d1, $d3, ($s6, $s9, $s12);
```

```
rdimage_2ddepth_s32_roimg_f32 $s0, $d2, $d3, ($s6, $s9);
rdimage_2ddepth_s32_roimg_f32 $s0, $d2, $d3, ($s6, $s9, $s10);
```

7.3 Load Image (ldimage) Instruction

The load image (ldimage) instruction uses image coordinates to load from image memory.

7.3.1 Syntax

Table 7–8 Syntax for Load Image Instruction

Opcode and Modifiers	Operands
ldimage_v4_1d_equiv(n)_destType_imageType_coordType	(<i>destR, destG, destB, destA</i>), <i>image, coordWidth</i>
ldimage_v4_2d_equiv(n)_destType_imageType_coordType	(<i>destR, destG, destB, destA</i>), <i>image, (coordWidth, coordHeight)</i>
ldimage_v4_3d_equiv(n)_destType_imageType_coordType	(<i>destR, destG, destB, destA</i>), <i>image, (coordWidth, coordHeight, coordDepth)</i>
ldimage_v4_1da_equiv(n)_destType_imageType_coordType	(<i>destR, destG, destB, destA</i>), <i>image, (coordWidth, coordArrayIndex)</i>
ldimage_v4_2da_equiv(n)_destType_imageType_coordType	(<i>destR, destG, destB, destA</i>), <i>image, (coordWidth, coordHeight, coordArrayIndex)</i>
ldimage_v4_1db_equiv(n)_destType_imageType_coordType	(<i>destR, destG, destB, destA</i>), <i>image, coordByteIndex</i>
ldimage_2ddepth_equiv(n)_destType_imageType_coordType	<i>destR, image, (coordWidth, coordHeight)</i>
ldimage_2ddepth_equiv(n)_destType_imageType_coordType	<i>destR, image, (coordWidth, coordHeight, coordArrayIndex)</i>

Explanation of Modifiers

v4: If present, specifies the instruction 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 \(page 196\)](#).

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, which indicates that any memory location may be aliased. See [6.1.4. Equivalence Classes \(page 168\)](#).

destType: Destination type: u32, s32, or f32. See [Table 4–2 \(page 99\)](#).

imageType: Image object type: roimg, rwimg. See [Table 4–4 \(page 101\)](#).

coordType: Source coordinate element type: u32. See [Table 4–2 \(page 99\)](#).

Explanation of Operands (see [4.16. Operands \(page 104\)](#))

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 \(page 211\)](#) and [7.1.9. Using Image Instructions \(page 216\)](#).

coordWidth, coordHeight, coordDepth, coordArrayIndex: A source s register or immediate value of type coordType that specifies the coordinates being read.

Exceptions (see [Chapter 12. Exceptions \(page 269\)](#))

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 Instructions \(page 355\)](#).

Description

The load image (`ldimage`) instruction loads from image memory using image coordinates. The instruction 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`. The results are 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` instruction and the `rdimage` instruction 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 $d2, [%roimg2];
ldimage_v4_ld_equiv(12)_f32_rwimg_u32 ($s1, $s2, $s3, $s4), $d1, $s5;
ldimage_v4_2d_f32_rwimg_u32 ($s1, $s2, $s3, $s4), $d1, ($s5, $s6);
ldimage_v4_3d_f32_rwimg_u32 ($s1, $s2, $s3, $s4), $d1, ($s5, $s6, $s7);
ldimage_v4_lda_f32_rwimg_u32 ($s1, $s2, $s3, $s4), $d1, ($s5, $s6);
ldimage_v4_2da_f32_roimg_u32 ($s1, $s2, $s3, $s4), $d2, ($s5, $s6, $s7);
ldimage_v4_ldb_f32_roimg_u32 ($s1, $s2, $s3, $s4), $d2, $s5;
ldimage_2dddepth_f32_rwimg_u32 $s1, $d1, ($s5, $s6);
ldimage_2dadepth_f32_rwimg_u32 $s1, $d1, ($s5, $s6, $s7);
```

7.4 Store Image (stimage) Instruction

The store image (`stimage`) instruction uses image coordinates to store to image memory.

7.4.1 Syntax

Table 7-9 Syntax for Store Image Instruction

Opcode and Modifiers	Operands
stimage_v4_ld_equiv(<i>n</i>)_srcType_imageType_coordType	<i>(srcR, srcG, srcB, srcA), image, coordWidth</i>
stimage_v4_2d_equiv(<i>n</i>)_srcType_imageType_coordType	<i>(srcR, srcG, srcB, srcA), image, (coordWidth, coordHeight)</i>
stimage_v4_3d_equiv(<i>n</i>)_srcType_imageType_coordType	<i>(srcR, srcG, srcB, srcA), image, (coordWidth, coordHeight, coordDepth)</i>
stimage_v4_lda_equiv(<i>n</i>)_srcType_imageType_coordType	<i>(srcR, srcG, srcB, srcA), image, (coordWidth, coordArrayIndex)</i>
stimage_v4_2da_equiv(<i>n</i>)_srcType_imageType_coordType	<i>(srcR, srcG, srcB, srcA), image, (coordWidth, coordHeight, coordArrayIndex)</i>
stimage_v4_ldb_equiv(<i>n</i>)_srcType_imageType_coordType	<i>(srcR, srcG, srcB, srcA), image, coordArrayIndex</i>
stimage_2ddepth_equiv(<i>n</i>)_srcType_imageType_coordType	<i>srcR, image, (coordWidth, coordHeight)</i>
stimage_2dadepth_equiv(<i>n</i>)_srcType_imageType_coordType	<i>srcR, image, (coordWidth, coordHeight, coordArrayIndex)</i>

Explanation of Modifiers
<i>v4</i> : If present, specifies the instruction takes 4 components, otherwise only 1 component is taken.
<i>1d</i> , <i>2d</i> , <i>3d</i> , <i>1da</i> , <i>2da</i> , <i>1db</i> , <i>2ddepth</i> , <i>2dadepth</i> : Image geometry. Specifies the number and meaning of coordinates required to access an image element. Can be <i>1d</i> or <i>1db</i> (width); <i>2d</i> or <i>2ddepth</i> (width and height); <i>3d</i> (width, height, and depth); <i>1da</i> (width and array index); or <i>2da</i> or <i>2dadepth</i> (width, height and array index). See 7.1.3. Image Geometry (page 196) .
<i>equiv(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, which indicates that any memory location may be aliased. See 6.1.4. Equivalence Classes (page 168) .
<i>srcType</i> : Source type: <i>u32</i> , <i>s32</i> , or <i>f32</i> . See Table 4-2 (page 99) .
<i>imageType</i> : Image object type: <i>woimg</i> , <i>rwimg</i> . See Table 4-4 (page 101) .
<i>coordType</i> : Source coordinate element type: <i>u32</i> . See Table 4-2 (page 99) .
Explanation of Operands (see 4.16. Operands (page 104))
<i>srcR</i> , <i>srcG</i> , <i>srcB</i> , <i>srcA</i> : Source. Can be a register or immediate value.
<i>image</i> : A source operand <i>d</i> register that contains a value of an image object of type <i>imageType</i> . See 7.1.7. Image Creation and Image Handles (page 211) and 7.1.9. Using Image Instructions (page 216) .
<i>coordWidth</i> , <i>coordHeight</i> , <i>coordDepth</i> , <i>coordArrayIndex</i> : A source <i>s</i> register or immediate value of type <i>coordType</i> that specifies the coordinates being read.
Exceptions (see Chapter 12. Exceptions (page 269))
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 Instructions \(page 355\)](#).

Description

The store image (*stimage*) instruction stores to image memory using image coordinates. The instruction 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 to use a coordinate that 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, $s5;
stimage_v4_2d_f32_woimg_u32 ($s1, $s2, $s3, $s4), $d1, ($s5, $s6);
```

```

stimage_v4_3d_f32_woimg_u32 ($s1, $s2, $s3, $s4), $d1, ($s5, $s6, $s7);
stimage_v4_1da_f32_rwimg_u32 ($s1, $s2, $s3, $s4), $d2, ($s5, $s6);
stimage_v4_2da_f32_rwimg_u32 ($s1, $s2, $s3, $s4), $d2, ($s5, $s6, $s7);
stimage_v4_1db_f32_rwimg_u32 ($s1, $s2, $s3, $s4), $d2, $s5;
stimage_2ddepth_f32_rwimg_u32 $s1, $d2, ($s5, $s6);
stimage_2dadepth_f32_rwimg_u32 $s1, $d2, ($s5, $s6, $s7);
st_arg_rwimg $d2, [%rwimg_arg1];

```

7.5 Query Image and Query Sampler Instructions

The query image and query sampler instructions query an attribute of an image object or a sampler object.

7.5.1 Syntax

Table 7–10 Syntax for Query Image and Query Sampler Instructions

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, 2ddepth, 2dadepth. See [7.1.3. Image Geometry \(page 196\)](#).

imageProperty: Image property: width, height, depth, array, channelorder, channeltype. 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 7–11 \(below\)](#).

samplerProperty: Sampler property: addressing, coord, filter. See [Table 7–12 \(facing page\)](#).

destType: Destination type: u32. See [Table 4–2 \(page 99\)](#).

imageType: Image object type: roimg, woimg, rwimg. See [Table 4–4 \(page 101\)](#).

Explanation of Operands (see [4.16. Operands \(page 104\)](#))

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 \(page 211\)](#) and [7.1.9. Using Image Instructions \(page 216\)](#).

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 \(page 214\)](#) and [7.1.9. Using Image Instructions \(page 216\)](#).

Exceptions (see [Chapter 12. Exceptions \(page 269\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.3. BRIG Syntax for Image Instructions \(page 355\)](#).

7.5.2 Description

Each query returns a 32-bit value giving a property of the source:

Table 7–11 Explanation of *imageProperty* modifier

<i>imageProperty</i>	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.

<i>imageProperty</i>	Returns
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 18.3.11 . BrigImageChannelOrder (page 304).
channeltype	An image channel type property encoded as an integer according to 18.3.12 . BrigImageChannelType (page 304).

Table 7-12 Explanation of *samplerProperty* modifier

<i>samplerProperty</i>	Returns
addressing	A sampler addressing mode property encoded as an integer according to 18.3.27 . BrigSamplerAddressing (page 312). If <code>undefined</code> 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 <code>undefined</code> .
coord	A sampler coordinate property encoded as an integer according to 18.3.28 . BrigSamplerCoordNormalization (page 312).
filter	A sampler filter property encoded as an integer according to 18.3.29 . BrigSamplerFilter (page 313).

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;
quersampler_addressing_u32 $s0, $d4;
quersampler_coord_u32 $s0, $d4;
quersampler_filter_u32 $s0, $d4;
```

7.6 Image Fence (imagefence) Instruction

The image fence (`imagefence`) instruction synchronizes image operations. See [6.2. Memory Model](#) (page 169) and [7.1.10. Image Memory Model](#) (page 218).

7.6.1 Syntax

Table 7-13 Syntax for `imagefence` Instruction

Opcode and Modifiers
<code>imagefence</code>
Explanation of Modifiers
No modifiers are allowed.
Exceptions (see Chapter 12. Exceptions (page 269))
No exceptions are allowed.

For BRIG syntax, see [18.7.3. BRIG Syntax for Image Instructions](#) (page 355).

Description

The `imagefence` instruction allows image data access and updates to be synchronized both within a single work-item, and, when combined with an execution barrier, between work-items in the same wavefront or work-group. In addition, when combined with `memfence` and execution barriers it can synchronize both image instructions and global and group segment memory instructions. Execution is undefined when memory is accessed without synchronization.

To make the image writes performed by a single work-item visible to the image reads the same work-item performs, it must execute an `imagefence` between the image write and image read instructions. For example:

```
stimage_v4_ld_f32_rwimg_f32 ($s1, $s2, $s3, $s4), $d1, $s4;
imagefence; // Will ensure image data stored by stimage is visible to
            // subsequently ldimage in same work-item.
ldimage_v4_ld_f32_rwimg_f32 ($s5, $s6, $s7, $s8), $d1, $s4;
```

To make the image writes performed by work-item A visible to the image reads performed by work-item B, it is necessary for A to execute an `imagefence` after the image write, followed by a `barrier` or `wavebarrier` that both A and B participate in; and for B to execute an `imagefence` after the `barrier` or `wavebarrier` but before the image reads. For example:

```
stimage_v4_ld_f32_rwimg_f32 ($s1, $s2, $s3, $s4), $d1, $s4;
imagefence;
barrier;
imagefence;
ldimage_v4_ld_f32_rwimg_f32 ($s5, $s6, $s7, $s8), $d1, $s4;
```

Note that this is not enough to ensure an ordering between the image instructions and memory instructions performed by A and B to the global or group segment. To ensure that ordering, it is also necessary for A to perform a release `memfence` after the memory instructions but before the `barrier` or `wavebarrier`, and for B for perform an acquire `memfence` after the `barrier` or `wavebarrier` and before the memory instructions. A and B must both be inclusive members of the scope instances specified by the `memfence` instructions. For example:

```
imagefence;
memfence_rel_wg;
barrier;
memfence_acq_wg;
imagefence;
```

Note that an `fbarrier` cannot be used to achieve synchronization in the current version of HSAIL.

It is not possible to synchronize at a wider scope than work-group except at kernel dispatch granularity by using User Mode Queue packet memory fences.

The `imagefence` instruction can be used in conditional code.

See [7.1.10. Image Memory Model \(page 218\)](#).

Examples

```
imagefence;
```

CHAPTER 8.

Branch Instructions

Like many programming languages, HSAIL supports branch instructions that can alter the control flow.

8.1 Syntax

Table 8–1 Syntax for Branch Instructions

Opcode and Modifier	Operands
br	<i>label</i>
cbr <i>width</i> <i>b1</i>	<i>src</i> , <i>label</i>
sbr <i>width</i> <i>uLength</i>	<i>src</i> [<i>labelList</i>]

Explanation of Modifiers
<i>width</i> : Optional: <i>width</i> (<i>n</i>), <i>width</i> (WAVESIZE), or <i>width</i> (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 <i>width</i> (1), indicating each active work-item can branch independently. See 2.12.2. Using the Width Modifier with Control Transfer Instructions (page 44) .
<i>Length</i> : 32, 64.

Explanation of Operands (see 4.16. Operands (page 104))
<i>src</i> : Source. Can be a register or immediate value.
<i>label</i> : Must be an identifier of a label in the same code block as the branch instruction.
<i>labelList</i> : Must be a comma-separated list of one or more label identifiers that are all in the same code block as the branch instruction.

Exceptions (see Chapter 12. Exceptions (page 269))
No exceptions are allowed.

For BRIG syntax, see [18.7.4. BRIG Syntax for Branch Instructions \(page 356\)](#).

8.2 Description

The label or labels specified in the branch instruction must be in the code block, which includes any nested arg blocks, of the kernel or function containing the branch instruction. However, the label definition can either be lexically before or after the branch instruction. For restrictions on using branches with respect to arg blocks see [10.2. Function Call Argument Passing \(page 244\)](#).

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 instruction after the *cbr* instruction. *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 \(page 41\)](#).

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. The results are 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 \(page 41\)](#).

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 Instructions

This chapter describes instructions used for cross work-item communication.

9.1 Barrier Instructions

The `barrier` and `wavebarrier` instructions are used to synchronize work-item execution in a work-group and wavefront respectively.

9.1.1 Syntax

Table 9–1 Syntax for Barrier Instructions

Opcode and Modifiers
<code>barrier_width</code>
<code>wavebarrier</code>
Explanation of Modifiers
<i>width</i> : Optional: <code>width(n)</code> , <code>width(WAVESIZE)</code> , or <code>width(all)</code> . Used to specify the communication uniformity among the work-items of a work-group. If omitted, defaults to <code>width(all)</code> . See the Description below.
Exceptions (see Chapter 12. Exceptions (page 269))
No exceptions are allowed.

For BRIG syntax, see [18.7.5. BRIG Syntax for Parallel Synchronization and Communication Instructions \(page 356\)](#).

9.1.2 Description

The `barrier` and `wavebarrier` instructions are execution barriers. See [9.3. Execution Barrier \(page 238\)](#).

The `barrier` instruction supports the width modifier:

width

A `barrier` instruction can have an optional width modifier that can specify the communication uniformity (see [2.12. Divergent Control Flow \(page 41\)](#)). 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 wavefront size that is greater than or equal to *n*, it is free to optimize the code 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 instructions 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 \(*fbarrier*\) Instructions \(below\)](#).

Examples

```
barrier;
barrier_width(64);
barrier_width(WAVESIZE);
wavebarrier;
```

9.2 Fine-Grain Barrier (*fbarrier*) Instructions

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 \(page 68\)](#)). 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 \(page 36\)](#)): it has the same allocation and persistence 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 a kernel agent 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* instructions. 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* instructions 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` instruction. This returns a `u32` value in a register that can also be used by fbarrier instructions to specify which fbarrier to operate on.

9.2.2 Syntax

Table 9–2 Syntax for fbar Instructions

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 \(page 41\)](#).

Explanation of Operands (see [4.16. Operands \(page 104\)](#))

src: Either the name of an fbarrier, or an *s* register containing a value produced by an `ldf` instruction. 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 \(page 269\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.5. BRIG Syntax for Parallel Synchronization and Communication Instructions \(page 356\)](#).

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 instruction 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` instruction 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` instructions on the same `fbarrier`. This requirement can be satisfied by using the `barrier` instruction, or the `waitfbar` instruction (on another `fbarrier`) between the `initfbar` and the `fbarrier` instructions that use it.

Once an `fbarrier` has been initialized, its memory cannot be modified by any instruction except `fbarrier` instructions until it is released by an `releasefbar` instruction.

Every `fbarrier` that has been initialized must be released by an `releasefbar` instruction. 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 instruction (which includes the value of the `src` operand) must be wavefront execution uniform (see [2.12. Divergent Control Flow \(page 41\)](#)). This implies that all active work-items of a wavefront must be members of the same `fbarriers`.

`joinfbar` conceptually atomically increments the `member_count` for the wavefront.

A work-item must not join an `fbarrier` that has not been initialized, nor join an `fbarrier` of which it is already a member.

`waitfbar`

Is an execution barrier, see [9.3. Execution Barrier \(page 238\)](#).

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 instruction (which includes the value of the `src` operand) must be wavefront execution uniform (see [2.12. Divergent Control Flow \(page 41\)](#)). This implies that all active work-items of a wavefront arrive at an `waitfbar` together.

`waitfbar` conceptually atomically increments the `arrive_count` for the wavefront, and adds the wavefront to the `wait_set`. It then atomically checks and waits until the `arrive_count` equals the `member_count`, at which point any wavefronts in the `wait_set` are allowed to proceed, the `arrive_count` is reset to 0, and the `wait_set` reset to empty.

A work-item must not wait on an `fbarrier` that has not been initialized, nor wait on an `fbarrier` of which it is not a member.

`arrivefbar`

Is an execution barrier, see [9.3. Execution Barrier \(page 238\)](#).

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 instruction (which includes the value of the `src` operand) must be wavefront execution uniform (see 2.12. Divergent Control Flow (page 41)). This implies that all active work-items of a wavefront arrive at an `arrivefbar` together.

`arrivefbar` conceptually atomically increments the `arrive_count` for the wavefront, and checks if the `arrive_count` equals the `member_count`. If it does, then atomically any wavefronts in the `wait_set` are allowed to proceed, the `arrive_count` is reset to 0, and the `wait_set` is reset to empty.

A work-item must not arrive at an `fbarrier` that has not been initialized, nor arrive at an `fbarrier` of which it is not a member.

After a work-item has arrived at an `fbarrier`, it cannot wait, arrive, or leave the same `fbarrier` unless the `fbarrier` has been satisfied and the `arrive_count` has been reset to 0.

`leavefbar`

Indicates that the work-item is no longer a member of the `fbarrier`. It does not wait for other work-items that are members of the `fbarrier` to arrive. If the work-item is the last of the `fbarrier` members to arrive, then any work-items waiting on the `fbarrier` can proceed and the `fbarrier` is reset.

The `src` operand specifies the `fbarrier` to leave.

Every work-item that joins an `fbarrier` must leave the `fbarrier` before it exits.

A `leavefbar` instruction does not perform a memory fence before proceeding. An explicit `sync` instruction can be used if that is required in order to make any data being communicated visible.

This instruction (which includes the value of the `src` operand) must be wavefront execution uniform (see 2.12. Divergent Control Flow (page 41)). 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` instruction 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` instructions on the same `fbarrier` and the work-item that executes the `releasefbar`. This requirement can be satisfied by using the `barrier` instruction, or the `waitfbar` instruction (on another `fbarrier`) between the `fbarrier` instructions that use it and the `releasefbar`.

`ldf`

Places the address of an `fbarrier` into the destination `dest`. The address has work-group persistence (see [2.8.4. Memory Segment Access Rules \(page 36\)](#)) and the value can only be used in work-items that belong to the same work-group as the work-item that executed the `ldf` instruction. The compound type `dest` is always `u32` regardless of the machine model (see [2.9. Small and Large Machine Models \(page 39\)](#)). The value returned can be used with `fbarrier` instructions to specify which `fbarrier` they are to operate on.

9.2.4 Additional Information About Fbarrier Instructions

Additional information about the use of `fbarrier` instructions:

- `Fbarrier` instructions are allowed in divergent code. In fact, this is a primary reason to use `fbarriers` rather than the `barrier` instruction, which can only be used in work-group uniform code. However, `fbarrier` usage must be wavefront uniform.
- The `fbarrier` instruction that arrives at an `fbarrier` does not need to be the same instruction in each wavefront. The instruction simply needs to reference the same `fbarrier`.
- The `fbarrier` instructions 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 instructions once per work-group `fbarrier`, and then reuse the same `fbarrier` by using `joinfbar` and `leavefbar`. A `barrier` instruction, or `waitfbar` (to another `fbarrier`) instruction, can be used between the `leavefbar` and `joinfbar` instructions to avoid race conditions between the `fbarrier` instructions that use the `fbarrier` for different purposes.
- For more information on how `waitfbar` and `arrivefbar` interact with the memory operations performed by the work-items that are members of the associated `fbarrier`, see [9.3. Execution Barrier \(page 238\)](#).

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` instructions. 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` instructions and `waitfbar`, `arrivefbar`, and `leavefbar`, instructions 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` instruction, or the `waitfbar` instruction (on another fbarrier), between the `joinfbar` and `waitfbar`, `arrivefbar`, and `leavefbar` instructions. 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) instructions can also ensure the race condition free requirement.

- Similarly, there cannot be a race condition between an `arrivefbar` instruction and other fbarrier instructions that could result in the same work-item performing more than one fbarrier instruction 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) instruction after the `arrivefbar` instruction.

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_system;
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_system;
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_system;
18:     waitfbar %fb1; // fb1 only has work-items for which
                     // cond1 && cond2 is true
19:     memfence_scacq_system;
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 wavefront 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 produces and consumers are in the same wavefront
// so that the fbarrier instructions are wavefront 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_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_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 instruction is used to synchronize the execution of the work-items that participate in an associated execution barrier instance:

- For the `barrier` instruction 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` instruction.
- For the `wavebarrier` instruction the participating work-items are those that are members of the same wavefront and each wavefront has a distinct execution barrier instance per `wavebarrier` instruction.
- For the `waitfbar` and `arrivefbar` instructions 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 instruction that specifies the execution barrier instance.

The barrier instructions interact with the memory model (see [6.2. Memory Model \(page 169\)](#)) as if they use read-modify-write relaxed atomic memory instructions on a location associated with their barrier instance as described by the following pseudo code:

```

atomic_add_group_rlx_wg(barrier_instance.arrived_count, 1);
while (atomic_ld_group_rlx_wg(barrier_instance.arrived_count) !=
      barrier_instance.participant_count) sleep;
atomic_sub_group_rlx_wg(barrier_instance.arrived_count, 1);
if (operation != arrivefbar)
    while (atomic_ld_group_rlx_wg(barrier_instance.arrived_count) != 0) sleep;

```

However, it is permitted to implement barrier instructions in any manner, provided they interact with the memory model in the same way.

A consequence is that:

- A barrier instruction does not prevent memory instructions performed by the same thread being reordered with the barrier instruction.
- A barrier instruction does not ensure that the memory instructions that precede it become visible to the memory instructions that succeed it for the participating work-items after the barrier instance has been satisfied.

However, by using a release memory fence before a barrier instruction, and an acquire memory fence after a barrier instruction, for the desired memory scope, the following can be achieved:

- Prevent reordering of memory instructions across a barrier instruction.
- Ensure visibility of the memory instructions performed by participating work-items before the barrier instruction, to the memory instructions performed by participating work-items after the barrier instruction.

This is achieved as a consequence of the memory model rules that define how memory fences interact with the conceptual relaxed atomic instructions of the barrier to establish happens-before relations. Note that the read-modify-write loops ensure all participating work-items have accessed the same location on entry to the barrier, and also accessed the same location again on exit (except for `arrivefbar`) after the barrier instance has been satisfied. Therefore, an acquire memory fence executed by a work-item after the barrier instance has been satisfied, will synchronize-with each release memory fence executed by the participating work-items before the barrier instruction, and so make the memory instructions performed by those work-items visible.

See [7.1.10. Image Memory Model \(page 218\)](#) for additional rules related to `imagefence`.

The `clock` (see [11.4. Miscellaneous Instructions \(page 264\)](#)) and `signal` (see [6.8. Notification \(signal\) Instructions \(page 187\)](#)) instructions are defined to behave as if an atomic memory instruction, and so a memory fence will also control their reordering across a barrier instruction.

The `cross-lane` (see [9.4. Cross-Lane Instructions \(next page\)](#)), `cleardetectexcept`, `getdetectexcept`, and `setdetectexcept` (see [11.2. Exception Instructions \(page 260\)](#)) instructions are required to be execution uniform (see [2.12. Divergent Control Flow \(page 41\)](#)). Therefore, the communicating work-items will always execute together and so it is not observable if an implementation reorders them in either direction across a barrier instruction.

Instructions not otherwise specified above do not involve communication between work-items or other agents. Therefore, they can be moved (by the implementation) across a barrier instruction in either direction, since their execution order is not detectable from other work-items or other agents.

A barrier instruction 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 instruction 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 \(page 41\)](#).

A barrier instruction 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 instruction may be less than or equal to the wavefront size either because the instruction is `wavebarrier` or is `barrier` when the work-group size is less than or equal to the wavefront size. In such cases all participating work-items will be members of the same wavefront, and an implementation is free to optimize the code generated for the barrier when the gang-scheduled execution of work-items in wavefronts will ensure execution synchronization. However, even if such an optimization is performed, any memory fences that come before or after the original position of the barrier instruction must continue to behave in the same way.

Note it is undefined to omit a barrier instruction 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 wavefront size, a barrier instruction 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 \(page 278\)](#)) can be used to specify the work-group size. This can allow the implementation to optimize the barrier instruction when the size is less than or equal to the implementation's wavefront size.

9.4 Cross-Lane Instructions

These instructions perform work across lanes in a wavefront. These instructions apply only to active work-items within a wavefront (see [2.5. Active Work-Groups and Active Work-Items \(page 28\)](#)).

9.4.1 Syntax

Table 9–3 Syntax for Cross-Lane Instructions

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>activelanepermute_width_bLength</code>	<code>dest, src, laneId, identity, useIdentity</code>

Explanation of Modifier
<i>width</i> : Optional: <code>width(n)</code> , <code>width(WAVESIZE)</code> , or <code>width(all)</code> . 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 <code>width(1)</code> , indicating each lane of the wavefront can be independently active or inactive. See 2.12. Divergent Control Flow (page 41) .
<i>Length</i> : 1, 32, 64, 128.

Explanation of Operands (see 4.16. Operands (page 104))
<i>dest, dest0, dest1, dest2, dest3</i> : Destination register.
<i>src, laneId, identity, useIdentity</i> : Sources. Can be a register or immediate value.

Exceptions (see Chapter 12. Exceptions (page 269))
No exceptions are allowed.

For BRIG syntax, see [18.7.5. BRIG Syntax for Parallel Synchronization and Communication Instructions \(page 356\)](#).

9.4.2 Description

activelaneccount

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 instruction 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.

activelanepermute

If the lane *laneId* modulo WAVESIZE (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* modulo WAVESIZE is transferred to *dest*. Note that lanes not part of a work-group (due to partial wavefronts) are treated as inactive.

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 *activelanepermute* instruction. 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_b1 $s1, $c0;
activelaneid_u32 $s1;
activelaneid_width(WAVESIZE)_u32 $s1;
activelanemask_v4_b64_b1 ($d0, $d1, $d2, $d3), $c0;
activelanepermute_b32 $s1, $s2, $s2, $s3, $c1;
activelanepermute_b64 $d1, $d2, 0, 0, 0;
activelanepermute_width(all)_b128 $q1, $q2, $s2, $q3, $c1;

```

CHAPTER 10.

Function Instructions

This chapter describes how to use functions in HSAIL and the related instructions.

10.1 Functions in HSAIL

Like other programming languages, HSAIL provides support for user functions. A call instruction 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` instruction, control is transferred back to the instruction immediately after the call instruction.

In order that HSAIL can execute efficiently on a wide range of compute units, an abstract method is used for passing arguments, with the finalizer determining what to do. This is necessary because, on a GPU, stacks are not a good use of resources, especially if each work-item has its own stack. If an application is simultaneously running, for example, 30,000 work-items, then the stack-per-work-item is very limited. Having one return address per wavefront (not one address per work-item) is desirable.

Implementations should map the abstractions into appropriate hardware.

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` instruction transfers control to `foo`, implicitly saving the return address. Execution of the `ret` instruction within `foo` transfers control to the instruction 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 instruction 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. Instructions in the function 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 \(page 62\)](#) and [4.3.3. Function \(page 58\)](#).

A function specifies a list of zero or more output formal arguments and a list of zero or more input formal arguments. A call instruction 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 instruction.

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.
- Instructions to assign values to actual arguments used as input formal arguments of the function being called.
- Exactly one call instruction that uses those actual arguments.
- Instructions 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 instructions 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 \(page 78\)](#).

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 an error to use a `ret` instruction in an arg block.

It is not valid to use an `alloca` instruction 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 instruction in the arg block. Each actual argument must appear exactly once as either an input actual argument or output actual argument of the call instruction. It is an error if an actual argument does not appear as one of the call instructions 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 instruction 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 dimension declarations, and alignment. The array dimension declaration matches if both are not arrays (have no array dimension) or both are arrays and 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 \(page 248\)](#).

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 \(page 248\)](#)).

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, the results are undefined if they are accessed by any instruction other than a `st` instruction that is post-dominated (see [2.12.3. \(Post-\)Dominator and Immediate \(Post-\)Dominator \(page 45\)](#)) by the call instruction.

For the actual argument that corresponds to the output formal argument, the results are undefined if it is accessed by any instruction other than a `ld` instruction that is dominated by the call instruction.

It is undefined if the single call instruction 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 instruction within the arg block, or loop within the arg block to execute the call instruction 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 an error if they are accessed by any instruction other than an `ld` instruction.
- For the output formal argument, it is an error if it is accessed by any instruction other than an `st` instruction.

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 instruction 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 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 instruction can use the function as a target. See [10.6. Direct Call \(call\) Instruction \(page 250\)](#), [10.7. Switch Call \(scall\) Instruction \(page 251\)](#) and [10.8. Indirect Call \(icall\) Instruction \(page 252\)](#).

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 \(page 58\)](#).

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 \(page 58\)](#).

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` instruction. Syntactically, a signature is much like a function. See [4.3.4. Signature \(page 60\)](#).

In the following example, assume that `$d2` in each work-item contains an indirect function code handle:

```

signature &fun_t(arg_u32) (arg_u32, arg_u32);
function &caller1() ()
{
    // ...
    {
        arg_u32 %in1;
        arg_u32 %in2;
        arg_u32 %out;
        // ...
        icall_u64 $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.

For more information, see [10.8. Indirect Call \(icall\) Instruction \(page 252\)](#).

10.4 Variadic Functions

A variadic function is a function that accepts a variable number of arguments.

In HSA IL, 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 instruction 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
    chr_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 \(page 69\)](#).

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) Instruction

The `call` instruction transfers control to a specific function.

10.6.1 Syntax

Table 10–1 Syntax for direct call Instruction

Opcode and Modifiers	Operands
<code>call</code>	<i>function</i> (<i>outputArgs</i>) (<i>inputArgs</i>)

Explanation of Operands (see 4.16. Operands (page 104))	
<i>function</i> : Must be the identifier of a function (either non-indirect or indirect). The function output and input formal arguments must match the <i>outputArgs</i> and <i>inputArgs</i> specified.	
<i>outputArgs</i> : List of zero or one call argument.	
<i>inputArgs</i> : List of zero or more comma-separated call arguments.	

Exceptions (see Chapter 12. Exceptions (page 269))	
No exceptions are allowed.	

For BRIG syntax, see [18.7.6. BRIG Syntax for Function Instructions \(page 357\)](#).

Description

A direct call instruction 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` instruction 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. The exception is that global and readonly segment variables may be declared only, in which case the HSA executable must be used to provide the definition, such as to a host application variable. See [4.2. Program, Code Object, and Executable \(page 48\)](#).

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 \(page 244\)](#).

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 instruction. 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_f32 %a;
        arg_u32 %r;
        st_arg_f32 2.0f, [%a];
        // call &foo
        call &foo(%r) (%a);
    }
}
```

```

ld_arg_u32 $s1, [%r];
}
st_arg_u32 $s1, [%res];
};

```

10.7 Switch Call (scall) Instruction

The `scall` instruction uses an integer index to select the specific function to which control is transferred.

10.7.1 Syntax

Table 10–2 Syntax for switch call Instruction

Opcode and Modifiers	Operands
<code>scall_width_uLength</code>	<code>src (outputArgs) (inputArgs) [functionList]</code>
Explanation of Modifier	
<i>width</i> : Optional: <code>width(n)</code> , <code>width(WAVESIZE)</code> , or <code>width(all)</code> . 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 <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 Instructions (page 44) .	
<i>Length</i> : 32, 64.	
Explanation of Operands (see 4.16. Operands (page 104))	
<i>src</i> : Source. Can be a register or immediate value.	
<i>outputArgs</i> : List of zero or one call argument.	
<i>inputArgs</i> : List of zero or more comma-separated call arguments.	
<i>functionList</i> : 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 <i>outputArgs</i> and <i>inputArgs</i> specified.	
Exceptions (see Chapter 12. Exceptions (page 269))	
No exceptions are allowed.	

For BRIG syntax, see [18.7.6. BRIG Syntax for Function Instructions \(page 357\)](#).

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. The results are 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` instruction 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. The exception is that global and readonly segment variables may be declared only, in which case the HSA executable must be used to provide the definition, such as to a host application variable. See [4.2. Program, Code Object, and Executable \(page 48\)](#).

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 \(page 41\)](#).

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 \(page 244\)](#).

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 instruction. 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_arg_u32 $s1, [%arg1];
    {
        arg_f32 %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_u32 $s1, [%r];
    }
    st_arg_u32 $s1, [%res];
};
```

10.8 Indirect Call (icall) Instruction

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 code handles 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` instruction starting at the indirect function 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 10–3 Syntax for indirect call Instruction

Opcode and Modifiers	Operands
<code>icall_width_uLength</code>	<code>src (outputArgs) (inputArgs) signature</code>

Explanation of Modifier
<i>width</i> : 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 width 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 Instructions (page 44) .
<i>Length</i> : 32, 64. Must match the size of an indirect function code handle (see Table 2–3 (page 40)).

Explanation of Operands (see 4.16. Operands (page 104))
<i>outputArgs</i> : List of zero or one call argument.
<i>inputArgs</i> : List of zero or more comma-separated call arguments.
<i>src</i> : A register.
<i>signature</i> : Global identifier of a signature. The signature output and input formal arguments must match the <i>outputArgs</i> and <i>inputArgs</i> specified.

Exceptions (see Chapter 12. Exceptions (page 269))
No exceptions are allowed.

For BRIG syntax, see [18.7.6. BRIG Syntax for Function Instructions \(page 357\)](#).

10.8.2 Description

The `icall` instruction is not supported by the Base profile. See [16.2.1. Base Profile Requirements \(page 289\)](#).

An indirect call transfers control to the indirect function that corresponds to the indirect function code handle in *src*. The indirect function being called has formal arguments matching those of *signature*.

A host CPU agent can use an HSA runtime query to obtain an indirect function code handle. That code handle can then be passed into a kernel as a kernel argument or through global segment memory.

The results are undefined unless all the following are true:

- *src* is a valid indirect function code handle obtained from a code object that:
 - is currently loaded in the same kernel agent as the currently executing kernel;
 - was loaded before the currently executing kernel was launched.
- *src* refers to an indirect function with formal input and output arguments that match *signature*.
- *src* refers to an indirect function that was finalized with the same call convention as the currently executing kernel.

See [4.2. Program, Code Object, and Executable \(page 48\)](#).

At the time of finalizing, the actual indirect function that an `icall` will call at runtime does not have to be finalized.

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 \(page 41\)](#).

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 \(page 244\)](#).

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;

// 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_f32 %a;
        arg_u32 %r;
        st_arg_f32 2.0f, [%a];
        // $d1 must contain an indirect function code handle 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)_u64 $d1(%r)(%a) &bar_or_foo_t;
        ld_arg_u32 $s1, [%r];
        ld_kernarg_u64 $d1, [%res];
        st_global_u32 $s1, [$d1];
    }
};
```

10.9 Return (ret) Instruction

The return (`ret`) instruction returns from a function back to the caller's environment. `ret` can also be used to exit a kernel.

If there is no `ret` instruction before the exit of the kernel's or function's code block, the finalizer will act as if a `ret` instruction was present at the end of the code block.

10.9.1 Syntax

Table 10–4 Syntax for `ret` Instruction

Opcode
<code>ret</code>
Exceptions (see Chapter 12. Exceptions (page 269))
No exceptions are allowed.

For BRIG syntax, see [18.7.6. BRIG Syntax for Function Instructions \(page 357\)](#).

10.9.2 Description

Within a function, a `ret` instruction 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` instruction 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*) Instruction

The allocate memory (`alloca`) instruction 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 10–5 Syntax for Allocate Memory (*alloca*) Instruction

Opcode	Operands
<code>alloca_align(n)_u32</code>	<code>dest, src</code>
Explanation of Modifiers	
<code>align(n)</code> : 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 (page 104))	
<code>dest</code> : Destination. Must be a 32-bit register.	
<code>src</code> : Source. Can be a 32-bit register or immediate value. The value of <code>src</code> is the minimum amount of space (in bytes) requested.	

Exceptions (see [Chapter 12. Exceptions \(page 269\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.6. BRIG Syntax for Function Instructions \(page 357\)](#).

10.10.2 Description

The `alloca` instruction 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` instructions.

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 \(page 295\)](#).

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` instruction in an argument scope. See [10.2. Function Call Argument Passing \(page 244\)](#).

Example

```
alloca_u32 $s1, 24;
alloca_align(8)_u32 $s1, 24;
```

CHAPTER 11.

Special Instructions

This chapter describes special instructions that can be used to perform various miscellaneous actions and queries.

11.1 Kernel Dispatch Packet Instructions

The kernel dispatch packet instructions 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 kernel dispatch packet instructions in alphabetical order.

Table 11–1 Syntax for Kernel Dispatch Packet Instructions

Opcodes and Modifier	Operands
currentworkgroupsize_u32	<i>dest</i> , <i>dimNumber</i>
currentworkitemflatid_u32	<i>dest</i>
dim_u32	<i>dest</i>
gridgroups_u32	<i>dest</i> , <i>dimNumber</i>
gridsize_uLength	<i>dest</i> , <i>dimNumber</i>
packetcompletionSIG_signalType	<i>dest</i>
packetid_u64	<i>dest</i>
workgroupid_u32	<i>dest</i> , <i>dimNumber</i>
workgroupsize_u32	<i>dest</i> , <i>dimNumber</i>
workitemabsid_uLength	<i>dest</i> , <i>dimNumber</i>
workitemflatabsid_uLength	<i>dest</i>
workitemflatid_u32	<i>dest</i>
workitemid_u32	<i>dest</i> , <i>dimNumber</i>

Explanation of Modifiers

signalType: Must be `sig32` for small machine model and `sig64` for large machine model. See [Table 4–4 \(page 101\)](#) and [2.9. Small and Large Machine Models \(page 39\)](#).

Length: 32, 64.

Explanation of Operands (see [4.16. Operands \(page 104\)](#))

dest: Destination. For `packetcompletionSIG` and `packetid` must be a `d` register; otherwise must be an `s` register. See [Table 2–3 \(page 40\)](#).

dimNumber: Source that selects the dimension (X, Y, or Z). 0, 1, and 2 are used for X, Y, and Z, respectively. Must be a constant value of data type `u32`. `WAVESIZE` is not allowed.

Exceptions (see [Chapter 12. Exceptions \(page 269\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.7.1. BRIG Syntax for Kernel Dispatch Packet Instructions \(page 358\)](#).

11.1.2 Description

`currentworkgroupsize`

Accesses the work-group size that the currently executing work-item belongs to for the *dimNumber* dimension and stores the result in the destination *dest*. See [2.2. Work-Groups \(page 25\)](#)

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

If the kernel was dispatched with fewer dimensions than *dimNumber*, then `currentworkgroupsize` returns 1 for the unused dimensions.

`currentworkitemflatid`

Accesses the flattened form of the work-item identifier (ID) within the current work-group and stores the result in the destination *dest*. See [2.3.2. Work-Item Flattened ID and Current Work-Item Flattened ID \(page 27\)](#).

`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 \(page 23\)](#).

`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*. Can either be returned as a `u32` or `u64`. If `u32`, then the lower 32 bits of the size are returned.

If the grid was launched with fewer dimensions than *dimNumber*, then `gridsize` stores 1 in destination *dest*.

`packetcompletion sig`

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 \(signal\) Instructions \(page 187\)](#)). See *HSA Platform System Architecture Specification Version 1.0* section 2.8 Requirement: 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 kernel dispatch and stores the result in the destination *dest*. See *HSA Platform System Architecture Specification Version 1.0* section 2.8 *Requirement: User Mode Queuing*.

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. See [2.2.1. Work-Group ID \(page 25\)](#).

This instruction 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 and stores the result in the destination *dest*. See [2.2. Work-Groups \(page 25\)](#).

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` instruction 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. See [2.3.3. Work-Item Absolute ID \(page 27\)](#).

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. See [2.3.4. Work-Item Flattened Absolute ID \(page 27\)](#).

`workitemflatid`

Accesses the flattened form of the work-item identifier (ID) within the work-group and stores the result in the destination *dest*. See [2.3.2. Work-Item Flattened ID and Current Work-Item Flattened ID \(page 27\)](#).

workitemid

Accesses the work-item identifier (ID) within the work-group and stores the result for the *dimNumber* dimension in the destination *dest*. See 2.3.1. Work-Item ID (page 26).

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
currentworkitemflatid_u32 $s1; // access the current work-item flat ID
dim_u32 $s3;                   // dispatch dimensions
gridgroups_u32 $s2, 2;         // access the number of work-groups in the
                                // grid Z dimension
gridsize_u32 $s2, 1;           // access the number of work-items in the
                                // grid Y dimension
gridsize_u64 $d2, 2;           // access the number of work-items in the
                                // grid Z dimension
packetcompletion_sig_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 Instructions

The exception instructions can be used to determine what exceptions have been generated.

11.2.1 Syntax

The table below shows the syntax for the exception instructions in alphabetical order.

Table 11-2 Syntax for Exception Instructions

Opcodes and Modifier	Operands
cleardetectexcept_u32	<i>exceptionsNumber</i>
getdetectexcept_u32	<i>dest</i>
setdetectexcept_u32	<i>exceptionsNumber</i>

Explanation of Operands (see 4.16. Operands (page 104))

dest: Destination. Must be an *s* register. See Table 2-3 (page 40).

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 a constant value of data type *u32*. *WAVESIZE* is not allowed.

Exceptions (see Chapter 12. Exceptions (page 269))

No exceptions are allowed.

For BRIG syntax, see [18.7.7.2. BRIG Syntax for Exception Instructions \(page 358\)](#).

11.2.2 Description

`cleardetectexcept`

Clears DETECT exception flags specified in *exceptionsNumber* for the wavefront containing the work-item. The result is undefined if the instruction is not wavefront execution uniform (see [2.12. Divergent Control Flow \(page 41\)](#)), and might lead to deadlock.

`getdetectexcept`

Returns the current value of DETECT exception flags, which is a summarization for all work-items in the wavefront 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 wavefront containing the current work-item, as modified by any preceding `cleardetectexcept` or `cleardetectexcept` instructions executed by any work-item in the wavefront 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 instruction is not wavefront execution uniform (see [2.12. Divergent Control Flow \(page 41\)](#)), and might lead to deadlock.

`setdetectexcept`

Sets DETECT exception flags specified in *exceptionsNumber* for the wavefront containing the current work-item. The result is undefined if the instruction is not wavefront execution uniform (see [2.12. Divergent Control Flow \(page 41\)](#)), and might lead to deadlock.

11.2.3 Additional Information

DETECT exception processing operates on the five exceptions specified in [12.2. Hardware Exceptions \(page 269\)](#).

DETECT exception processing is performed independently for each wavefront. Each wavefront conceptually maintains a 5-bit *exception_detected* field which is initialized to 0 before the wavefront 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` instructions in any way it chooses, including by using dedicated hardware.

If any of the five exceptions occurs in any work-item of the wavefront, the bit corresponding to the exception is conceptually set in the *exception_detected* field.

The `cleardetectexcept`, `setdetectexcept`, and `getdetectexcept` instructions conceptually operate on the *exception_detected* field, and their execution must be wavefront uniform. If they are used inside of wavefront divergent control flow, the result is undefined, and might lead to deadlock. These instructions can be used in a loop, provided the loop introduces no wavefront divergent control flow. This requires that all work-items in the wavefront execute the loop the same number of iterations. See [2.12. Divergent Control Flow \(page 41\)](#).

The wavefront *exception_detected* field is not implicitly saved when the work-items of the wavefront 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` instruction 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 wavefronts 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 (see [13.4. Control Directives for Low-Level Performance Tuning \(page 278\)](#)). 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 \(page 296\)](#).

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 Instructions

The User Mode Queue instructions can be used to enqueue work to be executed by other agents. See *HSA Platform System Architecture Specification Version 1.0* section 2.8 *Requirement: User Mode Queuing*.

11.3.1 Syntax

The table below shows the syntax for the User Mode Queue instructions in alphabetical order.

Table 11–3 Syntax for Exception Instructions

Opcodes and Modifier	Operands
<code>addqueuewriteindex_segment_order_u64</code>	<i>dest, address, src</i>
<code>casqueuewriteindex_segment_order_u64</code>	<i>dest, address, src0, src1</i>
<code>ldqueuereadindex_segment_order_u64</code>	<i>dest, address</i>
<code>ldqueuewriteindex_segment_order_u64</code>	<i>dest, address</i>
<code>stqueuereadindex_segment_order_u64</code>	<i>address, src</i>
<code>stqueuewriteindex_segment_order_u64</code>	<i>address, src</i>

Explanation of Modifiers
<i>segment</i> : Optional segment. If omitted, flat is used. Only flat and global is allowed. See 2.8. Segments (page 31) .
<i>order</i> : Memory order used to specify synchronization. See 6.2.1. Memory Order (page 169) .
<i>Length</i> : 32, 64. Must match the address size for the global segment (see Table 2-3 (page 40)).

Explanation of Operands (see 4.16. Operands (page 104))
<i>dest</i> : Destination. Must be a d register.
<i>src</i> , <i>src0</i> , <i>src1</i> : Sources. Can be a register or immediate value.
<i>address</i> : Address expression for an address in the specified segment for a User Mode Queue created by the HSA runtime (see 4.18. Address Expressions (page 106)).

Exceptions (see Chapter 12. Exceptions (page 269))
No exceptions are allowed.

For BRIG syntax, see [18.7.7.3. BRIG Syntax for User Mode Queue Instructions \(page 358\)](#).

11.3.2 Description

`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 `rlx` (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 `rlx` (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.

`ldqueuereadindex`

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 `rlx` (relaxed) or `scacq` (sequentially consistent acquire). Can be used in conjunction with `ldqueuewriteindex` to determine how many User Mode Queue slots are available.

ldqueuewriteindex

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 *rlx* (relaxed) or *scacq* (sequentially consistent acquire). Can be used in conjunction with *ldqueuereadindex* to determine how many User Mode Queue slots are available.

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 a kernel agent 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 a kernel agent 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

```
ldqueuewriteindex_global_rlx_u64 $d3, [$d2];    // load queue write index
add_u64 $d4, $d3, 1;
casqueuewriteindex_global_scar_u64 $d1, [$d2], $d3, $d4;
                                                // compare-and-swap queue write index
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 Instructions

The miscellaneous instructions include various query and special operations.

11.4.1 Syntax

The table below shows the syntax for the miscellaneous instructions in alphabetical order.

Table 11–4 Syntax for Miscellaneous Instructions

Opcodes and Modifier	Operands
<code>clock_u64</code>	<i>dest</i>
<code>cuid_u32</code>	<i>dest</i>
<code>debugtrap_u32</code>	<i>src</i>
<code>groupbaseptr_u32</code>	<i>dest</i>
<code>kernargbaseptr_uLength</code>	<i>dest</i>
<code>laneid_u32</code>	<i>dest</i>
<code>maxcuid_u32</code>	<i>dest</i>
<code>maxwaveid_u32</code>	<i>dest</i>
<code>nop</code>	
<code>nullptr_segment_uLength</code>	<i>dest</i>
<code>waveid_u32</code>	<i>dest</i>

Explanation of Modifiers

Length: 32, 64. Must match the address size for the associated segment: for `nullptr` it is the segment specified; and for `kernargbaseptr` it is the kernarg segment (see [Table 2–3 \(page 40\)](#)).

segment: Optional segment. If omitted, flat is used. Can be flat, group, private, and kernarg. See [2.8. Segments \(page 31\)](#).

Explanation of Operands (see [4.16. Operands \(page 104\)](#))

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 2–3 \(page 40\)](#).

src: Source. Can be a register or immediate value.

Exceptions (see [Chapter 12. Exceptions \(page 269\)](#))

No exceptions are allowed.

For BRIG syntax, see [18.7.7.4. BRIG Syntax for Miscellaneous Instructions \(page 359\)](#).

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` instruction is treated as if it is a read-modify-write relaxed atomic memory instruction (see [6.2. Memory Model \(page 169\)](#)). This ensures that the `clock` instruction 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` instruction cannot be moved (by the implementation) before a preceding acquire memory instruction in the same work-item.

- A `clock` instruction cannot be moved (by the implementation) after a following release memory instruction in the same work-item.
- The order of two `clock` instructions cannot be changed (by the implementation).
- Multiple `clock` instructions cannot be combined (by the implementation) to a single instruction, 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`. The result is 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`

Halts execution of the wavefront executing the instruction and generates a debug exception. See [12.4. Debug Exceptions \(page 272\)](#).

If the optional HSA runtime debug interface is not present, or present and not active, the User Mode Queue executing the kernel dispatch will also be put into an error state. This will terminate all kernel dispatches executing on that queue. See [12.5.1. HSA Runtime Debug Interface Not Active \(page 272\)](#).

If the optional HSA runtime debug interface is present and is active, the behavior is controlled by the debug interface. See [12.5.2. HSA Runtime Debug Interface Active \(page 272\)](#). The value of the source operand `src` is accessible using the debug interface and could be used to identify the reason for the trap. The meaning of the value is user defined. For example, the values could be defined by a high level language implementation and used by that implementation's compiler, runtime, and debugger.

`groupbaseptr`

Returns the group segment address of the base of the group segment for the work-group of the work-item executing the instruction, 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 \(page 69\)](#)), 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 instruction. See [4.20. Dynamic Group Memory Allocation \(page 112\)](#).

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 \(page 69\)](#)) with *n* larger than 16, then the returned address will have alignment at least the maximum *n* specified. See [4.21. Kernarg Segment \(page 114\)](#).

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 kernel agent and stores the result in the destination *dest*. For example, if a kernel agent 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 a kernel agent 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 flat address null pointer value is the same for all agents, including host CPU agents, and is dependent on the host operating system.

The null pointer value used for the global and readonly segment is the same as that used for a flat address.

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` instruction.

The null pointer value for the group, private, and kernarg segments is agent dependent and different agents may use different values.

The implementation is required to ensure no segment variable is allocated, and no memory segment allocator will return an address, with the null pointer value used by the specified segment.

An HSA runtime query is available to obtain the null pointer value for the group, private, and kernarg segments for each agent. The host operating system provides the value used for the null pointer value for a flat address

`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
cuid_u32 $s0;            // access the compute unit id within the kernel agent
debugtrap_u32 $s1;       // halt execution and transfer control to debugger if debug
                        // interface is active
groupbaseptr_u32 $s2;    // 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 kernel agent
maxwaveid_u32 $s4;       // access the maximum number of wavefronts that can be executing
                        // at the same time by the kernel agent
nop;                     // no operation
nullptr_group_u32 $s0;   // null pointer value for group segment
nullptr_u64 $d1;         // null pointer value for a flat address or global segment
waveid_u32 $s3;          // access the wavefront ID within the kernel agent
```

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 \(below\)](#).
- Software-triggered exceptions corresponding to higher-level catch and throw operations. HSAIL provides no special instructions for handling software exceptions. They can be implemented in terms of the HSAIL branch instructions.
- Debug exceptions generated by `debugtrap` or as a consequence of actions performed by the optional HSA runtime debug interface. See [12.4. Debug Exceptions \(page 272\)](#).

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 \(page 271\)](#)). 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 instructions, 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 \(page 289\)](#)), 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 \(page 272\)](#). 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 instruction 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. Floating-Point Rounding \(page 109\)](#).

- Underflow

A non-zero tiny floating-point value is computed and either:

- the `ftz` modifier was specified,
- or the `ftz` modifier was not specified and the value cannot be represented exactly.

See [4.19.2. Floating-Point Rounding \(page 109\)](#).

- Division by zero

A finite non-zero floating-point value is divided by zero.

It is implementation defined whether integer `div` or `rem` instructions with a divisor of zero will generate a divide by zero exception.

See [4.19.2. Floating-Point Rounding \(page 109\)](#).

- Invalid operation

Instructions are performed on values for which the results are not defined. These are:

- Operations on signaling NaN floating-point values.
- Signaling comparisons: comparisons on quiet NaN floating point values.
- Multiplication: `mul(0.0, infinity)` or `mul(infinity, 0.0)`.
- Fused multiply add: `fma(0.0, infinity, c)` or `fma(infinity, 0.0, c)` unless `c` is a quiet NaN, in which case it is implementation defined if an exception is generated.
- Addition, subtraction, or fused multiply add: magnitude subtraction of infinities, such as: `add(positive infinity, negative infinity)`, `sub(positive infinity, positive infinity)`.
- Division: `div(0.0, 0.0)` or `div(infinity, infinity)`.
- Square root: `sqrt(negative)`.
- Conversion: A `cvt` with a floating-point source type, an integer destination type, and a nonsaturating rounding mode, when the source value is a NaN, infinity, or the rounded value, after any flush to zero, cannot be represented precisely in the integer type of the destination.

- Inexact

A computed floating-point value is not represented exactly in the destination. This can occur:

- Due to rounding. See [4.19.2. Floating-Point Rounding \(page 109\)](#).
- In addition, it is implementation defined whether instructions with the `ftz` modifier that cause a value to be flushed to zero generate the inexact exception. See [4.19.3. Flush to Zero \(ftz\) \(page 110\)](#).

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` instruction 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 \(page 269\)](#). Whether either exception policy is supported by a kernel agent depends on the profile specified (see [16.2. Profile-Specific Requirements \(page 289\)](#)).

- 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` instructions 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 Instructions \(page 260\)](#).

- 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 \(next page\)](#).

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 whether 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 \(page 296\)](#).

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 instruction 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 instructions 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 `enablebreakexceptions` control directive in the kernel, or any functions it calls directly or indirectly, being finalized.

12.4 Debug Exceptions

Debug exceptions include those generated by the `debugtrap` instruction (see [Chapter 11. Special Instructions \(page 257\)](#)), and those the optional HSA runtime debug interface may cause to be generated if it is active (for example, due to inserted breakpoints, single stepping machine instructions, or profile counter events).

When a debug exception is generated it always signals the exception. See [12.5. Handling Signaled Exceptions \(below\)](#). If the optional HSA runtime debug interface is active and causes execution to be 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 Signaled Exceptions

If an exception is signaled, the behavior depends on if the HSA runtime debug interface is active.

12.5.1 HSA Runtime Debug Interface Not Active

If the HSA runtime debug interface is not active, a wavefront that executes an instruction that signals an exception must halt execution of the wavefront. In reasonable time, the kernel agent 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 Debug Interface Active

The *HSA Runtime Programmer's Reference Manual Version 1.0* does not define a standard HSA runtime debug interface. However, the HSA runtime may optionally provide an implementation dependent debug interface.

As guidance, the following section provides an example of the functionality that a debug interface may provide. A future version of the HSA runtime may define a standard debug interface that may differ from that described below.

12.5.2.1 Sample Debug Interface

If the HSA runtime debug interface is active, a signaled exception causes the wavefront that executed the instruction to be halted and information about the exception is available through the debug interface. The debugger interface may optionally have the capability 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 instruction 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 debugger interface may optionally provide 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 that have been halted, 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 debugger interface may optionally have the ability 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 debugger interface may optionally have the ability 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 module header but before the first HSAIL module statement (see [4.3. Module \(page 53\)](#)). 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 a kernel agent does not support an extension that is enabled in a module, then the finalizer for that kernel agent must report an error.

An HSA runtime operation can be used to query if a kernel agent 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 instructions defined in [Chapter 7. Image Instructions \(page 194\)](#) are allowed in the module in which it appears:

```
extension "IMAGE";
```

If the "IMAGE" extension directive is not present, then the following HSAIL instructions are not allowed in the module in which it appears:

- rdimage
- ldimage
- stimage
- queryimage
- querysampler
- imagefence

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 formal arguments; and cannot be used with the `ld`, `st`, and `mov` instructions, including passing function arguments.

13.1.3 How to Set Up Finalizer Extensions

HSAIL opcodes are 16 bits in the BRIG binary format. Values 0 through 0x7FFF are reserved for HSA use, but values 0x8000 to 0xFFFF are available for vendor defined extensions.

For example, assume that a particular vendor `xyz` has implemented an extension called `newext` that provides a `max3` instruction which returns the maximum value of three floating-point inputs. The vendor's finalizer could choose to number the opcode for this instruction 0x8000. The HSAIL code that uses the extension would be:

```
module &ext:1:0:$full:$large:$default;
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 $[1, 2^{32})$. `WAVESIZE` is not allowed. The first line of the file is numbered 1.

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 $[1, 2^{32})$. `WAVESIZE` is not allowed. The first column of a line is numbered 1. 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 13–1 `pragma` Syntax Diagram

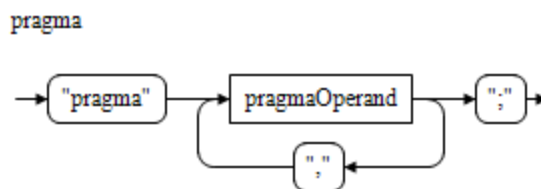
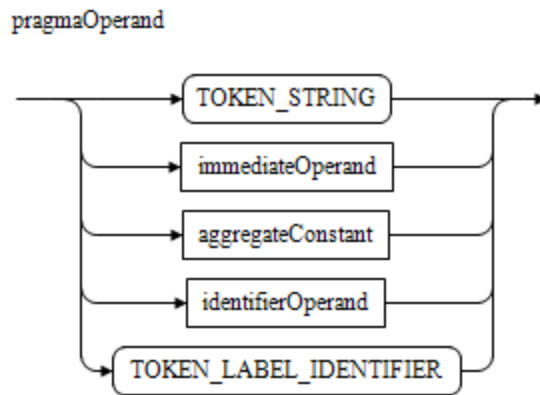


Figure 13–2 `pragmaOperand` Syntax Diagram

The `pragma` directive can appear anywhere in the HSAIL code that an annotation can appear (see [4.3.1. Annotations \(page 55\)](#)). 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. See [4.5. Strings \(page 76\)](#).
- An immediate operand. This includes integer constants that are treated as `u64`, double constants that are treated as `f64`, single constants that are treated as `f32`, half constants that are treated as `f16`, typed constants that are treated as the type of the constant, and `WAVESIZE` that is treated as `u64`. Note that all typed constants are allowed, including opaque types and arrays. See [4.8. Constants \(page 81\)](#).
- An aggregate constant. This is treated as an array of `b8` the byte size of the aggregate constant. See [4.8.4. Aggregate Constants \(page 91\)](#).
- An identifier. This includes the identifier of a variable, `fbarrier`, kernel, function, signature, module, register, and label. See [4.6. Identifiers \(page 77\)](#) and [4.7. Registers \(page 79\)](#).

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 \(page 78\)](#).

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. This includes determining what `pragma` operands are allowed.

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

HSA IL provides control directives to allow implementations to pass information to the finalizer. These directives are used for low-level performance tuning. See [Table 13–1 \(below\)](#).

Table 13–1 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>
requireddim	<i>nd</i>
requiredgridsize	<i>nx, ny, nz</i>
requiredworkgroupsize	<i>nx, ny, nz</i>
requirenopartialworkgroups	

Explanation of Arguments

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 a constant value of data type u32. WAVESIZE is not allowed. The bits corresponding to exceptions not supported by the kernel agent's profile must be 0 (see [16.2. Profile-Specific Requirements \(page 289\)](#)).

size: The number of bytes. Must be a constant value of data type u32. WAVESIZE is not allowed.

count: The number of work-items. Must be an immediate value, greater than 0, of data type u32 for maxflatworkgroupsize and u64 for maxflatgridsize. WAVESIZE is allowed.

nd: The number of dimensions. Must be a constant value, with the value 1, 2, or 3, of data type u32. WAVESIZE is not allowed.

nx, ny, nz: 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 u32 for requiredworkgroupsize and u64 for requiredgridsize. WAVESIZE is allowed.

See also [18.3.8. BrigControlDirective \(page 303\)](#).

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 HSA IL code block definition or statement (see [4.3.5. Code Block \(page 61\)](#)). 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 \(page 271\)](#). *exceptionsNumber* must be a constant value of data type `u32` (`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 set of exceptions enabled for the BREAK policy is the union of the sets specified by all the `enablebreakexceptions` control directives in the kernel or indirect function code block and the set of enable break exceptions 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 the results are 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 that are generated 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, are signaled, unless they contain `enablebreakexceptions` control directives or the finalizer was invoked specifying them in the enable break exceptions argument when they were finalized.

Whether the BREAK exception policy for the five exceptions is supported depends on the kernel agent and the profile specified (see [16.2. Profile-Specific Requirements \(page 289\)](#)). The finalizer is required to report an error if an exception that is not supported for the BREAK policy is enabled 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 \(page 112\)](#).

`enabledetectexceptions`

Specifies the set of exceptions that must be enabled for the DETECT policy. See [12.3. Hardware Exception Policies \(page 271\)](#). *exceptionsNumber* must be a constant value of data type `u32` (`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 set of exceptions enabled for the DETECT policy is the union of the sets of exceptions specified by all the `enabledetectexceptions` control directives in the kernel or indirect function code block and the set of enable detect exceptions 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 the results are 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 that are generated 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, update the conceptual *exception_detected* field (see 11.2.3. Additional Information (page 261)), unless they contain `enabledetectexceptions` control directives or the finalizer was invoked specifying them in the `enable detect exceptions` argument when they were finalized.

Whether the DETECT exception policy for the five exceptions is supported depends on the kernel agent and the profile specified (see 16.2. Profile-Specific Requirements (page 289)). The finalizer is required to report an error if an exception that is not supported for the DETECT policy is enabled 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 (page 112).

`maxdynamicgroupsize`

Specifies the maximum number of bytes of dynamic group memory (see 4.20. Dynamic Group Memory Allocation (page 112)) that will be allocated for a dispatch of the kernel. *size* must be a constant value of data type `u32`, with a value greater than or equal to 0 (`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` instructions. 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 (`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` instructions if the absolute grid size is less than $2^{24}-1$, because faster `mul24` instructions 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 (`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` instruction if the total work-group size is less than $2^{24}-1$, because faster `mul24` instructions 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`.

`requireddim`

Specifies the number of dimensions that will be used when the kernel is dispatched. `nd` must be a constant value of data type `u32` with the value 1, 2, or 3 (`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.

The results are undefined if the kernel is dispatched with a dimensions value that does not match `nd`.

With the use of this instruction, a finalizer might be able to generate better code for the `workitemid`, `workitemabsid`, `workitemflatid`, and `workitemflatabsid` instructions, 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 (`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.

The results are 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` instruction. Also, if the total grid size is less than $2^{24}-1$, then faster `mul24` instructions might be able to be used for the `workitemid`, `workitemabsid`, `workitemflatid`, and `workitemflatabsid` instructions, 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` instructions (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 `requiredddim` 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 (`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.

The results are 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` instructions.

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 `requiredddim` 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`.

`require nopartialworkgroups`

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.

The results are 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` instruction. A kernel agent might be able to dispatch a kernel more efficiently if it knows there are no partial work-groups.

The control directive applies to the whole kernel and all functions it calls. It can appear multiple times in a kernel or function. If it appears in a function (including external functions), then it must also appear in all kernels that call that function (or have been specified when the finalizer was invoked), either directly or indirectly.

If `require no partial work-groups` is specified when the finalizer is invoked, the kernel behaves as if the `require nopartialworkgroups` control directive has been specified.

CHAPTER 14.

module Header

This chapter describes the module header.

14.1 Syntax of the module Header

The `module` header specifies the module name, HSAIL version, the profile, target architecture, and default floating-point rounding mode required by the code in a module.

A single `module` header must appear at the top of each module, optionally preceded by only annotations (see [4.3.1. Annotations \(page 55\)](#)).

The syntax is:

```
module name : major : minor : profile : machine_model : default_float_rounding
```

name

The name of the module. The name must be a module scope identifier. See [4.6. Identifiers \(page 77\)](#).

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 instructions 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 instructions 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. 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 \(page 288\)](#).

machine_model

Specifies which machine model is used during finalization. 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 \(page 39\)](#).

default_float_rounding

Specifies which default floating-point rounding mode is used during finalization. Possibilities are:

- `$default` — Specifies that the finalizer must use the default floating-point rounding mode of the program that the module is added. If the program also has a default floating-point rounding mode of `default`, then the finalizer uses the default floating-point rounding mode of the kernel agent for which it is generating code, which can be either `zero` or `near`. The finalizer for a kernel agent must use the same default value for all finalizations regardless of the profile specified. An HSA runtime query can be used to determine the default floating-point rounding mode for a kernel agent.
- `$zero` — Specifies that `zero` floating-point rounding must be used. An error must be reported by the finalizer if the module header specifies the `$base` profile and the kernel agent does not support `zero` floating-point rounding mode for the Base profile.
- `$near` — Specifies that `near` floating-point rounding must be used. An error must be reported by the finalizer if the module header specifies the `$base` profile and the kernel agent does not support `near` floating-point rounding mode for the Base profile.

For more information, see [4.19.2. Floating-Point Rounding \(page 109\)](#) and [16.2.1. Base Profile Requirements \(page 289\)](#).

It is an error to add an HSAIL module to an HAIL program unless the rules defined in [4.2.1. Finalization \(page 49\)](#) are met.

See [4.2. Program, Code Object, and Executable \(page 48\)](#).

Examples

```
module &m1:1:0:$full:$small:$default;
module &m2:1:0:$full:$large:$zero;
module &m3:1:0:$base:$small:$near;
module &m41:0:$base:$large:$default;
```

CHAPTER 15.

Libraries

This chapter describes how to write HSAIL code for libraries.

15.1 Library Restrictions

HSAIL provides support for separately supplied HSAIL libraries.

Code written for an HSAIL 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 module that contains a call to a library should have a declaration specifying program linkage for each library function that it will call.

For HSAIL modules that use a library, the library module should be added to the HSAIL program before finalizing.

See [4.2. Program, Code Object, and Executable \(page 48\)](#) and [4.12. Linkage \(page 97\)](#).

15.2 Library Example

An example of library code is shown below:

```
module &lib1:0:$full:$small:$default;
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.
- All modules of an HSAIL program must specify the same profile. However, an application may have multiple HSAIL programs that specify different profiles. See [4.2. Program, Code Object, and Executable \(page 48\)](#).
- The required profile must be selected by a modifier on the `module` header. See [14.1. Syntax of the module Header \(page 284\)](#).
- 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 instructions on the types.
- The 16-bit floating-point type (`f16`), 32-bit floating-point type (`f32`) and all instructions on the types according to the declared profile. See [4.19.1. Floating-Point Numbers \(page 109\)](#).

- For all floating-point arithmetic instructions (see [5.11. Floating-Point Arithmetic Instructions \(page 140\)](#)); `cmp` with floating-point sources (see [5.18. Compare \(cmp\) Instruction \(page 155\)](#)); and `cvt` with a floating-point source type (see [5.19. Conversion \(cvt\) Instruction \(page 159\)](#)):
 - Must generate invalid operation exceptions for signaling NaN sources. Additionally, the signaling comparison forms of the `cmp` instruction 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 instructions (see [5.13. Floating-Point Bit Instructions \(page 146\)](#)) or native floating-point instructions (see [5.14. Native Floating-Point Instructions \(page 148\)](#)).

- The packed types and all instructions on the types with the exception of `f64x2`.
- Handling of `debugtrap` exceptions.

Both profiles are required to support all HSAIL requirements, except as specified in [16.2. Profile-Specific Requirements \(below\)](#).

See [Appendix A. Limits \(page 374\)](#) for details on limits that apply to both profiles.

The HSA runtime provides queries that enables an application to determine which optional features are available, the properties of implementation dependent features, and the values of implementation defined limits.

16.2 Profile-Specific Requirements

This section describes the requirements that an implementation must adhere to in order to claim support of the Base profile 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 result using the default rounding mode for `add`, `sub`, `mul`, `fma`, and `fractg` instructions.
 - Does not support the 64-bit floating-point type (`f64`), 64-bit packed floating-point type (`2xf64`), double-precision floating point constants, nor any instructions on the types.
 - Must provide `div` instructions within 2.5 ULP (see [4.19.6. Unit of Least Precision \(ULP\) \(page 112\)](#)) of the mathematically accurate result.
 - Must provide `sqrt` instructions less than or equal to 1 ULP of the mathematically accurate result.
 - All floating-point instructions (except `cvt`) that support the floating-point rounding mode must only allow the default floating-point rounding mode (see [4.19.2. Floating-Point Rounding \(page 109\)](#)).

- The `cvt` instruction from a floating-point type to a smaller floating-point type, and from integer type to floating-point type, must only allow the default floating-point rounding mode. The `cvt` instruction from floating-point type to integer type must only support the integer rounding modes (see [5.19.4. Description of Integer Rounding Modes \(page 162\)](#)) of `zeroi`, `zeroi_sat`, `szeroi`, and `szeroi_sat` (which correspond to the standard floating-point to integer conversion of C language).
 - Must flush subnormal values to zero. All HSAIL floating-point instructions must specify the `ftz` modifier (when `ftz` is valid).
 - For all floating-point arithmetic instructions (see [5.11. Floating-Point Arithmetic Instructions \(page 140\)](#)) and `cvt` with a floating-point source and destination type (see [5.19. Conversion \(cvt\) Instruction \(page 159\)](#)), 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 \(NaN\) \(page 111\)](#)).
 - 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.
- The finalizer must give an error if the rounding modifier is not omitted for an instruction that only allows the default floating-point rounding mode. The default floating-point rounding mode that will be used is specified by the module header ([14.1. Syntax of the module Header \(page 284\)](#)). The default floating-point rounding modes supported must be either `zero`, `near`, or both `zero` and `near`. An HSA runtime query is available to determine the floating-point rounding modes supported by a kernel agent if the Base profile is specified.
- The `icall` instruction is not supported. See [10.8. Indirect Call \(icall\) Instruction \(page 252\)](#).
- It is optional if the DETECT or BREAK exception policies (see [12.3. Hardware Exception Policies \(page 271\)](#)) for the five exceptions specified in [12.2. Hardware Exceptions \(page 269\)](#) are supported. An HSA runtime query can be used to determine the exceptions supported by the Base profile for the DETECT and BREAK policies for a kernel agent. See [4.19.5. Floating Point Exceptions \(page 112\)](#).
- An implementation is only required to support `system` scope on virtual address ranges allocated using the HSA runtime memory allocator for memory topology regions that support fine grain coherency (see [6.2.2. Memory Scope \(page 170\)](#)). In particular, it is not required that memory allocated by a system memory allocator support `system` scope.

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`, `sub`, `mul`, `fract`, `div`, `fma`, and `sqrt` instructions.
 - Must support the 64-bit floating-point type (`f64`), 64-bit packed floating-point type (`2xf64`), double-precision floating point constants and all instructions on the types.
 - Must support all floating-point rounding modes (see [4.19.2. Floating-Point Rounding \(page 109\)](#)) and all integer rounding modes ([5.19.4. Description of Integer Rounding Modes \(page 162\)](#)).

- Must support floating-point subnormal values.
- Must support the `ftz` modifier and IEEE/ANSI Standard 754-2008 gradual underflow.
- For all floating-point arithmetic instructions (see [5.11. Floating-Point Arithmetic Instructions \(page 140\)](#)) and `cvt` with a floating-point source and destination type (see [5.19. Conversion \(cvt\) Instruction \(page 159\)](#)), 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 \(NaN\) \(page 111\)](#).
 - 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.

The default floating-point rounding mode specified by the module header (see [14.1. Syntax of the module Header \(page 284\)](#)) must support both `zero` and `near`.

- Must support the DETECT exception policy and can optionally support the BREAK exception policy (see [12.3. Hardware Exception Policies \(page 271\)](#)) for the five exceptions specified in [12.2. Hardware Exceptions \(page 269\)](#). An HSA runtime query can be used to determine the exceptions supported by the Full profile for the DETECT and BREAK policies for a kernel agent. See [4.19.5. Floating Point Exceptions \(page 112\)](#).

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 instructions 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 hardware registers and remove the spills.

17.2 Using Lower-Precision Faster Instructions

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 instructions. For example, `div(src0, src1)` could be replaced by `src0 * nrcp(src1)` whenever the lower precision is permitted.

17.3 Functions

Function calls 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 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 instructions 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 instructions so that instructions 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 requirements of divergent control flow (see [2.12. Divergent Control Flow \(page 41\)](#)) 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 instructions that are involved in cross-lane interaction, such as barrier and cross-lane instructions (see [Chapter 9. Parallel Synchronization and Communication Instructions \(page 229\)](#)), to behave differently.

Consider the following pseudo HSAIL example:

```
if (x || y) {
    A;
    cross-lane-operation;
    B;
    if (x) {
        C;
    }
}
```

Reconverging control flow involving communication instructions later than the immediate post-dominator, as in the following pseudo machine code control flow, is not legal. It would result in the cross-lane instructions executing 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;
}
```

Also consider the following two pseudo HSAIL examples:

```

if (x) {
    A;
    cross-lane-operation;
} else {
    B;
    cross-lane-operation;
}

if (x) {
    cross-lane-operation;
    A;
} else {
    cross-lane-operation;
    B;
}

```

If control flow involving cross-lane instructions is reconverged earlier than the immediate post-dominator, or diverged later than the immediate dominator, as in the following pseudo machine code control flows, it is also not legal for the same reason.

```

if (x) {
    A;
} else {
    B;
}
cross-lane-operation;

cross-lane-operation;
if (x) {
    A;
} else {
    B;
}

```

In general it is not legal to clone instructions that can result in communication between lanes within a wavefront, or to hoist cross-lane instructions out of control flow, as that can change the execution behavior.

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 `ld` instructions 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 instructions to the group and global segment, and non-atomic image instructions to read-write images, cannot affect, or be affected by, other work-items or agents, except by an intervening synchronizing memory instruction or memory fence, as that would constitute a data race and so be undefined.

However, ordinary stores cannot be introduced that would not have been executed in the original program if they can introduce a data race. Consider the following pseudo HSAIL program where all memory instructions are ordinary:

```

Initial: x = y = 0;

Thread 1:
if (x == 1) {
    y = 1;
}

Thread 2:
if (y == 1) {
    x = 1;
}

Result: x == y == 0

```

The HSA memory model defines that this program does not have a data race as all reads that can influence the address, data or whether a write instruction is performed at all must appear to complete before the write instruction is initiated. Therefore, despite all memory instructions being ordinary, a compiler cannot introduce an ordinary store, even if the single work-item result would appear to be the same based on only considering the single work-item. Therefore, it would not be legal to transform it into the following pseudo machine code as that introduces a data race into a program that did not previously have a data-race and would likely cause results other than the only legal outcome:

```
Initial: x = y = 0;
```

```
Thread 1:
y = 1;
if (x != 1) {
    y = 0;
}

Thread 2:
x = 1;
if (y != 1) {
    x = 0;
}
```

```
Results: undefined as now has a data race
```

Atomic memory instructions, or ordinary memory instructions that are made visible to other work-items or agents through synchronizing memory instructions, 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 instructions 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, it should be marked with the `const` modifier. On some implementations, knowing a load is accessing constant memory might be more efficient. The results are 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 \(ld\) Instruction \(page 173\)](#).

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 \(page 69\)](#).

An HSAIL global or readonly segment variable definition marked with the `const` qualifier are required to have an initializer. The finalizer can replace usage of the variable by 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 do this replacement as the value 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 instructions should be marked with the `nonnull` modifier. On some implementations, knowing an address will not be the null pointer value might be more efficient. The results are undefined if a segment address conversion instruction 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 wavefronts 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.

In particular, this applies to accessing the global segment, even though the flat address of a global segment location is the same value as a global segment address to the same location, and that the null pointer value for a flat address and a global segment address are the same (see [2.8.3. Addressing for Segments \(page 35\)](#)). If a high-level compiler can determine that an address is either the null pointer value or an address in the global segment, it should use a global segment instruction rather than a flat instruction when accessing memory with the address, even though both produce the same result.

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 \(page 271\)](#)), 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 constant 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` instruction 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. Instructions 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 module 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 \(page 74\)](#).
- 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 instruction or directive. For example: an integer constant may be truncated from the textual value specified; an integer typed constant may be changed to an integer constant; a float typed constant may be changed to a float constant; a constant for a bit type may be changed to an integer or packed constant; or adjacent aggregate constant elements of the same type or array element type may be collapsed to a single array type element. See [18.6.1. Constant Operands \(page 340\)](#).
- The use of explicit instruction modifier values that are the default value used when the modifier is omitted (such as for `align`, `equiv`, `width`, and `zeroi` integer rounding mode).
- 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 byte size of the initializer divided by the variable element type. 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.
- BRIG has a `BrigDirectiveNone` directive which can be used to reserve space in the `hsa_code` section. But this has no representation in the HSAIL textual form.

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 BRIG Module

An HSAIL module (see [4.3. Module \(page 53\)](#)) is represented in BRIG as a single contiguous block of memory that contains the following elements:

- a `BrigModuleHeader`
- a BRIG section index
- three or more BRIG sections

These elements can be positioned within the BRIG module in any order, except that the `BrigModuleHeader` must start at offset 0 from the start of the BRIG module. Elements must not overlap.

The base of the `BrigModuleHeader` and each BRIG section is required to be 16-byte aligned, and the base of the BRIG section index is required to be 8-byte aligned. Padding between elements is only allowed in order to satisfy these alignment requirements and must be set to 0.

`BrigModule_t` is a pointer to the contiguous memory for a single BRIG module:

```
typedef BrigModuleHeader* BrigModule_t;
```

The BRIG section index is represented as an array of `uint64_t` offsets from the start of the BRIG module to the base of each BRIG section contained in the BRIG module. The order of the elements of the array do not have to match the order of the BRIG sections within the BRIG module.

BRIG defines three standard BRIG sections that are used to represent an HSAIL module:

- `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 \(page 319\)](#).
- `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 instructions which the finalizer uses to generate executable machine code. See [18.5. hsa_code Section \(page 320\)](#).
- `hsa_operand` — The operands of directives and instructions in the code section. For example, immediate values, registers, and address expressions. See [18.6. hsa_operand Section \(page 339\)](#).

The BRIG section index is indexed by `BrigSectionIndex` (see [18.3.31. BrigSectionIndex \(page 313\)](#)). The first three elements must be for the standard sections in the above order.

HSAIL supports an arbitrary number of additional sections that can come in any order after the standard sections in the section index. However, the layout of these sections, beyond the standard `BrigSectionHeader`, is not specified by HSAIL (see [18.3.32. BrigSectionHeader \(page 313\)](#)). 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.

Every BRIG section starts with a `BrigSectionHeader` which contains the section size, name and offset from the beginning of the section to the first entry. It must be 16 -byte aligned which allows sections to contain naturally aligned data up to 16 bytes in size. (Note, the standard sections actually only depend on being 4-byte aligned.) See [18.3.32. BrigSectionHeader \(page 313\)](#).

For the standard 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 these entries 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 \(page 302\)](#)) 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 encoded explicitly. This allows future expansion of BRIG directives, instructions, or operands to add additional fields at the end of entries. The use of a length field allows old finalizers to process new BRIG sections (ignoring any new fields).

A reference between entries in the `hsa_code` and `hsa_operand` sections is encoded as a byte offset from the beginning of the section that contains the referenced entry (not from the beginning of the BRIG module). The offset is represented as a `uint32_t` that must be a multiple of 4. Therefore, the standard sections are limited to 4 GiB. (Note, non-standard sections can be any size as the `BrigSectionHeader` uses `uint64_t` for the section size.) See [18.3.1. Section Offsets \(below\)](#).

A number of entries in the `hsa_code` and `hsa_operand` sections (for example, `BrigDirectiveControl` and `BrigOperandCodeList`) refer to a variable length list of other entries. A list is represented as a single entry in the `hsa_data` section that is an array of `uint32_t` 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.

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 bytes.

BRIG structures are accessible in C language style using structs. (C++ language classes are not used.) All standard BRIG values are stored in little endian format: including the fields in `BrigModuleHeader`, the fields in `BrigSectionHeader`, the section index elements, the fields in all entries in the `hsa_code` and `hsa_operand` sections, and all data values in the `hsa_data` section. The endian format of the non-standard sections, beyond the standard `BrigSectionHeader` header, is implementation defined.

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 or byte data.
- `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 \(page 69\)](#).

```
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 \(page 69\)](#).

```
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 instructions.

```
typedef uint8_t BrigAluModifier8_t;
enum BrigAluModifierMask {
    BRIG_ALU_FTZ = 1
};
```

- `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 instruction does not support the `ftz` modifier, then must be a 0 value. Otherwise, a 0 value means it is absent and a 1 value means it is present.

18.3.5 BrigAtomicOperation

`BrigAtomicOperation` is used to specify the type of atomic memory, signal and atomic image instructions. For more information, see [6.5. Atomic Memory Instructions \(page 180\)](#) and [6.8. Notification \(signal\) Instructions \(page 187\)](#).

```
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 [18.3.15. BrigKind \(page 305\)](#).

18.3.7 BrigCompareOperation

`BrigCompareOperation` is used to specify the type of compare operation. For more information, see [5.18. Compare \(cmp\) Instruction \(page 155\)](#).

```
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 \(page 278\)](#).

```
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_REQUIREDDIM = 6,
    BRIG_CONTROL_REQUIREDGRIDSIZE = 7,
    BRIG_CONTROL_REQUIREDWORKGROUPSIZE = 8,
    BRIG_CONTROL_REQUIRENOPARTIALWORKGROUPS = 9
};
```

18.3.9 BrigExceptionsMask

BrigExceptionsMask defines the bit mask used to specify a set of exceptions for each of the five exceptions specified in [12.2. Hardware Exceptions \(page 269\)](#). For more information, see [11.2. Exception Instructions \(page 260\)](#).

```
typedef uint32_t BrigExceptions32_t;
enum BrigExceptionsMask {
    BRIG_EXCEPTIONS_INVALID_OPERATION = 1 << 0,
    BRIG_EXCEPTIONS_DIVIDE_BY_ZERO = 1 << 1,
    BRIG_EXCEPTIONS_OVERFLOW = 1 << 2,
    BRIG_EXCEPTIONS_UNDERFLOW = 1 << 3,
    BRIG_EXCEPTIONS_INEXACT = 1 << 4,
```

```

    BRIG_EXCEPTIONS_FIRST_USER_DEFINED = 1 << 16
};

```

Bits 5 through 15 are reserved, but bits 16 to 32 are available for implementation defined extensions.

18.3.10 BrigExecutableModifierMask

`BrigExecutableModifierMask` defines bit masks that can be used to access properties about an executable kernel or function.

```

typedef uint8_t BrigExecutableModifier8_t;
enum BrigExecutableModifierMask {
    BRIG_EXECUTABLE_DEFINITION = 1
};

```

- `BRIG_EXECUTABLE_DEFINITION` — A bit mask that can be used to select the setting for whether an executable is a declaration or a definition. A 0 value means a declaration and a 1 value means a definition.

See [18.5.1.5. BrigDirectiveExecutable \(page 322\)](#).

18.3.11 BrigImageChannelOrder

`BrigImageChannelOrder` is used to specify the order of image components. For more information, see [7.1.4.1. Channel Order \(page 198\)](#).

```

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_RGX = 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,
    BRIG_CHANNEL_ORDER_FIRST_USER_DEFINED = 128
};

```

Values 20 through 127 are reserved, but values 128 to 255 are available for implementation defined extensions.

18.3.12 BrigImageChannelType

`BrigImageChannelType` is used to specify the image channel type. For more information, see [7.1.4.2. Channel Type \(page 200\)](#).

```

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_INT_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,
BRIG_CHANNEL_TYPE_FIRST_USER_DEFINED = 128
};

```

Values 16 through 127 are reserved, but values 128 to 255 are available for implementation defined extensions.

18.3.13 BrigImageGeometry

`BrigImageGeometry` is used to specify the number of coordinates needed to access an image. For more information, see [7.1.3. Image Geometry \(page 196\)](#).

```

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,
    BRIG_GEOMETRY_FIRST_USER_DEFINED = 128
};

```

Values 8 through 127 are reserved, but values 128 to 255 are available for implementation defined extensions.

18.3.14 BrigImageQuery

`BrigImageQuery` is used to specify the image property being queried by the `queryimage` instruction. For more information, see [7.5. Query Image and Query Sampler Instructions \(page 224\)](#).

```

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
};

```

18.3.15 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 BrigKind16_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_FBARRIER = 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_MODULE = 0x100b,
    BRIG_KIND_DIRECTIVE_PRAGMA = 0x100c,
    BRIG_KIND_DIRECTIVE_SIGNATURE = 0x100d,
    BRIG_KIND_DIRECTIVE_VARIABLE = 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_ALIGN = 0x3001,
    BRIG_KIND_OPERAND_CODE_LIST = 0x3002,
    BRIG_KIND_OPERAND_CODE_REF = 0x3003,
    BRIG_KIND_OPERAND_CONSTANT_BYTES = 0x3004,
    BRIG_KIND_OPERAND_RESERVED = 0x3005,
    BRIG_KIND_OPERAND_CONSTANT_IMAGE = 0x3006,
    BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST = 0x3007,
    BRIG_KIND_OPERAND_CONSTANT_SAMPLER = 0x3008,
    BRIG_KIND_OPERAND_OPERAND_LIST = 0x3009,
    BRIG_KIND_OPERAND_REGISTER = 0x300a,
    BRIG_KIND_OPERAND_STRING = 0x300b,
    BRIG_KIND_OPERAND_WAVESIZE = 0x3009c,
    BRIG_KIND_OPERAND_END = 0x300d
};

```

18.3.16 BrigLinkage

BrigLinkage is used to specify linkage. For more information, see [4.12. Linkage \(page 97\)](#).

```

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
};

```

18.3.17 BrigMachineModel

`BrigMachineModel` is used to specify the kind of machine model. For more information, see [2.9. Small and Large Machine Models \(page 39\)](#).

```

typedef uint8_t BrigMachineModel8_t;
enum BrigMachineModel {
    BRIG_MACHINE_SMALL = 0,
    BRIG_MACHINE_LARGE = 1
};

```

18.3.18 BrigMemoryModifierMask

`BrigMemoryModifierMask` defines bit masks that can be used to access the modifiers for memory instructions.

```

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 1 value means it is present. If the instruction does not support the `const` modifier, then the value must be 0.

18.3.19 BrigMemoryOrder

`BrigMemoryOrder` is used to specify the memory order of an atomic memory instruction. For more information, see [6.2.1. Memory Order \(page 169\)](#).

```

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
};

```

18.3.20 BrigMemoryScope

`BrigMemoryScope` is used to specify the memory scope for an atomic memory, signal or memory fence instruction. For more information, see [6.2.2. Memory Scope \(page 170\)](#).

```

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_AGENT = 4,
    BRIG_MEMORY_SCOPE_SYSTEM = 5
};

```

18.3.21 BrigModuleHeader

The first entry in a BRIG module must be `BrigModuleHeader`. It must be 16-byte aligned. See [18.2. BRIG Module \(page 299\)](#).

Syntax is:

```
struct BrigModuleHeader {
    char identification[8];
    BrigVersion32_t brigMajor;
    BrigVersion32_t brigMinor;
    uint64_t byteCount;
    uint8_t hash[64];
    uint32_t reserved;
    uint32_t sectionCount;
    uint64_t sectionIndex;
};
```

Fields are:

- `char identification[8]` — A magic number used to identify that this is a BRIG module. Must have the ASCII character string value of “HSA BRIG”.
- `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 [18.3.38. BrigVersion \(page 318\)](#).
- `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 [18.3.38. BrigVersion \(page 318\)](#).
- `uint64_t byteCount` — Size in bytes of the contiguous chunk of memory that contains the entire BRIG module, including the section index, all the sections and any padding between sections. Must be a multiple of 16.
- `uint8_t hash[64]` — A 512-bit value that can be used as a hash of the contents of the BRIG module. The hash function used, and the data included, is implementation dependent. If unused then must be set to all zero.
- `uint32_t reserved` — Must be 0.
- `uint32_t sectionCount` — Number of sections in the module. Must be at least 3 for the standard sections. See [18.2. BRIG Module \(page 299\)](#).
- `uint64_t sectionIndex` — Byte offset from start of the BRIG module to the base of the BRIG section index. Must be a multiple of 8. There must be exactly `sectionCount` entries of type `uint64_t` in the the array. See [18.2. BRIG Module \(page 299\)](#).

18.3.22 BrigOpcode

BrigOpcode is used to specify the opcode for the HSAIL instruction.

```
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_NRCF = 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_RDIMAGE = 78,
BRIG_OPCODE_LDIMAGE = 79,
BRIG_OPCODE_STIMAGE = 80,
BRIG_OPCODE_IMAGEFENCE = 81,
BRIG_OPCODE_QUERYIMAGE = 82,
BRIG_OPCODE_QUERYAMPLER = 83,
BRIG_OPCODE_CBR = 84,
BRIG_OPCODE_BR = 85,
BRIG_OPCODE_SBR = 86,
BRIG_OPCODE_BARRIER = 87,
BRIG_OPCODE_WAVEBARRIER = 88,
BRIG_OPCODE_ARRIVEFBAR = 89,
BRIG_OPCODE_INITFBAR = 90,
BRIG_OPCODE_JOINFBAR = 91,
BRIG_OPCODE_LEAVEFBAR = 92,
BRIG_OPCODE_RELEASEFBAR = 93,
BRIG_OPCODE_WAITFBAR = 94,
BRIG_OPCODE_LDF = 95,
BRIG_OPCODE_ACTIVELANECOUNT = 96,
BRIG_OPCODE_ACTIVELANEID = 97,
BRIG_OPCODE_ACTIVELANEMASK = 98,
BRIG_OPCODE_ACTIVELANEPERMUTE = 99,
BRIG_OPCODE_CALL = 100,
BRIG_OPCODE_SCALL = 101,
BRIG_OPCODE_ICALL = 102,
BRIG_OPCODE_RET = 103,
BRIG_OPCODE_ALLOCA = 104,
BRIG_OPCODE_CURRENTWORKGROUPSIZE = 105,
BRIG_OPCODE_CURRENTWORKITEMFLATID = 106,
BRIG_OPCODE_DIM = 107,
BRIG_OPCODE_GRIDGROUPS = 108,
BRIG_OPCODE_GRIDSIZE = 109,
BRIG_OPCODE_PACKETCOMPLETIONSIG = 110,
BRIG_OPCODE_PACKETID = 111,
BRIG_OPCODE_WORKGROUPID = 112,
BRIG_OPCODE_WORKGROUPSIZE = 113,
BRIG_OPCODE_WORKITEMABSID = 114,
BRIG_OPCODE_WORKITEMFLATABSID = 115,
BRIG_OPCODE_WORKITEMFLATID = 116,
BRIG_OPCODE_WORKITEMID = 117,
BRIG_OPCODE_CLEARDETECTEXCEPT = 118,
BRIG_OPCODE_GETDETECTEXCEPT = 119,
BRIG_OPCODE_SETDETECTEXCEPT = 120,
BRIG_OPCODE_ADDQUEUEWRITEINDEX = 121,
BRIG_OPCODE_CASQUEUEWRITEINDEX = 122,
BRIG_OPCODE_LDQUEUEWRITEINDEX = 123,
BRIG_OPCODE_LDQUEUEWRITEINDEX = 124,
BRIG_OPCODE_STQUEUEWRITEINDEX = 125,
BRIG_OPCODE_STQUEUEWRITEINDEX = 126,
BRIG_OPCODE_CLOCK = 127,
BRIG_OPCODE_CUID = 128,
BRIG_OPCODE_DEBUGTRAP = 129,
BRIG_OPCODE_GROUPBASEPTR = 1230,
BRIG_OPCODE_KERNARGBASEPTR = 131,
BRIG_OPCODE_LANEID = 132,
BRIG_OPCODE_MAXCUID = 133,
BRIG_OPCODE_MAXWAVEID = 134,
BRIG_OPCODE_NULLPTR = 135,

```

```
    BRIG_OPCODE_WAVEID = 136,
};
```

Values 136 through 32767 are reserved, but values 32768 to 65535 are available for implementation defined extensions.

18.3.23 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 \(page 101\)](#).

```
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
};
```

18.3.24 BrigProfile

BrigProfile is used to specify the kind of profile. For more information, see [16.1. What Are Profiles? \(page 288\)](#).

```
typedef uint8_t BrigProfile8_t;
enum BrigProfile {
    BRIG_PROFILE_BASE = 0,
    BRIG_PROFILE_FULL = 1
};
```

18.3.25 BrigRegisterKind

BrigRegisterKind is used to specify the kind of HSAIL register. For more information, see [4.7. Registers \(page 79\)](#).

```
typedef uint16_t BrigRegisterKind16_t;
enum BrigRegisterKind {
    BRIG_REGISTER_KIND_CONTROL = 0,
    BRIG_REGISTER_KIND_SINGLE = 1,
    BRIG_REGISTER_KIND_DOUBLE = 2,
    BRIG_REGISTER_KIND_QUAD = 3
};
```

18.3.26 BrigRound

BrigRound is used to specify rounding. For more information, see [4.19.2. Floating-Point Rounding \(page 109\)](#) and [5.19.3. Rules for Rounding for Conversions \(page 162\)](#).

If the instruction does not support a rounding mode, then `BRIG_ROUND_NONE` must be used.

If the instruction supports a floating-point rounding mode but does not explicitly specify one, then `BRIG_ROUND_FLOAT_DEFAULT` must be specified. If the instruction supports an integer rounding mode but does not explicitly specify one, then `BRIG_ROUND_INTEGER_ZERO` must be specified. Otherwise, the appropriate rounding mode must be used.

```
typedef uint8_t BrigRound8_t;
enum BrigRound {
    BRIG_ROUND_NONE = 0,
    BRIG_ROUND_FLOAT_DEFAULT = 1,
    BRIG_ROUND_FLOAT_NEAR_EVEN = 2,
    BRIG_ROUND_FLOAT_ZERO = 3,
    BRIG_ROUND_FLOAT_PLUS_INFINITY = 4,
    BRIG_ROUND_FLOAT_MINUS_INFINITY = 5,
    BRIG_ROUND_INTEGER_NEAR_EVEN = 6,
    BRIG_ROUND_INTEGER_ZERO = 7,
    BRIG_ROUND_INTEGER_PLUS_INFINITY = 8,
    BRIG_ROUND_INTEGER_MINUS_INFINITY = 9,
    BRIG_ROUND_INTEGER_NEAR_EVEN_SAT = 10,
    BRIG_ROUND_INTEGER_ZERO_SAT = 11,
    BRIG_ROUND_INTEGER_PLUS_INFINITY_SAT = 12,
    BRIG_ROUND_INTEGER_MINUS_INFINITY_SAT = 13,
    BRIG_ROUND_INTEGER_SIGNALING_NEAR_EVEN = 14,
    BRIG_ROUND_INTEGER_SIGNALING_ZERO = 15,
    BRIG_ROUND_INTEGER_SIGNALING_PLUS_INFINITY = 16,
    BRIG_ROUND_INTEGER_SIGNALING_MINUS_INFINITY = 17,
    BRIG_ROUND_INTEGER_SIGNALING_NEAR_EVEN_SAT = 18,
    BRIG_ROUND_INTEGER_SIGNALING_ZERO_SAT = 19,
    BRIG_ROUND_INTEGER_SIGNALING_PLUS_INFINITY_SAT = 20,
    BRIG_ROUND_INTEGER_SIGNALING_MINUS_INFINITY_SAT = 21
};
```

18.3.27 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 \(page 207\)](#).

```
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,
    BRIG_ADDRESSING_FIRST_USER_DEFINED = 128
};
```

Values 5 through 127 are reserved, but values 128 to 255 are available for implementation defined extensions.

18.3.28 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 \(page 206\)](#).

```
typedef uint8_t BrigSamplerCoordNormalization8_t;
enum BrigSamplerCoordNormalization {
    BRIG_COORD_UNNORMALIZED = 0,
    BRIG_COORD_NORMALIZED = 1
};
```

18.3.29 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 \(page 209\)](#).

```
typedef uint8_t BrigSamplerFilter8_t;
enum BrigSamplerFilter {
    BRIG_FILTER_NEAREST = 0,
    BRIG_FILTER_LINEAR = 1,
    BRIG_FILTER_FIRST_USER_DEFINED = 128
};
```

Values 2 through 127 are reserved, but values 128 to 255 are available for implementation defined extensions.

18.3.30 BrigSamplerQuery

`BrigSamplerQuery` is used to specify the sampler property being queried by the `query_sampler` instruction. For more information, see [7.5. Query Image and Query Sampler Instructions \(page 224\)](#).

```
typedef uint8_t BrigSamplerQuery8_t;
enum BrigSamplerQuery {
    BRIG_SAMPLER_QUERY_ADDRESSING = 0,
    BRIG_SAMPLER_QUERY_COORD = 1,
    BRIG_SAMPLER_QUERY_FILTER = 2
};
```

18.3.31 BrigSectionIndex

A BRIG module can have a number of BRIG sections. Every module must have a data, code and operand section with the indices in the BRIG section index array defined by `BrigSectionIndex`. Any additional sections have an index starting after these. See [18.2. BRIG Module \(page 299\)](#).

```
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
};
```

18.3.32 BrigSectionHeader

The first entry in every BRIG section must be `BrigSectionHeader`. It must be 16-byte aligned. See [18.2. BRIG Module \(page 299\)](#).

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 {
    uint64_t byteCount;
    uint32_t headerByteCount;
    uint32_t nameLength;
    uint8_t name[1];
};
```

Field is:

- `uint64_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 from the beginning of the section to 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 section name may be followed by any implementation specific data. This must be followed by sufficient zero padding bytes to make `headerByteCount` a multiple of 4.

18.3.33 BrigSegCvtModifierMask

`BrigSegCvtModifierMask` defines bit masks that can be used to access the modifiers for instructions which convert between segment and flat addresses.

```
typedef uint8_t BrigSegCvtModifier8_t;
enum BrigSegCvtModifierMask {
    BRIG_SEG_CVT_NONULL = 1
};
```

- `BRIG_SEG_CVT_NONULL` — A bit mask that can be used to select the setting for the `nonnull` modifier. A 0 value means it is absent and a 1 value means it is present. If the instruction does not support the `nonnull` modifier, then the value must be 0.

18.3.34 BrigSegment

`BrigSegment` is used to specify the memory segment for a symbol or memory address. In the case of a memory address, it can also specify that a flat address is being used. For more information, see [2.8. Segments \(page 31\)](#).

```
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,
    BRIG_SEGMENT_FIRST_USER_DEFINED = 128
};
```

Values 9 through 127 are reserved, but values 128 to 255 are available for implementation defined extensions.

18.3.35 BrigType

`BrigType` is used to specify the data compound type of instructions, operands, and variables.

The `BrigType` enumeration is encoded to make it easy to determine if the type is packed or an array. If packed, it is also easy to determine the packed element compound type and the bit size of the packed type. If array, it is also easy to determine the array element compound type.

The base type is encoded in the bottom 5 bits, the packed type size recorded in the next 2 bits, and whether it is an array type in the next bit.

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.

NOTE: An array of the b1 type is not allowed.

For more information, see [4.13. Data Types \(page 99\)](#).

```
enum {
    BRIG_TYPE_BASE_SIZE = 5,
    BRIG_TYPE_PACK_SIZE = 2,
    BRIG_TYPE_ARRAY_SIZE = 1,

    BRIG_TYPE_BASE_SHIFT = 0,
    BRIG_TYPE_PACK_SHIFT = BRIG_TYPE_BASE_SHIFT + BRIG_TYPE_BASE_SIZE,
    BRIG_TYPE_ARRAY_SHIFT = BRIG_TYPE_PACK_SHIFT + BRIG_TYPE_PACK_SIZE,

    BRIG_TYPE_BASE_MASK = ((1 << BRIG_TYPE_BASE_SIZE) - 1) << BRIG_TYPE_BASE_SHIFT,
    BRIG_TYPE_PACK_MASK = ((1 << BRIG_TYPE_PACK_SIZE) - 1) << BRIG_TYPE_PACK_SHIFT,
    BRIG_TYPE_ARRAY_MASK = ((1 << BRIG_TYPE_ARRAY_SIZE) - 1) << BRIG_TYPE_ARRAY_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,

    BRIG_TYPE_ARRAY = 1 << BRIG_TYPE_ARRAY_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,

BRIG_TYPE_U8_ARRAY = BRIG_TYPE_U8 | BRIG_TYPE_ARRAY,
BRIG_TYPE_U16_ARRAY = BRIG_TYPE_U16 | BRIG_TYPE_ARRAY,
BRIG_TYPE_U32_ARRAY = BRIG_TYPE_U32 | BRIG_TYPE_ARRAY,
BRIG_TYPE_U64_ARRAY = BRIG_TYPE_U64 | BRIG_TYPE_ARRAY,

BRIG_TYPE_S8_ARRAY = BRIG_TYPE_S8 | BRIG_TYPE_ARRAY,
BRIG_TYPE_S16_ARRAY = BRIG_TYPE_S16 | BRIG_TYPE_ARRAY,
BRIG_TYPE_S32_ARRAY = BRIG_TYPE_S32 | BRIG_TYPE_ARRAY,
BRIG_TYPE_S64_ARRAY = BRIG_TYPE_S64 | BRIG_TYPE_ARRAY,

BRIG_TYPE_F16_ARRAY = BRIG_TYPE_F16 | BRIG_TYPE_ARRAY,
BRIG_TYPE_F32_ARRAY = BRIG_TYPE_F32 | BRIG_TYPE_ARRAY,
BRIG_TYPE_F64_ARRAY = BRIG_TYPE_F64 | BRIG_TYPE_ARRAY,

BRIG_TYPE_B8_ARRAY = BRIG_TYPE_B8 | BRIG_TYPE_ARRAY,
BRIG_TYPE_B16_ARRAY = BRIG_TYPE_B16 | BRIG_TYPE_ARRAY,
BRIG_TYPE_B32_ARRAY = BRIG_TYPE_B32 | BRIG_TYPE_ARRAY,
BRIG_TYPE_B64_ARRAY = BRIG_TYPE_B64 | BRIG_TYPE_ARRAY,
BRIG_TYPE_B128_ARRAY = BRIG_TYPE_B128 | BRIG_TYPE_ARRAY,

BRIG_TYPE_SAMP_ARRAY = BRIG_TYPE_SAMP | BRIG_TYPE_ARRAY,
BRIG_TYPE_ROIMG_ARRAY = BRIG_TYPE_ROIMG | BRIG_TYPE_ARRAY,
BRIG_TYPE_WOIMG_ARRAY = BRIG_TYPE_WOIMG | BRIG_TYPE_ARRAY,
BRIG_TYPE_RWIMG_ARRAY = BRIG_TYPE_RWIMG | BRIG_TYPE_ARRAY,

BRIG_TYPE_SIG32_ARRAY = BRIG_TYPE_SIG32 | BRIG_TYPE_ARRAY,
BRIG_TYPE_SIG64_ARRAY = BRIG_TYPE_SIG64 | BRIG_TYPE_ARRAY,

BRIG_TYPE_U8X4_ARRAY = BRIG_TYPE_U8X4 | BRIG_TYPE_ARRAY,
BRIG_TYPE_U8X8_ARRAY = BRIG_TYPE_U8X8 | BRIG_TYPE_ARRAY,

```

```

BRIG_TYPE_U8X16_ARRAY = BRIG_TYPE_U8X16 | BRIG_TYPE_ARRAY,

BRIG_TYPE_U16X2_ARRAY = BRIG_TYPE_U16X2 | BRIG_TYPE_ARRAY,
BRIG_TYPE_U16X4_ARRAY = BRIG_TYPE_U16X4 | BRIG_TYPE_ARRAY,
BRIG_TYPE_U16X8_ARRAY = BRIG_TYPE_U16X8 | BRIG_TYPE_ARRAY,

BRIG_TYPE_U32X2_ARRAY = BRIG_TYPE_U32X2 | BRIG_TYPE_ARRAY,
BRIG_TYPE_U32X4_ARRAY = BRIG_TYPE_U32X4 | BRIG_TYPE_ARRAY,

BRIG_TYPE_U64X2_ARRAY = BRIG_TYPE_U64X2 | BRIG_TYPE_ARRAY,

BRIG_TYPE_S8X4_ARRAY = BRIG_TYPE_S8X4 | BRIG_TYPE_ARRAY,
BRIG_TYPE_S8X8_ARRAY = BRIG_TYPE_S8X8 | BRIG_TYPE_ARRAY,
BRIG_TYPE_S8X16_ARRAY = BRIG_TYPE_S8X16 | BRIG_TYPE_ARRAY,

BRIG_TYPE_S16X2_ARRAY = BRIG_TYPE_S16X2 | BRIG_TYPE_ARRAY,
BRIG_TYPE_S16X4_ARRAY = BRIG_TYPE_S16X4 | BRIG_TYPE_ARRAY,
BRIG_TYPE_S16X8_ARRAY = BRIG_TYPE_S16X8 | BRIG_TYPE_ARRAY,

BRIG_TYPE_S32X2_ARRAY = BRIG_TYPE_S32X2 | BRIG_TYPE_ARRAY,
BRIG_TYPE_S32X4_ARRAY = BRIG_TYPE_S32X4 | BRIG_TYPE_ARRAY,

BRIG_TYPE_S64X2_ARRAY = BRIG_TYPE_S64X2 | BRIG_TYPE_ARRAY,

BRIG_TYPE_F16X2_ARRAY = BRIG_TYPE_F16X2 | BRIG_TYPE_ARRAY,
BRIG_TYPE_F16X4_ARRAY = BRIG_TYPE_F16X4 | BRIG_TYPE_ARRAY,
BRIG_TYPE_F16X8_ARRAY = BRIG_TYPE_F16X8 | BRIG_TYPE_ARRAY,

BRIG_TYPE_F32X2_ARRAY = BRIG_TYPE_F32X2 | BRIG_TYPE_ARRAY,
BRIG_TYPE_F32X4_ARRAY = BRIG_TYPE_F32X4 | BRIG_TYPE_ARRAY,

BRIG_TYPE_F64X2_ARRAY = BRIG_TYPE_F64X2 | BRIG_TYPE_ARRAY
};

```

18.3.36 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:

$$\text{value} = (\text{uint64_t}(\text{hi}) \ll 32) \mid \text{uint64_t}(\text{lo})$$
- `uint32_t hi` — The high 32 bits of the 64-bit integer.

18.3.37 BrigVariableModifierMask

BrigVariableModifierMask defines bit masks that can be used to access properties about a variable.

```

typedef uint8_t BrigVariableModifier8_t;
enum BrigVariableModifierMask {
    BRIG_VARIABLE_DEFINITION = 1,
    BRIG_VARIABLE_CONST = 2
};

```

- `BRIG_VARIABLE_DEFINITION` — A bit mask that can be used to select the setting for whether a variable is a declaration or a definition. A 0 value means a declaration and a 1 value means a definition.
- `BRIG_VARIABLE_CONST` — A bit mask that can be used to select the setting for the `const` qualifier. A 0 value means the variable can change value after it has been created and initialized; a 1 value means the variable value will not change after it has been created and initialized. Only global or read-only segment variables can be constant.

See [18.5.1.13. BrigDirectiveVariable \(page 327\)](#).

18.3.38 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)*.

```
typedef uint32_t BrigVersion32_t;
enum BrigVersion {
    BRIG_VERSION_HSAIL_MAJOR = 1,
    BRIG_VERSION_HSAIL_MINOR = 0,
    BRIG_VERSION_BRIG_MAJOR = 1,
    BRIG_VERSION_BRIG_MINOR = 0
};
```

- `BRIG_VERSION_HSAIL_MAJOR` — The major version of this revision of the HSAIL virtual ISA specification. This is the value used in the `module` header `major` operand. See [Chapter 14. module Header \(page 284\)](#). 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 `module` header `minor` operand. See [Chapter 14. module Header \(page 284\)](#). 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.

18.3.39 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 \(page 41\)](#).

```
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 [18.3.32. BrigSectionHeader \(page 313\)](#).

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 \(page 300\)](#).

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 BRIG module. They appear in the same order 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 [18.3.32. BrigSectionHeader \(page 313\)](#).

All entries in the `hsa_code` section must start with a `BrigBase` structure (see [18.3.6. BrigBase \(page 302\)](#)). The `kind` field of `BrigBase` specifies the kind of the entry, which also indicates if it is a directive entry (see [18.5.1. Directive Entries \(below\)](#)) or instruction entry (see [18.5.2. Instruction Entries \(page 329\)](#)).

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 module header, 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 [18.3.15. BrigKind \(page 305\)](#).

The table below shows the possible formats for the directives. Every directive uses one of these formats.

Table 18–1 Formats of Directives in the hsa_code Section

Name	Description
BrigDirectiveArgBlock	Start and end of an arg block. See 18.5.1.2. BrigDirectiveArgBlock (below) .
BrigDirectiveComment	Comment string. See 18.5.1.3. BrigDirectiveComment (below) .
BrigDirectiveControl	Assorted finalizer controls. See 18.5.1.4. BrigDirectiveControl (next page) .
BrigDirectiveExecutable	Describes a kernel, function or signature. See 18.5.1.5. BrigDirectiveExecutable (next page) .
BrigDirectiveExtension	Used to enable device-specific extensions. See 18.5.1.6. BrigDirectiveExtension (page 324) .
BrigDirectiveFbarrier	Used for fbarrier definitions. See 18.5.1.7. BrigDirectiveFbarrier (page 324) .
BrigDirectiveLabel	Declare a label. See 18.5.1.8. BrigDirectiveLabel (page 325) .
BrigDirectiveLoc	Source-level line position. See 18.5.1.9. BrigDirectiveLoc (page 325) .
BrigDirectiveModule	Module name, HSAIL version, and target information. See 18.5.1.10. BrigDirectiveModule (page 326) .
BrigDirectiveNone	Special directive that is always ignored. See 18.5.1.11. BrigDirectiveNone (page 326) .
BrigDirectivePragma	Additional information to control the finalizer and other consumers of HSAIL. See 18.5.1.12. BrigDirectivePragma (page 327) .
BrigDirectiveVariable	Declares a variable. See 18.5.1.13. BrigDirectiveVariable (page 327) .

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 \(page 62\)](#).

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`.
- `BrigDataOffsetString32_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 \(page 278\)](#).

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 \(page 303\)](#).
- `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_CONSTANT_BYTES` 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 \(page 56\)](#)):

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 \(page 58\)](#) and [10.3. Function Declarations, Function Definitions, and Function Signatures \(page 247\)](#)):

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 \(page 248\)](#)):

1. `BrigDirectiveExecutable` with kind of `BRIG_KIND_DIRECTIVE_SIGNATURE`
2. Zero or more signature output formal arguments (currently HSAIL only supports at most one output formal argument)
3. Zero or more signature input formal arguments
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 of a signature formal argument 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;
    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 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.
- `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 [18.3.10. BrigExecutableModifierMask \(page 304\)](#).
- `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 [18.3.16. BrigLinkage \(page 306\)](#).
- `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 \(page 274\)](#).

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` declarations and definitions.

Syntax is:

```
struct BrigDirectiveFbarrier {
    BrigBase base;
    BrigDataOffsetString32_t name;
    BrigVariableModifier8_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.
- `BrigVariableModifier8_t modifier` — Modifier for the `fbarrier`. The `BRIG_VARIABLE_DEFINITION` must be 0 if a declaration; and 1 if a definition. The values for other bitmask fields must be 0. See [18.3.37. BrigVariableModifierMask \(page 317\)](#).

- `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 \(page 78\)](#) and [18.3.16. BrigLinkage \(page 306\)](#)
- `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 `.linecpp` directive. For more information, see [13.2. loc Directive \(page 276\)](#).

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 instruction which follows this directive corresponds to `line`. Multiple `BrigDirectiveLoc` statements can refer to the same `line`.
- `uint32_t column` — The finalizer and other tools should assume that the instruction which follows this directive corresponds to `column`. Multiple `BrigDirectiveLoc` statements can refer to the same `column`.

18.5.1.10 BrigDirectiveModule

`BrigDirectiveModule` specifies the module name, HSAIL virtual ISA specification version, and target information. For more information, see [Chapter 14. module Header \(page 284\)](#).

There must be exactly one `BrigDirectiveModule` directive in the `hsa_code` section. It may be optionally preceded only by `BrigDirectiveComment`, `BrigDirectiveLoc` and `BrigDirectivePragma` directives.

Syntax is:

```
struct BrigDirectiveModule {
    BrigBase base;
    BrigDataOffsetString32_t name;
    BrigVersion32_t hsailMajor;
    BrigVersion32_t hsailMinor;
    BrigProfile8_t profile;
    BrigMachineModel8_t machineModel;
    BrigRound8_t defaultFloatRound;
    uint8_t reserved;
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_DIRECTIVE_MODULE`.
- `BrigDataOffsetString32_t name` — Byte offset to the place in the `hsa_data` section giving the name of the module.
- `BrigVersion32_t hsailMajor` — The HSAIL virtual 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 [18.3.38. BrigVersion \(page 318\)](#).
- `BrigVersion32_t hsailMinor` — The HSAIL virtual 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 [18.3.38. BrigVersion \(page 318\)](#).
- `BrigProfile8_t profile` — The profile. A member of the `BrigProfile` enumeration. See [18.3.24. BrigProfile \(page 311\)](#).
- `BrigMachineModel8_t machineModel` — The machine model. A member of the `BrigMachineModel` enumeration. See [18.3.17. BrigMachineModel \(page 307\)](#).
- `BrigRound8_t defaultFloatRound` — The default floating-point rounding mode. A member of the `BrigRound` enumeration: only `BRIG_ROUND_FLOAT_DEFAULT`, `BRIG_ROUND_FLOAT_NEAR_EVEN`, and `BRIG_ROUND_FLOAT_ZERO` are allowed. See [18.3.26. BrigRound \(page 311\)](#).
- `uint8_t reserved;` — Must be 0.

18.5.1.11 BrigDirectiveNone

The `BrigDirectiveNone` format is a special format that allows a tool to overwrite long instructions 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:

- `BrigBase base` — `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.

18.5.1.12 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 \(page 276\)](#).

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_CONSTANT_BYTES`, `BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST`, `BRIG_KIND_OPERAND_CONSTANT_IMAGE`, `BRIG_KIND_OPERAND_CONSTANT_SAMPLER`, `BRIG_KIND_OPERAND_REGISTER`, `BRIG_KIND_OPERAND_CODE_REF`, `BRIG_KIND_OPERAND_STRING`, or `BRIG_KIND_OPERAND_WAVESIZE`.

If any operand is a constant, it must be compatible with the rules in [18.6.1. Constant Operands \(page 340\)](#).

18.5.1.13 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 constant operand with a `kind` field of `BRIG_KIND_CONSTANT_BYTES`, `BRIG_KIND_CONSTANT_OPERAND_LIST`, `BRIG_KIND_OPERAND_CONSTANT_IMAGE`, or `BRIG_KIND_OPERAND_CONSTANT_SAMPLER`.

The byte size of the constant must match the byte size of the variable.

The constant must be compatible with the type of the variable specified by the `type` field according to the rules in [18.6.1. Constant Operands \(page 340\)](#).

- `BrigType 16_t type` — The BRIG type of the variable. If the variable is an array then must be an array type. `BRIG_TYPE_B1` is not allowed.
- `BrigSegment8_t segment` — Segment that will hold the variable. A member of the `BrigSegment` enumeration. See [18.3.34. BrigSegment \(page 314\)](#).
- `BrigAlignment8_t align` — The required variable 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 \(page 301\)](#).
- `BrigUInt64 dim` — The array dimension size *dim*. See [18.3.36. BrigUInt64 \(page 317\)](#).

The variable is an array if the `type` field is an array type.

If the variable is not an array, then *dim* must be 0. If the variable is an array with a size, then *dim* must be the number of elements in the array which is not allowed to be 0. If the variable is an array without a size, but with an initializer, then *dim* must be set to the number of elements specified by the size of the initializer. Otherwise, the array variable must be the last argument of a function (see [10.4. Variadic Functions \(page 248\)](#)) or a declaration with no size specified, and *dim* must be set to 0.

- `BrigVariableModifier8_t modifier` — Modifier for the variable. See [18.3.37. BrigVariableModifierMask \(page 317\)](#).
- `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 \(page 78\)](#) and [18.3.16. BrigLinkage \(page 306\)](#).
- `BrigAllocation8_t allocation` — Values are specified by the `BrigAllocation` enumeration. For global segment variable must be `BRIG_ALLOCATION_PROGRAM` or `BRIG_ALLOCATION_AGENT` depending on the allocation specified; for readonly segment variable must be `BRIG_ALLOCATION_AGENT`; otherwise must be `BRIG_ALLOCATION_AUTOMATIC`. See [18.3.3. BrigAllocation \(page 301\)](#).
- `uint8_t reserved` — Must be 0.

18.5.2 Instruction Entries

BRIG instructions corresponding to HSAIL instructions. They can only appear in the code block of kernels and functions. The finalizer uses these to generate executable machine code for kernels and indirect functions.

Every `BrigInst*` must start with a `BrigInstBase`. See [18.5.2.1. BrigInstBase \(below\)](#).

The table below shows the possible formats for the instructions. Every instruction uses one of these formats.

Table 18–2 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 (below) .
<code>BrigInstAddr</code>	Address instructions. See 18.5.2.2. BrigInstAddr (next page) .
<code>BrigInstAtomic</code>	Atomic instructions. See 18.5.2.3. BrigInstAtomic (next page) .
<code>BrigInstBasic</code>	Used for all instructions that require no extra modifier information. See 18.5.2.4. BrigInstBasic (page 331) .
<code>BrigInstBr</code>	Branch, call, barrier and fbarrier instructions. See 18.5.2.5. BrigInstBr (page 331) .
<code>BrigInstCmp</code>	Compare instruction. See 18.5.2.6. BrigInstCmp (page 332) .
<code>BrigInstCvt</code>	Conversion instruction. See 18.5.2.7. BrigInstCvt (page 332) .
<code>BrigInstImage</code>	Image-related instructions. See 18.5.2.8. BrigInstImage (page 333) .
<code>BrigInstLane</code>	Cross lane instructions. See 18.5.2.9. BrigInstLane (page 333) .
<code>BrigInstMem</code>	Load and store memory instructions. See 18.5.2.10. BrigInstMem (page 334) .
<code>BrigInstMemFence</code>	Memory fence instruction. See 18.5.2.11. BrigInstMemFence (page 335) .
<code>BrigInstMod</code>	Instructions with a single modifier, such as a rounding mode. See 18.5.2.12. BrigInstMod (page 335) .
<code>BrigInstQueryImage</code>	Image query instructions. See 18.5.2.13. BrigInstQueryImage (page 336) .
<code>BrigInstQuerySampler</code>	Sampler query instruction. See 18.5.2.14. BrigInstQuerySampler (page 336) .
<code>BrigInstQueue</code>	User Mode Queue instructions. See 18.5.2.15. BrigInstQueue (page 337) .
<code>BrigInstSeg</code>	Instructions with memory segments. See 18.5.2.16. BrigInstSeg (page 337) .
<code>BrigInstSegCvt</code>	Instructions which convert between segment and flat addresses. See 18.5.2.17. BrigInstSegCvt (page 338) .
<code>BrigInstSignal</code>	Signal instructions. See 18.5.2.18. BrigInstSignal (page 338) .
<code>BrigInstSourceType</code>	Instructions that have different types for their destination and source operands. See 18.5.2.19. BrigInstSourceType (page 338) .

18.5.2.1 BrigInstBase

Every other `BrigInst*` must start with `BrigInstBase` which in turn starts with `BrigBase`.

Syntax is:

```
struct BrigInstBase {
    BrigBase base;
    BrigOpcodel6_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 [18.3.15. BrigKind \(page 305\)](#).
- `BrigOpcode16_t opcode` — Opcode associated with the instruction.
- `BrigType16_t type` — Data type of the destination of the instruction. If the instruction 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 instruction 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 $(4 * \text{number of operands})$. Any destination operand is first, followed by any source operands.

If any operand is a constant, it must be compatible with the rules in [18.6.1. Constant Operands \(page 340\)](#). The operand kinds allowed for each `opcode` value are defined in [18.7. BRIG Syntax for Instructions \(page 348\)](#).

18.5.2.2 BrigInstAddr

The `BrigInstAddr` format is used for address instructions.

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 instruction.
- `BrigSegment8_t segment` — Segment. A member of the `BrigSegment` enumeration. If the instruction does not specify a segment, this field must be set to `BRIG_SEGMENT_FLAT`. See [18.3.34. BrigSegment \(page 314\)](#).
- `uint8_t reserved[3]` — Must be 0.

18.5.2.3 BrigInstAtomic

The `BrigInstAtomic` format is used for atomic and atomic no return instructions.

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 instruction.
- `BrigSegment8_t segment` — Segment. A member of the `BrigSegment` enumeration. If the instruction 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 [18.3.34. BrigSegment \(page 314\)](#).
- `BrigMemoryOrder8_t memoryOrder` — Memory order of the atomic instruction. See [18.3.19. BrigMemoryOrder \(page 307\)](#).
- `BrigMemoryScope8_t memoryScope` — Memory scope of the atomic instruction. If `segment` is `BRIG_SEGMENT_GLOBAL` or `BRIG_SEGMENT_FLAT` then must be `BRIG_MEMORY_SCOPE_WAVEFRONT`, `BRIG_MEMORY_SCOPE_WORKGROUP`, `BRIG_MEMORY_SCOPE_AGENT`, or `BRIG_MEMORY_SCOPE_SYSTEM`. If `segment` is `BRIG_SEGMENT_GROUP` then must be `BRIG_MEMORY_SCOPE_WAVEFRONT` or `BRIG_MEMORY_SCOPE_WORKGROUP`. See [18.3.20. BrigMemoryScope \(page 307\)](#).
- `BrigAtomicOperation8_t atomicOperation` — The atomic instruction such as `add` or `or`. The wait atomic instructions are not allowed. See [18.3.5. BrigAtomicOperation \(page 302\)](#).
- `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 \(page 168\)](#).
- `uint8_t reserved[3]` — Must be 0.

18.5.2.4 BrigInstBasic

The `BrigInstBasic` format is used for all instructions 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 instruction.

18.5.2.5 BrigInstBr

The `BrigInstBr` format is used for the branch, call, barrier and fbarrier instructions.

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 instruction has no typed operands.
- `BrigWidth8_t width` — The width modifier. If the instruction does not support the width modifier, then this must be `BRIG_WIDTH_ALL` for the direct branch and direct call instructions, and `BRIG_WIDTH_WAVESIZE` for the wavebarrier instruction. If the instruction supports the width modifier but does not specify it, then this must be the default value defined by the instruction: for indirect branch and indirect call instructions, it is `BRIG_WIDTH_1`; for barrier instructions, it is `BRIG_WIDTH_ALL`; and for the fbarrier instructions, it is `BRIG_WIDTH_WAVESIZE`. Otherwise, this must be the value from `BrigWidth` that corresponds to the specified width modifier. See [18.3.39. BrigWidth \(page 318\)](#).
- `uint8_t reserved[3]` — Must be 0.

18.5.2.6 BrigInstCmp

The `BrigInstCmp` format is used for compare instructions. The compare instruction needs a special format because it has a comparison operator and a second type.

Syntax is:

```
struct BrigInstCmp {
    BrigInstBase base;
    BrigType16_t sourceType;
    BrigAluModifier8_t modifier;
    BrigCompareOperation8_t compare;
    BrigPack8_t pack;
    uint8_t reserved[3];
};
```

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 instruction: for packed compares, must be `u` with the same length as `sourceType`.
- `BrigType16_t sourceType` — Type of the sources.
- `BrigAluModifier8_t modifier` — The modifier flags for this instruction. See [18.3.4. BrigAluModifierMask \(page 301\)](#).
- `BrigCompareOperation8_t compare` — The specific comparison (greater than, less than, and so forth).
- `BrigPack8_t pack` — Packing control. See [18.3.23. BrigPack \(page 311\)](#).
- `uint8_t reserved[3]` — Must be 0.

18.5.2.7 BrigInstCvt

The `BrigInstCvt` format is used for conversion instructions.

Syntax is:

```
struct BrigInstCvt {
    BrigInstBase base;
    BrigType16_t sourceType;
    BrigAluModifier8_t modifier;
    BrigRound8_t round;
};
```

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 instruction.
- `BrigType16_t sourceType` — Type of the sources.
- `BrigAluModifier8_t modifier` — The modifier flags for this instruction. See [18.3.4. BrigAluModifierMask \(page 301\)](#).
- `BrigRound8_t round` — Rounding mode. See [18.3.26. BrigRound \(page 311\)](#).

18.5.2.8 BrigInstImage

The `BrigInstImage` format is used for the image instructions.

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 instruction.
- `BrigType16_t imageType` — Type of the image. Must be `BRIG_TYPE_ROIMG`, `BRIG_TYPE_WOIMG` or `BRIG_TYPE_RWIMG`.
- `BrigType16_t coordType` — Type of the coordinates.
- `BrigImageGeometry8_t geometry` — Image geometry. See [18.3.13. BrigImageGeometry \(page 305\)](#).
- `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 \(page 168\)](#).
- `uint16_t reserved` — Must be 0.

18.5.2.9 BrigInstLane

The `BrigInstLane` format is used for cross-lane instructions.

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 instruction.
- `BrigType16_t sourceType` — Type of the source. If the instruction does not have a source type modifier then must be `BRIG_TYPE_NONE`.
- `BrigWidth8_t width` — The width modifier. If the instruction does not specify the width modifier, then this must be `BRIG_WIDTH_1` (the default for the cross-lane instructions). Otherwise, this must be the value from `BrigWidth` that corresponds to the specified width modifier. See [18.3.39. BrigWidth \(page 318\)](#).
- `uint16_t reserved` — Must be 0.

18.5.2.10 BrigInstMem

The `BrigInstMem` format is used for memory instructions.

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 instruction.
- `BrigSegment8_t segment` — Segment. A member of the `BrigSegment` enumeration. If the instruction does not support the segment modifier, then this must be `BRIG_SEGMENT_NONE`. If the instruction supports the segment modifier but does not specify it, then this must be `BRIG_SEGMENT_FLAT`. See [18.3.34. BrigSegment \(page 314\)](#).
- `BrigAlignment8_t align` — The align modifier. If the instruction does not specify the align modifier, then this must be `BRIG_ALIGNMENT_1` (the default for memory instructions). Otherwise, this must be the value from `BrigAlignment` that corresponds to the specified align modifier. See [18.3.2. BrigAlignment \(page 301\)](#).
- `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 \(page 168\)](#).

- `BrigWidth8_t width` — The width modifier. If the instruction does not support the width modifier, then this must be `BRIG_WIDTH_NONE`. If the instruction supports the width modifier but does not specify it, then this must be `BRIG_WIDTH_1` (the default for memory instructions). Otherwise, this must be the value from `BrigWidth` that corresponds to the specified width modifier. See [18.3.39. BrigWidth \(page 318\)](#).
- `BrigMemoryModifier8_t modifier` — Memory modifier flags of the instruction. See [18.3.18. BrigMemoryModifierMask \(page 307\)](#).
- `uint8_t reserved[3]` — Must be 0.

18.5.2.11 BrigInstMemFence

The `BrigInstMemFence` format is used for the `memfence` instruction.

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 a memory fence instruction has no destination operand.
- `BrigMemoryOrder8_t memoryOrder` — Memory order of the memory fence instruction. Must be `BRIG_MEMORY_ORDER_SC_ACQUIRE`, `BRIG_MEMORY_ORDER_SC_RELEASE`, or `BRIG_MEMORY_ORDER_SC_ACQUIRE_RELEASE`. See [18.3.19. BrigMemoryOrder \(page 307\)](#).
- `BrigMemoryScope8_t globalSegmentMemoryScope` — Memory scope for the global segment of the memory fence instruction. Must be `BRIG_MEMORY_SCOPE_WAVEFRONT`, `BRIG_MEMORY_SCOPE_WORKGROUP`, `BRIG_MEMORY_SCOPE_AGENT`, or `BRIG_MEMORY_SCOPE_SYSTEM`. See [18.3.20. BrigMemoryScope \(page 307\)](#).
- `BrigMemoryScope8_t groupSegmentMemoryScope` — Memory scope for the group segment of the memory fence instruction. Must be the same value as `globalSegmentMemoryScope` as the memory orders currently supported by `memfence` synchronize with both the group and global segment. See [18.3.20. BrigMemoryScope \(page 307\)](#).
- `BrigMemoryScope8_t imageSegmentMemoryScope` — Memory scope for the image segment of the memory fence instruction. The image segment is not one of the regular segments, but is implicitly used by the image instructions to access image data. Must be `BRIG_MEMORY_SCOPE_NONE` as `memfence` does not currently support synchronizing with the image segment. See [18.3.20. BrigMemoryScope \(page 307\)](#).

18.5.2.12 BrigInstMod

The `BrigInstMod` format is used for ALU instructions with a modifier.

Syntax is:

```
struct BrigInstMod {
    BrigInstBase base;
    BrigAluModifier8_t modifier;
    BrigPack8_t pack;
    BrigRound8_t round;
    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 instruction.
- `BrigAluModifier8_t modifier` — The modifier flags for this instruction. See [18.3.4. BrigAluModifierMask \(page 301\)](#).
- `BrigRound8_t round` — Rounding mode. See [18.3.26. BrigRound \(page 311\)](#).
- `BrigPack8_t pack` — Packing control. If the instruction does not have a packing modifier, this must be set to `BRIG_PACK_NONE`. See [18.3.23. BrigPack \(page 311\)](#).
- `uint8_t reserved` — Must be 0.

18.5.2.13 BrigInstQueryImage

The `BrigInstQueryImage` format is used for the `queryimage` instruction.

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 instruction.
- `BrigType16_t imageType` — Type of the image. Must be `BRIG_TYPE_ROIMG`, `BRIG_TYPE_WOIMG`, or `BRIG_TYPE_RWIMG`.
- `BrigImageGeometry8_t geometry` — Image geometry. See [18.3.13. BrigImageGeometry \(page 305\)](#).
- `BrigImageQuery8_t query` — Image property being queried. See [18.3.14. BrigImageQuery \(page 305\)](#).

18.5.2.14 BrigInstQuerySampler

The `BrigInstQuerySampler` format is used for the `querysampler` instruction.

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 instruction.
- `BrigSamplerQuery8_t query` — Sampler property being queried. See [18.3.30. BrigSamplerQuery \(page 313\)](#).
- `uint8_t reserved[3]` — Must be 0.

18.5.2.15 BrigInstQueue

The `BrigInstQueue` format is used for User Mode Queue instructions.

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 User Mode Queue instruction.
- `BrigSegment8_t segment` — Segment. A member of the `BrigSegment` enumeration. If the instruction does not specify a segment, this field must be set to `BRIG_SEGMENT_FLAT`. See [18.3.34. BrigSegment \(page 314\)](#).
- `BrigMemoryOrder8_t memoryOrder` — Memory order of the User Mode Queue instruction. See [18.3.19. BrigMemoryOrder \(page 307\)](#).
- `uint16_t reserved` — Must be 0.

18.5.2.16 BrigInstSeg

The `BrigInstSeg` format is used for instructions 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 instruction.
- `BrigSegment8_t segment` — Segment. A member of the `BrigSegment` enumeration. If the instruction does not specify a segment, this field must be set to `BRIG_SEGMENT_FLAT`. See [18.3.34. BrigSegment \(page 314\)](#).
- `uint8_t reserved[3]` — Must be 0.

18.5.2.17 BrigInstSegCvt

The `BrigInstSegCvt` format is used for instructions 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 instruction.
- `BrigType16_t sourceType` — Type of the source.
- `BrigSegment8_t segment` — Segment. A member of the `BrigSegment` enumeration. See [18.3.34. BrigSegment \(page 314\)](#).
- `BrigSegCvtModifier8_t modifier` — Segment conversion modifier flags of the instruction. See [18.3.33. BrigSegCvtModifierMask \(page 314\)](#).

18.5.2.18 BrigInstSignal

The `BrigInstSignal` format is used for signal instructions.

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 instruction.
- `BrigType16_t signalType` — Type of the signal. Must be `BRIG_TYPE_SIG32` or `BRIG_TYPE_SIG64`.
- `BrigMemoryOrder8_t memoryOrder` — Memory order of the signal instruction. See [18.3.19. BrigMemoryOrder \(page 307\)](#).
- `BrigAtomicOperation8_t signalOperation` — The signal instruction such as add or or. See [18.3.5. BrigAtomicOperation \(page 302\)](#).

18.5.2.19 BrigInstSourceType

The `BrigInstSourceType` format is used for instructions 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 instruction.
- `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 BRIG module.

The `hsa_operand` section must start with a `BrigSectionHeader` entry. The name of the section must be `hsa_operand`. See [18.3.32. BrigSectionHeader \(page 313\)](#).

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 [18.3.15. BrigKind \(page 305\)](#).

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 18–3 Formats of Operands in the `hsa_operand` Section

Name	Description
<code>BrigOperandAddress</code>	Used for address expressions. See 18.6.2. BrigOperandAddress (page 341) .
<code>BrigOperandAlign</code>	Used for aligning aggregate data constants. See 18.6.3. BrigOperandAlign (page 341) .
<code>BrigOperandCodeList</code>	List of references to entries in the <code>hsa_code</code> section. See 18.6.4. BrigOperandCodeList (page 342) .
<code>BrigOperandCodeRef</code>	A reference to an entry in the <code>hsa_code</code> section. See 18.6.5. BrigOperandCodeRef (page 342) .
<code>BrigOperandConstantBytes</code>	Declares a constant value as an array of bytes. See 18.6.6. BrigOperandConstantBytes (page 343) .
<code>BrigOperandConstantImage</code>	Declares the properties of an image referenced by an image handle constant. See 18.6.7. BrigOperandConstantImage (page 344) .
<code>BrigOperandConstantOperandList</code>	Declares a constant as a list of operands. See 18.6.8. BrigOperandConstantOperandList (page 345) .
<code>BrigOperandConstantSampler</code>	Declares the properties of a sampler referenced by a sample handle constant. 18.6.9. BrigOperandConstantSampler (page 346) .
<code>BrigOperandOperandList</code>	List of references to entries in the <code>hsa_operand</code> section. See 18.6.10. BrigOperandOperandList (page 346) .
<code>BrigOperandRegister</code>	A register (<code>c</code> , <code>s</code> , <code>d</code> , or <code>q</code>). See 18.6.11. BrigOperandRegister (page 347) .

Name	Description
<code>BrigOperandString</code>	A textual string. See 18.6.12. <code>BrigOperandString</code> (page 347).
<code>BrigOperandWavesize</code>	The wavesize operand. See 18.6.13. <code>BrigOperandWavesize</code> (page 347).

18.6.1 Constant Operands

Constant values are represented by operands with a `kind` field of `BRIG_KIND_OPERAND_CONSTANT_BYTES`, `BRIG_KIND_OPERAND_CONSTANT_IMAGE`, `BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST`, or `BRIG_KIND_OPERAND_CONSTANT_SAMPLER`. The type of the constant is given by the `type` field of the `BrigOperandConstant*`.

A constant operand may be used as:

- The value of a variable initializer: the expected data type is given by the `type` field of the `BrigDirectiveVariable`.
- The operand of an instruction: the expected data type is the type of the corresponding instruction operand which depends on the actual instruction which is given by the `base.opcode` field of the `BrigInst*`.
- The operand of a directive: the expected data type is the type of the corresponding directive operand which depends on the actual directive which is given by the `base.kind` field of the `BrigDirective*`.
- An element of an array typed constant: the expected data type is the array element type of the array typed constant.
- An element of an aggregate constant: the expected type is the type of the constant itself which is always a typed constant.

The data type of the constant must correspond to the expected data type as described below:

- If the expected type is `b1`, the constant type must be `u8` and have a value of 0 or 1.
- If the expected type is `b128`, the constant type must be `u8x16`.
- If the expected type is a bit type, other than `b1` or `b128`, the constant type must be an unsigned integer type of the same byte size as the expected type.
- If the expected type is a bit type array (note that a `b1` array type is not allowed), the constant must be an aggregate constant represented as a `BrigOperandConstantOperandList` with a `type` field of `BRIG_TYPE_NONE`. The byte size of the aggregate constant value must be the same byte size as expected type array.
- In all other cases, the constant type must be the same as the expected type. If the constant type is an array type, then the byte size of the constant must be the same as the byte size of the expected type array. If the constant type is a signal or signal array type, then the value must be 0.

It is allowed to canonicalize a series of adjacent aggregate constant elements into an array typed constant if it denotes the same byte value. This allows a series of constants to be collapsed into a single `BrigOperandConstantBytes`. However, such collapsing is only valid if endianness properties are preserved which are impacted by the data type byte size and by whether the type is a packed type.

It is allowed to canonicalize an aggregate constant element that is an array typed constant into a list of aggregate elements for each element of the array typed constant. This allows an aggregate element that is an array of image or sampler typed constants to become a list of the image or sampler constants. This avoids the need for an `BrigOperandConstantOperandList` to denote the array.

These canonicalizations can be combined to create more compact BRIG that still preserves the same value of the constant independent of endianness. This can be important for large byte size constants to improve the performance of processing the BRIG.

18.6.2 BrigOperandAddress

`BrigOperandAddress` is used for address expressions. See [4.18. Address Expressions \(page 106\)](#).

Syntax is:

```
struct BrigOperandAddress {
    BrigBase base;
    BrigCodeOffset32_t symbol;
    BrigOperandOffset32_t reg;
    BrigUInt64 offset;
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_OPERAND_ADDRESS`.
- `BrigCodeOffset32_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 \(page 321\)](#). If the address expression has no symbol name then must be 0.
- `BrigOperandOffset32_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 [18.3.36. BrigUInt64 \(page 317\)](#). If the address expression has no offset then *offset* must be 0. If the type of the address expression is `u32`, the `hi` field of the `BrigUInt64` must be 0. The finalizer will order the bytes of the *offset* value according to the byte endianness of the HSA platform for which code is being generated.

18.6.3 BrigOperandAlign

`BrigOperandAlign` is used for aligning aggregate data constants. See [4.8.4. Aggregate Constants \(page 91\)](#).

Syntax is:

```
struct BrigOperandAlign {
    BrigBase base;
    BrigAlignment8_t align;
    uint8_t reserved[3];
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_OPERAND_ALIGN`.
- `BrigAlignment8_t align` — The required alignment in bytes for the next aggregate constant element. Causes zero padding between aggregate constant elements, or zero fill if the last

aggregate constant element. See [18.3.2. BrigAlignment](#) (page 301).

- `uint8_t reserved[3]` — Must be 0.

18.6.4 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 is available. The `byteCount` of the array must be exactly $(4 * \text{number of elements})$.
 - 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](#) (page 321).
 - When used as a label list, each element must reference a `BRIG_KIND_DIRECTIVE_LABEL` directive in the same function scope.

18.6.5 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 \(page 321\)](#).

18.6.6 BrigOperandConstantBytes

`BrigOperandConstantBytes` specifies a constant value as an array of bytes.

Syntax is:

```
struct BrigOperandConstantBytes {
    BrigBase base;
    BrigType16_t type;
    uint16_t reserved;
    BrigDataOffsetString32_t bytes;
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_OPERAND_CONSTANT_BYTES`.
- `BrigType16_t type` — Data type of the constant. Must be an integer type, floating-point type, packed type, signal type, or array of integer type, float type, packed type or signal type. Note, bit types or array of bit types are not allowed.
- `uint16_t reserved` — Must be 0.
- `BrigDataOffsetString32_t bytes` — Byte offset into the place in the `hsa_data` section where the bytes of the constant value are available. The byte size is the `byteCount` of the `BrigData`.

This operand is used to represent the value of all integer constants, float constants, integer typed constants, float typed constants, packed typed constants, signal typed constants, array typed constants of integer, float, packed and signal. Note that HSAIL has no bit typed constants.

If `type` is a non-array type then the element type is `type`. The byte size must be the size of the element type. If the element type is an integer type then the constant corresponds to an integer constant or integer typed constant. If the element type is a float type then the constant corresponds to a float constant or float typed constant. If the element type is a packed type then the constant corresponds to a packed typed constant. If the element type is a signal type then the constant corresponds to a signal typed constant: the bytes must have a value of 0.

If `type` is an array type then the element type is the array element type of `type`. The byte size must be an integral multiple of the element type. The constant corresponds to an array typed constant. The element type must be an integer, float, packed or signal type. If the element type is a signal type the bytes must be 0.

The data is stored in the `hsa_data` section as a stream of consecutive values of the element type, with each value encoded from least significant to most significant byte (little endian byte format). The elements of an array typed constant are encoded in the element order. The elements of a packed typed constant are encoded in the reverse element order. However, when finalized the bytes are ordered according to the byte endianness of the HSA platform for which code is being generated. Note that for packed typed constants, both the bytes of the elements and the order of the elements must be reversed if not finalizing for a little endian byte format HSA platform.

18.6.7 BrigOperandConstantImage

`BrigOperandConstantImage` specifies the properties of an image referenced by an image handle constant. For more information, see [7.1.7. Image Creation and Image Handles \(page 211\)](#).

Syntax is:

```
struct BrigOperandConstantImage {
    BrigBase base;
    BrigType16_t type;
    BrigImageGeometry8_t geometry;
    BrigImageChannelOrder8_t channelOrder;
    BrigImageChannelType8_t channelType;
    uint8_t reserved[3];
    BrigUInt64 width;
    BrigUInt64 height;
    BrigUInt64 depth;
    BrigUInt64 array;
};
```

Fields are:

- `BrigDirectiveKinds16_t kind` — `base.kind` must be `BRIG_KIND_OPERAND_CONSTANT_IMAGE`.
- `BrigType16_t type` — Data type of the constant. Must be `BRIG_TYPE_ROIMG`, `BRIG_TYPE_WOIMG`, or `BRIG_TYPE_RWIMG`.
- `BrigImageGeometry8_t geometry` — Geometry for the image. A member of the `BrigImageGeometry` enumeration. See [18.3.13. BrigImageGeometry \(page 305\)](#).
- `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 [18.3.11. BrigImageChannelOrder \(page 304\)](#).

- `BrigImageChannelType8_t channelType` — Channel type for storing images. Images can be stored and accessed in assorted formats. A member of the `BrigImageChannelType` enumeration. See [18.3.12. BrigImageChannelType \(page 304\)](#).
- `uint8_t reserved[3]` — Must be 0.
- `BrigUInt64 width` — The image width. Must be greater than zero for all image geometries.
- `BrigUInt64 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.
- `BrigUInt64 depth` — The image depth. Must be greater than zero if geometry is `BRIG_GEOMETRY_3D`; otherwise must be 0.
- `BrigUInt64 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

18.6.8 BrigOperandConstantOperandList

`BrigOperandConstantOperandList` specifies the data value.

Syntax is:

```
struct BrigOperandConstantOperandList {
    BrigBase base;
    BrigType16_t type;
    uint16_t reserved;
    BrigDataOffsetOperandList32_t elements;
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST`.
- `BrigType16_t type` — Must be an image or sampler array type if an array typed constant, or `BRIG_TYPE_NONE` if an aggregate constant.
- `uint16_t reserved` — Must be 0.
- `BrigDataOffsetOperandList32_t elements` — 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 is available. The `byteCount` of the array must be exactly $(4 * \text{number of operands})$. The operands must either be `BRIG_KIND_OPERAND_CONSTANT_BYTES`, `BRIG_KIND_OPERAND_CONSTANT_IMAGE`, `BRIG_KIND_OPERAND_CONSTANT_SAMPLER`, or `BRIG_KIND_OPERAND_ALIGN`.

If the constant is an aggregate constant, then the `type` field must be `BRIG_TYPE_NONE`. The operands must either be `BRIG_KIND_OPERAND_CONSTANT_BYTES`, `BRIG_KIND_OPERAND_CONSTANT_IMAGE`, `BRIG_KIND_OPERAND_CONSTANT_SAMPLER`, or `BRIG_KIND_OPERAND_ALIGN` that correspond to the elements of the aggregate constant. If an element of the aggregate constant is an image or sampler array typed constant, then the array typed constant elements are represented directly as elements of the aggregate constant. The byte size of the constant is the sum of the byte sizes of the operands, accounting for any padding created by any `BRIG_KIND_OPERAND_ALIGN` operands.

If the constant is an array typed constant, then the `type` field must be an array type with an image or sampler array element type. The operands must either all be `BRIG_KIND_OPERAND_CONSTANT_IMAGE` for an image array type, or all be `BRIG_KIND_OPERAND_CONSTANT_SAMPLER` for a sampler array type. The byte size of the constant is the byte size of the array element type multiplied by the number of operands.

18.6.9 BrigOperandConstantSampler

`BrigOperandConstantSampler` specifies the properties of a sampler referenced by a sampler handle constant. For more information, see [7.1.8. Sampler Creation and Sampler Handles \(page 214\)](#).

Syntax is:

```
struct BrigOperandConstantSampler {
    BrigBase base;
    BrigType16_t type;
    BrigSamplerCoordNormalization8_t coord;
    BrigSamplerFilter8_t filter;
    BrigSamplerAddressing8_t addressing;
    uint8_t reserved[3];
};
```

Fields are:

- `BrigDirectiveKinds16_t kind` — `base.kind` must be `BRIG_KIND_OPERAND_CONSTANT_SAMPLER`.
- `BrigType16_t type` — Data type of the constant. Must be `BRIG_TYPE_SAMP`.
- `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 [18.3.28. BrigSamplerCoordNormalization \(page 312\)](#).
- `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 [18.3.29. BrigSamplerFilter \(page 313\)](#).
- `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 [18.3.27. BrigSamplerAddressing \(page 312\)](#).
- `uint8_t reserved[3]` — Must be 0.

18.6.10 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`.
- `BrigDataOffsetOperandList32_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 $(4 * \text{number of elements})$.
 - When used as a destination vector operand, each element must reference a `BRIG_KIND_OPERAND_REGISTER` directive.
 - When used as a source vector operand, each element must reference a `BRIG_KIND_OPERAND_REGISTER`, `BRIG_KIND_OPERAND_CONSTANT_BYTES`, or `BRIG_KIND_OPERAND_WAVESIZE` directive.

18.6.11 BrigOperandRegister

`BrigOperandRegister` is used for a register (c, s, d, or q).

Syntax is:

```
struct BrigOperandRegister {
    BrigBase base;
    BrigRegisterKind16_t regKind;
    uint16_t regNum;
};
```

Fields are:

- `BrigBase base` — `base.kind` must be `BRIG_KIND_OPERAND_REGISTER`.
- `BrigRegisterKind16_t regKind` — The register kind. Must be `BRIG_REG_KIND_CONTROL` for c register, `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.

18.6.12 BrigOperandString

`BrigOperandString` is used for a textual string.

Syntax is:

```
struct BrigOperandString {
    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.

18.6.13 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 Instructions

This section describes the BRIG syntax for instructions.

18.7.1 BRIG Syntax for Arithmetic Instructions

Some instructions support modifiers that have default values. These instructions 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 Instructions

Table 18–4 BRIG Syntax for Integer Arithmetic Instructions

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_REGISTER`.

src: must be `BRIG_KIND_OPERAND_REGISTER`, `BRIG_KIND_OPERAND_CONSTANT_BYTES`, or `BRIG_KIND_OPERAND_WAVESIZE`.

18.7.1.2 BRIG Syntax for Integer Optimization Instruction

Table 18–5 BRIG Syntax for Integer Optimization Instruction

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
BRIG_OPCODE_MAD	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>

dest: must be BRIG_KIND_OPERAND_REGISTER.

src: must be BRIG_KIND_OPERAND_REGISTER, BRIG_KIND_OPERAND_CONSTANT_BYTES, or BRIG_KIND_OPERAND_WAVESIZE.

18.7.1.3 BRIG Syntax for 24-Bit Integer Optimization Instructions

Table 18–6 BRIG Syntax for 24-Bit Integer Optimization Instructions

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_REGISTER.

src: must be BRIG_KIND_OPERAND_REGISTER, BRIG_KIND_OPERAND_CONSTANT_BYTES, or BRIG_KIND_OPERAND_WAVESIZE.

18.7.1.4 BRIG Syntax for Integer Shift Instructions

Table 18–7 BRIG Syntax for Integer Optimization Instructions

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_REGISTER.

src: must be BRIG_KIND_OPERAND_REGISTER, BRIG_KIND_OPERAND_CONSTANT_BYTES, or BRIG_KIND_OPERAND_WAVESIZE.

18.7.1.5 BRIG Syntax for Individual Bit Instructions

Table 18–8 BRIG Syntax for Individual Bit Instructions

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_REGISTER.

src: must be BRIG_KIND_OPERAND_REGISTER, BRIG_KIND_OPERAND_CONSTANT_BYTES, or BRIG_KIND_OPERAND_WAVESIZE.

18.7.1.6 BRIG Syntax for Bit String Instructions

Table 18–9 BRIG Syntax for Bit String Instructions

Opcode	Format	Oper. 0	Oper. 1	Oper. 2	Oper. 3	Oper. 4
BRIG_OPCODE_BITEXTRACT	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>src</i>	<i>src</i>	<i>src</i>	
BRIG_OPCODE_BITINSERT	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>src</i>	<i>src</i>	<i>src</i>	<i>src</i>
BRIG_OPCODE_BITMASK	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>src</i>	<i>src</i>		
BRIG_OPCODE_BITREV	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>src</i>			
BRIG_OPCODE_BITSELECT	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>src</i>	<i>src</i>	<i>src</i>	
BRIG_OPCODE_FIRSTBIT	BRIG_KIND_INST_SOURCE_TYPE	<i>dest</i>	<i>src</i>			
BRIG_OPCODE_LASTBIT	BRIG_KIND_INST_SOURCE_TYPE	<i>dest</i>	<i>src</i>			

dest: must be BRIG_KIND_OPERAND_REGISTER.

src: must be BRIG_KIND_OPERAND_REGISTER, BRIG_KIND_OPERAND_CONSTANT_BYTES, or BRIG_KIND_OPERAND_WAVESIZE.

18.7.1.7 BRIG Syntax for Copy (Move) Instructions

Table 18–10 BRIG Syntax for Copy (Move) Instructions

Opcode	Format	Operand 0	Operand 1
BRIG_OPCODE_COMBINE	BRIG_KIND_INST_SOURCE_TYPE	<i>dest</i>	<i>src-vector</i>
BRIG_OPCODE_EXPAND	BRIG_KIND_INST_SOURCE_TYPE	<i>dest-vector</i>	<i>src</i>
BRIG_OPCODE_LDA	BRIG_KIND_INST_ADDR	<i>dest</i>	<i>address</i>
BRIG_OPCODE_MOV	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>src</i>

dest: must be BRIG_KIND_OPERAND_REGISTER.

src-vector: must be BRIG_KIND_OPERAND_OPERAND_LIST that references a list of BRIG_KIND_OPERAND_REGISTER, BRIG_KIND_OPERAND_CONSTANT_BYTES or BRIG_KIND_OPERAND_WAVESIZE operands.

dest-vector: must be BRIG_KIND_OPERAND_OPERAND_LIST that references a list of BRIG_KIND_OPERAND_REGISTER operands.

src: must be BRIG_KIND_OPERAND_REGISTER, BRIG_KIND_OPERAND_CONSTANT_BYTES, or BRIG_KIND_OPERAND_WAVESIZE.

address: must be BRIG_KIND_OPERAND_ADDRESS.

18.7.1.8 BRIG Syntax for Packed Data Instructions

Table 18–11 BRIG Syntax for Packed Data Instructions

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>	

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
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_REGISTER.

src: must be BRIG_KIND_OPERAND_REGISTER, BRIG_KIND_OPERAND_CONSTANT_BYTES, or BRIG_KIND_OPERAND_WAVESIZE.

number: must be BRIG_KIND_OPERAND_CONSTANT_BYTES.

18.7.1.9 BRIG Syntax for Bit Conditional Move (cmov) Instruction

Table 18–12 BRIG Syntax for Bit Conditional Move (cmov) Instruction

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_REGISTER.

src: must be BRIG_KIND_OPERAND_REGISTER, BRIG_KIND_OPERAND_CONSTANT_BYTES, or BRIG_KIND_OPERAND_WAVESIZE.

18.7.1.10 BRIG Syntax for Floating-Point Arithmetic Instructions

Table 18–13 BRIG Syntax for Floating-Point Arithmetic Instructions

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>
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>		

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
BRIG_OPCODE_SORT	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_REGISTER.

src: must be BRIG_KIND_OPERAND_REGISTER or BRIG_KIND_OPERAND_CONSTANT_BYTES.

18.7.1.11 BRIG Syntax for Floating-Point Optimization Instruction

Table 18–14 BRIG Syntax for Floating-Point Optimization Instruction

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
BRIG_OPCODE_MAD	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>

dest: must be BRIG_KIND_OPERAND_REGISTER.

src: must be BRIG_KIND_OPERAND_REGISTER or BRIG_KIND_OPERAND_CONSTANT_BYTES.

18.7.1.12 BRIG Syntax for Floating-Point Bit Instructions

Table 18–15 BRIG Syntax for Floating-Point Classify (class) Instructions

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>cond</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_REGISTER.

src: must be BRIG_KIND_OPERAND_REGISTER or BRIG_KIND_OPERAND_CONSTANT_BYTES.

cond: must be BRIG_KIND_OPERAND_REGISTER, BRIG_KIND_OPERAND_CONSTANT_BYTES, or BRIG_KIND_OPERAND_WAVESIZE.

18.7.1.13 BRIG Syntax for Native Floating-Point Instructions

Table 18–16 BRIG Syntax for Native Floating-Point Instructions

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
BRIG_OPCODE_NCOS	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>src</i>		
BRIG_OPCODE_NEXP2		<i>dest</i>	<i>src</i>		
BRIG_OPCODE_NFMA		<i>dest</i>	<i>src</i>	<i>src</i>	<i>src</i>
BRIG_OPCODE_NLOG2		<i>dest</i>	<i>src</i>		
BRIG_OPCODE_NRCP		<i>dest</i>	<i>src</i>		
BRIG_OPCODE_NRSQRT		<i>dest</i>	<i>src</i>		
BRIG_OPCODE_NSIN		<i>dest</i>	<i>src</i>		
BRIG_OPCODE_NSQRT		<i>dest</i>	<i>src</i>		

dest: must be BRIG_KIND_OPERAND_REGISTER.

src: must be BRIG_KIND_OPERAND_REGISTER or BRIG_KIND_OPERAND_CONSTANT_BYTES.

18.7.1.14 BRIG Syntax for Multimedia Instructions

Table 18–17 BRIG Syntax for Multimedia Instructions

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3	Operand 4
BRIG_OPCODE_BITALIGN	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>src</i>	<i>src</i>	<i>src</i>	
BRIG_OPCODE_BYTEALIGN	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>src</i>	<i>src</i>	<i>src</i>	
BRIG_OPCODE_LERP	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>src</i>	<i>src</i>	<i>src</i>	
BRIG_OPCODE_PACKCVT	BRIG_KIND_INST_SOURCE_TYPE	<i>dest</i>	<i>src</i>	<i>src</i>	<i>src</i>	<i>src</i>
BRIG_OPCODE_UNPACKCVT	BRIG_KIND_INST_SOURCE_TYPE	<i>dest</i>	<i>src</i>	<i>number</i>		
BRIG_OPCODE_SAD	BRIG_KIND_INST_SOURCE_TYPE	<i>dest</i>	<i>src</i>	<i>src</i>	<i>src</i>	
BRIG_OPCODE_SADHI	BRIG_KIND_INST_SOURCE_TYPE	<i>dest</i>	<i>src</i>	<i>src</i>	<i>src</i>	

dest: must be BRIG_KIND_OPERAND_REGISTER.

src: must be BRIG_KIND_OPERAND_REGISTER, BRIG_KIND_OPERAND_CONSTANT_BYTES, or BRIG_KIND_OPERAND_WAVESIZE.

number: must be BRIG_KIND_OPERAND_CONSTANT_BYTES with value 0, 1, 2, or 3.

18.7.1.15 BRIG Syntax for Segment Checking (segmentp) Instruction

Table 18–18 BRIG Syntax for Segment Checking (segmentp) Instruction

Opcode	Format	Operand 0	Operand 1
BRIG_OPCODE_SEGMENTP	BRIG_KIND_INST_SEG_CVT	<i>dest</i>	<i>src</i>

dest: must be BRIG_KIND_OPERAND_REGISTER.

src: must be BRIG_KIND_OPERAND_REGISTER, BRIG_KIND_OPERAND_CONSTANT_BYTES, or BRIG_KIND_OPERAND_WAVESIZE.

18.7.1.16 BRIG Syntax for Segment Conversion Instructions

Table 18–19 BRIG Syntax for Segment Conversion Instructions

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_REGISTER.

src: must be BRIG_KIND_OPERAND_REGISTER, BRIG_KIND_OPERAND_CONSTANT_BYTES, or BRIG_KIND_OPERAND_WAVESIZE.

18.7.1.17 BRIG Syntax for Compare (cmp) Instruction

Table 18–20 BRIG Syntax for Compare (cmp) Instruction

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_REGISTER.

src: must be BRIG_KIND_OPERAND_REGISTER, BRIG_KIND_OPERAND_CONSTANT_BYTES, 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.

18.7.1.18 BRIG Syntax for Conversion (cvf) Instruction

Table 18–21 BRIG Syntax for Conversion (cvf) Instruction

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_REGISTER.

src: must be BRIG_KIND_OPERAND_REGISTER, BRIG_KIND_OPERAND_CONSTANT_BYTES, or BRIG_KIND_OPERAND_WAVESIZE.

18.7.2 BRIG Syntax for Memory Instructions

Table 18–22 BRIG Syntax for Memory Instructions

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
BRIG_OPCODE_ATOMIC (for atomic_ld)	BRIG_KIND_INST_ATOMIC	<i>dest</i>	<i>address</i>		
BRIG_OPCODE_ATOMIC (for atomic_cas)	BRIG_KIND_INST_ATOMIC	<i>dest</i>	<i>address</i>	<i>src</i>	<i>src</i>
BRIG_OPCODE_ATOMICNORET	BRIG_KIND_INST_ATOMIC	<i>address</i>	<i>src</i>		
BRIG_OPCODE_SIGNAL	BRIG_KIND_INST_SIGNAL	<i>dest</i>	<i>signal</i>	<i>src</i>	
BRIG_OPCODE_SIGNAL (for signal_ld)	BRIG_KIND_INST_SIGNAL	<i>dest</i>	<i>signal</i>		
BRIG_OPCODE_SIGNAL (for signal_cas and signal_waittimeout)	BRIG_KIND_INST_SIGNAL	<i>dest</i>	<i>signal</i>	<i>src</i>	<i>src</i>
BRIG_OPCODE_SIGNALNORET	BRIG_KIND_INST_SIGNAL	<i>signal</i>	<i>src</i>		
BRIG_OPCODE_MEMFENCE	BRIG_KIND_INST_MEMFENCE				

reg-or-vector: must be BRIG_KIND_OPERAND_REGISTER; or BRIG_KIND_OPERAND_OPERAND_LIST that references a list of BRIG_KIND_OPERAND_REGISTER operands.

address: must be BRIG_KIND_OPERAND_ADDRESS.

signal: must be BRIG_KIND_OPERAND_REGISTER.

reg-or-vector: must be BRIG_KIND_OPERAND_REGISTER; BRIG_KIND_OPERAND_CONSTANT_BYTES; BRIG_KIND_OPERAND_WAVESIZE; or BRIG_KIND_OPERAND_OPERAND_LIST that references a list of BRIG_KIND_OPERAND_REGISTER, BRIG_KIND_OPERAND_CONSTANT_BYTES, or BRIG_KIND_OPERAND_WAVESIZE operands.

dest: must be BRIG_KIND_OPERAND_REGISTER.

src: must be BRIG_KIND_OPERAND_REGISTER, BRIG_KIND_OPERAND_CONSTANT_BYTES, or BRIG_KIND_OPERAND_WAVESIZE.

18.7.3 BRIG Syntax for Image Instructions

Table 18–23 BRIG Syntax for Image Instructions

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-or-num</i>
BRIG_OPCODE_LDIMAGE	BRIG_KIND_INST_IMAGE	<i>reg-or-4-vector-reg</i>	<i>image</i>	<i>reg- or- vector-or-num</i>	
BRIG_OPCODE_STIMAGE	BRIG_KIND_INST_IMAGE	<i>reg-or-4-vector-reg</i>	<i>image</i>	<i>reg- or- vector-or-num</i>	
BRIG_OPCODE_QUERYIMAGE	BRIG_KIND_INST_QUERYIMAGE	<i>dest</i>	<i>image</i>		
BRIG_OPCODE_QUERYSAMPLER	BRIG_KIND_INST_QUERYSAMPLER	<i>dest</i>	<i>sampler</i>		
BRIG_OPCODE_IMAGEFENCE	BRIG_KIND_INST_BASIC				

dest: must be BRIG_KIND_OPERAND_REGISTER.

reg-or-4-vector-reg: must be `BRIG_KIND_OPERAND_REGISTER`; or `BRIG_KIND_OPERAND_OPERAND_LIST` that references a list of `BRIG_KIND_OPERAND_REGISTER` operands.

image: must be `BRIG_KIND_OPERAND_REGISTER`.

sampler: must be `BRIG_KIND_OPERAND_REGISTER`.

reg-or-vector-or-num: must be `BRIG_KIND_OPERAND_REGISTER`; `BRIG_KIND_OPERAND_CONSTANT_BYTES`; `BRIG_KIND_OPERAND_WAVESIZE`; or `BRIG_KIND_OPERAND_OPERAND_LIST` that references a list of `BRIG_KIND_OPERAND_REGISTER`, `BRIG_KIND_OPERAND_CONSTANT_BYTES`, or `BRIG_KIND_OPERAND_WAVESIZE` operands.

18.7.4 BRIG Syntax for Branch Instructions

Table 18-24 BRIG Syntax for Branch Instructions

Opcode	Format	Operand 0	Operand 1
<code>BRIG_OPCODE_BR</code>	<code>BRIG_KIND_INST_BR</code>	<i>label</i>	
<code>BRIG_OPCODE_CBR</code>	<code>BRIG_KIND_INST_BR</code>	<i>condition</i>	<i>label</i>
<code>BRIG_OPCODE_SBR</code>	<code>BRIG_KIND_INST_BR</code>	<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_REGISTER` for a `c` register, `BRIG_KIND_OPERAND_CONSTANT_BYTES`, or `BRIG_KIND_OPERAND_WAVESIZE`.

index: must be `BRIG_KIND_OPERAND_REGISTER` for an `s` or `d` register according to the instruction type which must be `u32`, `u64`, `BRIG_KIND_OPERAND_CONSTANT_BYTES`, or `BRIG_KIND_OPERAND_WAVESIZE`.

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 Instructions

Table 18-25 BRIG Syntax for Parallel Synchronization and Communication Instructions

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3	Operand 4
<code>BRIG_OPCODE_BARRIER</code>	<code>BRIG_KIND_INST_BR</code>					
<code>BRIG_OPCODE_WAVEBARRIER</code>	<code>BRIG_KIND_INST_BR</code>					
<code>BRIG_OPCODE_INITFBAR</code>	<code>BRIG_KIND_INST_BASIC</code>	<i>fbarrier-or-reg</i>				
<code>BRIG_OPCODE_JOINFBAR</code>	<code>BRIG_KIND_INST_BR</code>	<i>fbarrier-or-reg</i>				
<code>BRIG_OPCODE_WAITFBAR</code>	<code>BRIG_KIND_INST_BR</code>	<i>fbarrier-or-reg</i>				
<code>BRIG_OPCODE_ARRIVEFBAR</code>	<code>BRIG_KIND_INST_BR</code>	<i>fbarrier-or-reg</i>				
<code>BRIG_OPCODE_LEAVEFBAR</code>	<code>BRIG_KIND_INST_BR</code>	<i>fbarrier-or-reg</i>				
<code>BRIG_OPCODE_RELEASEFBAR</code>	<code>BRIG_KIND_INST_BASIC</code>	<i>fbarrier-or-reg</i>				

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3	Operand 4
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_ACTIVELANEPERMUTE	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_REGISTER; 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_REGISTER.

4-vector-reg: must be BRIG_KIND_OPERAND_OPERAND_LIST that references a list of BRIG_KIND_OPERAND_REGISTER operands.

src: must be BRIG_KIND_OPERAND_REGISTER, BRIG_KIND_OPERAND_CONSTANT_BYTES, or BRIG_KIND_OPERAND_WAVESIZE.

18.7.6 BRIG Syntax for Function Instructions

Table 18–26 BRIG Syntax for Instructions Related to Functions

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
BRIG_OPCODE_CALL	BRIG_KIND_INST_BR	<i>out-args</i>	<i>func</i>	<i>in-args</i>	
BRIG_OPCODE_SCALL	BRIG_KIND_INST_BR	<i>out-args</i>	<i>src</i>	<i>in-args</i>	<i>funcs</i>
BRIG_OPCODE_ICALL	BRIG_KIND_INST_BR	<i>out-args</i>	<i>reg</i>	<i>in-args</i>	<i>signature</i>
BRIG_OPCODE_RET	BRIG_KIND_INST_BASIC				
BRIG_OPCODE_ALLOCA	BRIG_KIND_INST_MEM	<i>dest</i>	<i>src</i>		

dest: must be BRIG_KIND_OPERAND_REGISTER.

src: must be BRIG_KIND_OPERAND_REGISTER, BRIG_KIND_OPERAND_CONSTANT_BYTES, or BRIG_KIND_OPERAND_WAVESIZE.

reg: must be BRIG_KIND_OPERAND_REGISTER.

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.

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 Instructions

18.7.7.1 BRIG Syntax for Kernel Dispatch Packet Instructions

Table 18–27 BRIG Syntax for Kernel Dispatch Packet Instructions

Opcode	Format	Operand 0	Operand 1
BRIG_OPCODE_CURRENTWORKGROUPSIZE	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>dimNumber</i>
BRIG_OPCODE_CURRENTWORKITEMFLATID	BRIG_KIND_INST_BASIC	<i>dest</i>	
BRIG_OPCODE_DIM	BRIG_KIND_INST_BASIC	<i>dest</i>	
BRIG_OPCODE_GRIDGROUPS	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>dimNumber</i>
BRIG_OPCODE_GRIDSIZE	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>dimNumber</i>
BRIG_OPCODE_PACKETCOMPLETIONSIG	BRIG_KIND_INST_BASIC	<i>dest</i>	
BRIG_OPCODE_PACKETID	BRIG_KIND_INST_BASIC	<i>dest</i>	
BRIG_OPCODE_WORKGROUPID	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>dimNumber</i>
BRIG_OPCODE_WORKGROUPSIZE	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>dimNumber</i>
BRIG_OPCODE_WORKITEMABSID	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>dimNumber</i>
BRIG_OPCODE_WORKITEMFLATABSID	BRIG_KIND_INST_BASIC	<i>dest</i>	
BRIG_OPCODE_WORKITEMFLATID	BRIG_KIND_INST_BASIC	<i>dest</i>	
BRIG_OPCODE_WORKITEMID	BRIG_KIND_INST_BASIC	<i>dest</i>	<i>dimNumber</i>

dest: must be BRIG_KIND_OPERAND_REGISTER.

dimNumber: must be BRIG_KIND_OPERAND_CONSTANT_BYTES with the value 0, 1, or 2 corresponding to the X, Y, and Z dimensions respectively.

18.7.7.2 BRIG Syntax for Exception Instructions

Table 18–28 BRIG Syntax for Exception Instructions

Opcode	Format	Operand 0
BRIG_OPCODE_CLEARDETECTEXCEPT	BRIG_KIND_INST_BASIC	<i>exceptionsNumber</i>
BRIG_OPCODE_GETDETECTEXCEPT	BRIG_KIND_INST_BASIC	<i>dest</i>
BRIG_OPCODE_SETDETECTEXCEPT	BRIG_KIND_INST_BASIC	<i>exceptionsNumber</i>

dest: must be BRIG_KIND_OPERAND_REGISTER.

exceptionsNumber: must be BRIG_KIND_OPERAND_CONSTANT_BYTES. The value must be encoded according to BrigExceptionsMask (see 18.3.9. BrigExceptionsMask (page 303)).

18.7.7.3 BRIG Syntax for User Mode Queue Instructions

Table 18–29 BRIG Syntax for User Mode Queue Instructions

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
BRIG_OPCODE_ADDQUEUEWRITEINDEX	BRIG_KIND_INST_QUEUE	<i>dest</i>	<i>address</i>	<i>src</i>	

Opcode	Format	Operand 0	Operand 1	Operand 2	Operand 3
BRIG_OPCODE_CASQUEUEWRITEINDEX	BRIG_KIND_INST_QUEUE	<i>dest</i>	<i>address</i>	<i>src</i>	<i>src</i>
BRIG_OPCODE_LDQUEUEREADINDEX	BRIG_KIND_INST_QUEUE	<i>dest</i>	<i>address</i>		
BRIG_OPCODE_LDQUEUEWRITEINDEX	BRIG_KIND_INST_QUEUE	<i>dest</i>	<i>address</i>		
BRIG_OPCODE_STQUEUEREADINDEX	BRIG_KIND_INST_QUEUE	<i>address</i>	<i>src</i>		
BRIG_OPCODE_STQUEUEWRITEINDEX	BRIG_KIND_INST_QUEUE	<i>address</i>	<i>src</i>		

dest: must be BRIG_KIND_OPERAND_REGISTER.

src: must be BRIG_KIND_OPERAND_REGISTER, BRIG_KIND_OPERAND_CONSTANT_BYTES, or BRIG_KIND_OPERAND_WAVESIZE.

address: must be BRIG_KIND_OPERAND_ADDRESS.

18.7.7.4 BRIG Syntax for Miscellaneous Instructions

Table 18–30 BRIG Syntax for Miscellaneous Instructions

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_REGISTER.

src: must be BRIG_KIND_OPERAND_REGISTER, BRIG_KIND_OPERAND_CONSTANT_BYTES, 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 moduleHeader 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 } ";"
moduleHeader                         ::= "module"
                                   TOKEN_GLOBAL_IDENTIFIER ":"
                                   TOKEN_INTEGER_CONSTANT ":"
                                   TOKEN_INTEGER_CONSTANT ":"
                                   profile ":"
                                   machineModel ":"
                                   defaultFloatRounding ";"

profile                              ::= "$full"
                                   | "$base"

machineModel                         ::= "$small"
                                   | "$large"

defaultFloatRounding                ::= "$default"
                                   | "$zero"
                                   | "$near"

moduleDirective                      ::= extension
extension                           ::= "extension" TOKEN_STRING ";"
moduleStatement                      ::= moduleVariable
                                   | moduleFbarrier
                                   | kernel
                                   | function
                                   | signature

moduleVariable                       ::= optDeclQual linkageQual variable
variable                            ::= optAllocQual optAlignQual optConstQual variableSegment
                                   dataTypeMod TOKEN_GLOBAL_IDENTIFIER optArrayDimension
                                   optInitializer ";"

optInitializer                       ::= [ "=" initializerConstant ]
initializerConstant                  ::= integerConstant
                                   | floatConstant
                                   | typedConstant
                                   | aggregateConstant

aggregateConstant                    ::= "{" { aggregateConstantItem "," } aggregateConstantItem "}"
aggregateConstantItem                ::= typedConstant
                                   | aggregateConstantAlign

aggregateConstantAlign               ::= "align" "(" TOKEN_INTEGER_CONSTANT ")"
typedConstant                        ::= integerTypedConstant
                                   | floatTypedConstant
                                   | packedTypedConstant
                                   | imageTypedConstant
                                   | samplerTypedConstant
                                   | signalTypedConstant
                                   | arrayTypedConstant

integerTypedConstant                 ::= integerType "(" integerConstant ")"
integerConstant                      ::= [ "+" | "-" ] TOKEN_INTEGER_CONSTANT
floatTypedConstant                   ::= "f16" "(" halfConstant ")"
                                   | "f32" "(" singleConstant ")"
                                   | "f64" "(" doubleConstant ")"

floatConstant                        ::= halfConstant
                                   | singleConstant
                                   | doubleConstant

halfConstant                         ::= [ "+" | "-" ] TOKEN_HALF_CONSTANT
singleConstant                       ::= [ "+" | "-" ] TOKEN_SINGLE_CONSTANT
doubleConstant                       ::= [ "+" | "-" ] TOKEN_DOUBLE_CONSTANT

```

```

packedTypedConstant ::= integerPackedType
                      "(" { integerConstant "," } integerConstant ")"
                      | halfPackedType
                      "(" { halfConstant "," } halfConstant ")"
                      | singlePackedType
                      "(" { singleConstant "," } singleConstant ")"
                      | doublePackedType
                      "(" { doubleConstant "," } doubleConstant ")"

imageTypedConstant ::= imageType "(" { imageProperty "," } imageProperty ")"
imageProperty      ::= "geometry" "=" imageGeometry
imageGeometry      ::= "1d"
                      | "2d"
                      | "3d"
                      | "1da"
                      | "2da"
                      | "1db"
                      | "2ddepth"
                      | "2dadepth"

imageChannelType    ::= "snorm_int8"
                      | "snorm_int16"
                      | "unorm_int8"
                      | "unorm_int16"
                      | "unorm_int24"
                      | "unorm_short_555"
                      | "unorm_short_565"
                      | "unorm_int_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"
                      | "srgbx"
                      | "srgba"
                      | "sbgra"
                      | "intensity"
                      | "luminance"
                      | "depth"
                      | "depth_stencil"

samplerTypedConstant ::= samplerType "(" { samplerProperty "," } samplerProperty ")"
samplerProperty      ::= "coord" "=" samplerCoord
                      | "filter" "=" samplerFilter
                      | "addressing" "=" samplerAddressing

```

```

samplerCoord          ::= "normalized"
                        | "unnormalized"
samplerFilter         ::= "nearest"
                        | "linear"
samplerAddressing     ::= "undefined"
                        | "clamp_to_edge"
                        | "clamp_to_border"
                        | "repeat"
                        | "mirrored_repeat"
signalTypedConstant   ::= signalType "(" integerConstant ")"
arrayTypedConstant    ::= integerArrayTypedConstant
                        | halfArrayTypedConstant
                        | singleArrayTypedConstant
                        | doubleArrayTypedConstant
                        | packedArrayTypedConstant
                        | imageArrayTypedConstant
                        | samplerArrayTypedConstant
                        | signalArrayTypedConstant
integerArrayTypedConstant ::= integerType "[" "("
                        { ( integerConstant | integerTypedConstant ) "," }
                        ( integerConstant | integerTypedConstant ) ")"
halfArrayTypedConstant  ::= "f16" "[" "("
                        { ( halfConstant | "f16" "(" halfConstant ")" ) "," }
                        ( halfConstant | "f16" "(" halfConstant ")" ) ")"
singleArrayTypedConstant ::= "f32" "[" "("
                        { ( singleConstant | "f32" "(" singleConstant ")" ) "," }
                        ( singleConstant | "f32" "(" singleConstant ")" ) ")"
doubleArrayTypedConstant ::= "f64" "[" "("
                        { ( doubleConstant | "f64" "(" doubleConstant ")" ) "," }
                        ( doubleConstant | "f64" "(" doubleConstant ")" ) ")"
packedArrayTypedConstant ::= packedType "[" "("
                        { packedTypedConstant "," } packedTypedConstant ")"
imageArrayTypedConstant ::= imageType "[" "("
                        { imageTypedConstant "," } imageTypedConstant ")"
samplerArrayTypedConstant ::= samplerType "[" "("
                        { samplerTypedConstant "," } samplerTypedConstant ")"
signalArrayTypedConstant ::= signalType "[" "("
                        { signalTypedConstant "," } signalTypedConstant ")"
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" dataTypeMod
                        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" dataTypeMod
                        TOKEN_LOCAL_IDENTIFIER optArrayDimension
signature                  ::= "signature" TOKEN_GLOBAL_IDENTIFIER
                        sigOutputFormalArgumentList sigInputFormalArgumentList ";"
sigOutputFormalArgumentList ::= sigFormalArgumentList
sigInputFormalArgumentList  ::= sigFormalArgumentList
sigFormalArgumentList      ::= "(" [ { sigFormalArgument "," } sigFormalArgument ] ")"
sigFormalArgument          ::= optAlignQual "arg" dataTypeMod
                        [ TOKEN_LOCAL_IDENTIFIER ] optArrayDimension
linkageQual                ::= [ "prog" ]
optDeclQual                ::= [ declQual ]

```

```

declQual ::= "decl"
optConstQual ::= [ "const" ]
optAlignQual ::= [ "align" "(" TOKEN_INTEGER_CONSTANT ")" ]
optAllocQual ::= [ "alloc" "(" allocationKind ")" ]
allocationKind ::= "agent"
optArrayDimension ::= [ "[" [ TOKEN_INTEGER_CONSTANT ] "]" ]
codeBlock ::= "{" annotations
              { codeBlockDirective annotations }
              { codeBlockDefinition annotations }
              { codeBlockStatement annotations }
              "}"

codeBlockDirective ::= control
control ::= "enablebreakexceptions" TOKEN_INTEGER_CONSTANT ";"
          | "enabledetectexceptions" TOKEN_INTEGER_CONSTANT ";"
          | "maxdynamicgroupsize" TOKEN_INTEGER_CONSTANT ";"
          | "maxflatgridsize"
            ( TOKEN_INTEGER_CONSTANT | TOKEN_WAVESIZE ) ";"
          | "maxflatworkgroupsize"
            ( TOKEN_INTEGER_CONSTANT | TOKEN_WAVESIZE ) ";"
          | "requireddim" TOKEN_INTEGER_CONSTANT ";"
          | "requiredgridsize"
            ( TOKEN_INTEGER_CONSTANT | TOKEN_WAVESIZE ) ","
            ( TOKEN_INTEGER_CONSTANT | TOKEN_WAVESIZE ) ","
            ( TOKEN_INTEGER_CONSTANT | TOKEN_WAVESIZE ) ";"
          | "requiredworkgroupsize"
            ( TOKEN_INTEGER_CONSTANT | TOKEN_WAVESIZE ) ","
            ( TOKEN_INTEGER_CONSTANT | TOKEN_WAVESIZE ) ","
            ( TOKEN_INTEGER_CONSTANT | TOKEN_WAVESIZE ) ";"
          | "requirepartialworkgroups" ";"

codeBlockDefinition ::= codeBlockVariable
                    | codeBlockFbarrier

codeBlockVariable ::= variable
codeBlockFbarrier ::= fbarrier
codeBlockStatement ::= argBlock
                    | label
                    | instruction

argBlock ::= "{" annotations
           { argBlockDefinition annotations }
           { argBlockStatement annotations }
           "}"

argBlockDefinition ::= argBlockVariable
argBlockVariable ::= variable
argBlockStatement ::= label
                   | instruction
                   | call

label ::= TOKEN_LABEL_IDENTIFIER ":"
instruction ::= 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
| ldimage
| queryimage
| querysampler
| branch
| barrier
| wavebarrier
| fbarrier
| crossLane
| ret
| alloca
| packetcompletionsig
| queue

instruction0 ::= ( "nop"
| "imagefence"
)
";"

instruction1 ::= ( instruction1Opcode optRoundingMod nonOpaqueTypeMod
| "nullptr" optSegmentMod nonOpaqueTypeMod
)
operand ";"

instruction1Opcode ::= "cleardetectexcept"
| "clock"
| "cuid"
| "debugtrap"
| "dim"
| "getdetectexcept"
| "groupbaseptr"
| "kernargbaseptr"
| "laneid"
| "maxcuid"
| "maxwaveid"
| "packetid"
| "setdetectexcept"
| "waveid"
| "workitemflatabsid"
| "workitemflatid"

instruction2 ::= ( instruction2Opcode optRoundingMod optPackingMod
| instruction2OpcodeFtz optFtzMod optPackingMod
| "popcount" nonOpaqueTypeMod
| "firstbit" nonOpaqueTypeMod
| "lastbit" nonOpaqueTypeMod
)
nonOpaqueTypeMod operand "," operand ";"

instruction2Opcode ::= "abs"
| "bitrev"
| "currentworkgroupsize"
| "currentworkitemflatid"
| "fract"
| "ncos"
| "neg"
| "nexp2"
| "nlog2"
| "nrcp"
| "nrsqrt"

```

```

| "nsin"
| "nsqrt"
| "gridgroups"
| "gridsize"
| "not"
| "sqrt"
| "workgroupid"
| "workgroupsize"
| "workitemabsid"
| "workitemid"
instruction2OpcodeFtz ::= "ceil"
| "floor"
| "rint"
| "trunc"
instruction3 ::= ( instruction3Opcode optRoundingMod optPackingMod
| instruction3OpcodeFtz optFtzMod optPackingMod
| "class" nonOpaqueTypeMod
)
nonOpaqueTypeMod 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 optFtzMod optPackingMod
)
nonOpaqueTypeMod operand "," operand "," operand "," operand ","
instruction4Opcode ::= "fma"
| "mad"
| "bitextract"
| "bitselect"
| "shuffle"
| "cmov"
| "bitalign"
| "bytealign"
| "lerp"
instruction4OpcodeFtz ::= "nfma"
mul ::= ( "mul" optRoundingMod optPackingMod nonOpaqueTypeMod
| "mulhi" optPackingMod nonOpaqueTypeMod
| "mul24hi" nonOpaqueTypeMod
| "mul24" nonOpaqueTypeMod
| "mad24" nonOpaqueTypeMod operand ","
| "mad24hi" nonOpaqueTypeMod operand ","
) operand "," operand "," operand ","
bitinsert ::= "bitinsert" nonOpaqueTypeMod operand "," operand ","
operand "," operand "," operand ","
combine ::= "combine" vectorMod nonOpaqueTypeMod nonOpaqueTypeMod
operand "," vectorOperand ";"
expand ::= "expand" vectorMod nonOpaqueTypeMod nonOpaqueTypeMod
vectorOperand "," operand ";"
lda ::= "lda" optSegmentMod nonOpaqueTypeMod operand ","

```

```

memoryOperand ::= "memoryOperand ";
mov            ::= "mov" dataTypeMod operand "," operand ";";
pack          ::= "pack" nonOpaqueTypeMod nonOpaqueTypeMod operand ","
operand "," operand "," operand ";";
unpack        ::= "unpack" nonOpaqueTypeMod nonOpaqueTypeMod operand ","
operand "," operand ";";
packcvt       ::= "packcvt" nonOpaqueTypeMod nonOpaqueTypeMod operand ","
operand "," operand "," operand "," operand ";";
unpackcvt     ::= "unpackcvt" nonOpaqueTypeMod nonOpaqueTypeMod operand ","
operand "," operand ";";
sad           ::= ( "sad" | "sadhi" ) nonOpaqueTypeMod nonOpaqueTypeMod
operand "," operand "," operand "," operand ";";
segmentConversion ::= ( "segmenttp" | "ftos" | "stof" )
segmentMod optNullMod nonOpaqueTypeMod nonOpaqueTypeMod
operand "," operand ";";
cmp           ::= "cmp" comparisonOp optFtzMod optPackingMod nonOpaqueTypeMod
nonOpaqueTypeMod 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"
| "_sltu"
| "_sleu"
| "_sgtu"
| "_sgeu";
cvt           ::= "cvt" optCvtRoundingMod nonOpaqueTypeMod
nonOpaqueTypeMod operand "," operand ";";
optCvtRoundingMod ::= [ cvtRoundingMod ];
cvtRoundingMod ::= floatRoundingMod
| "_ftz"
| "_ftz" floatRoundingMod
| intRoundingMod
| "_ftz" intRoundingMod;
ld           ::= "ld" optVectorMod optSegmentMod
optAlignMod optConstMod optEquivMod optWidthMod dataTypeMod
possibleVectorOperand "," memoryOperand ";";
st           ::= "st" optVectorMod optSegmentMod
optAlignMod optEquivMod dataTypeMod
possibleVectorOperand "," memoryOperand ";";
atomic       ::= "atomic"
( atomicOp
optSegmentMod memOrderMod memScopeMod optEquivMod
nonOpaqueTypeMod operand "," memoryOperand "," operand
| "_ld"

```

```

                                optSegmentMod ldMemOrderMod memScopeMod optEquivMod
                                nonOpaqueTypeMod operand "," memoryOperand
                                | "_cas"
                                | optSegmentMod memOrderMod memScopeMod optEquivMod
                                | nonOpaqueTypeMod
                                | operand "," memoryOperand "," operand "," operand
                                | ")" ";"
atomicnoret ::= "atomicnoret"
              ( atomicOp
                | optSegmentMod memOrderMod memScopeMod optEquivMod
                | nonOpaqueTypeMod memoryOperand "," operand
                | "_st"
                | optSegmentMod stMemOrderMod memScopeMod optEquivMod
                | nonOpaqueTypeMod memoryOperand "," operand
                | ")" ";"
atomicOp ::= "_add"
           | "_and"
           | "_exch"
           | "_max"
           | "_min"
           | "_or"
           | "_sub"
           | "_wrapdec"
           | "_wrapinc"
           | "_xor"
signal ::= "signal"
         ( signalOp
           | memOrderMod nonOpaqueTypeMod signalTypeMod
           | operand "," operand "," operand
           | "_ld"
           | ldMemOrderMod nonOpaqueTypeMod signalTypeMod
           | operand "," operand
           | "_cas"
           | memOrderMod nonOpaqueTypeMod signalTypeMod
           | operand "," operand "," operand "," operand
           | "_wait"
           | waitOp memOrderMod nonOpaqueTypeMod signalTypeMod
           | operand "," operand "," operand
           | "_waittimeout"
           | waitOp memOrderMod nonOpaqueTypeMod signalTypeMod
           | operand "," operand "," operand "," operand
           | ")" ";"
signalnoret ::= "signalnoret"
             ( signalOp
               | memOrderMod nonOpaqueTypeMod signalTypeMod
               | operand "," operand
               | "_st"
               | stMemOrderMod nonOpaqueTypeMod signalTypeMod
               | operand "," operand
               | ")" ";"
signalOp ::= "_add"
           | "_and"
           | "_exch"
           | "_or"
           | "_sub"
           | "_xor"
waitOp ::= "_eq"
         | "_ne"
         | "_lt"
         | "_gte"
memfence ::= "memfence" fenceMemOrderMod memScopeMod ";"
memOrderMod ::= "_scacq" | "_screl" | "_scar" | "_rlx"
stMemOrderMod ::= "_screl" | "_rlx"
ldMemOrderMod ::= "_scacq" | "_rlx"

```

```

fenceMemOrderMod      ::= "_scacq" | "_screl" | "_scar"
memScopeMod           ::= "_wave" | "_wg" | "_agent" | "_system"
rdimage                ::= "rdimage" [ "_v4" ] geometryMod optEquivMod
                        nonOpaqueTypeMod imageTypeMod nonOpaqueTypeMod
                        possibleVectorOperand "," operand "," operand ","
                        possibleVectorOperand ";";
ldimage                ::= "ldimage" [ "_v4" ] geometryMod optEquivMod
                        nonOpaqueTypeMod imageTypeMod nonOpaqueTypeMod
                        possibleVectorOperand "," operand ","
                        possibleVectorOperand ";";
stimage                ::= "stimage" [ "_v4" ] geometryMod optEquivMod
                        nonOpaqueTypeMod imageTypeMod nonOpaqueTypeMod
                        possibleVectorOperand "," operand ","
                        possibleVectorOperand ";";
geometryMod            ::= "_1d"
                        | "_2d"
                        | "_3d"
                        | "_1da"
                        | "_2da"
                        | "_1db"
                        | "_2ddepth"
                        | "_2dadepth";
queryimage             ::= "queryimage" geometryMod queryimageOp nonOpaqueTypeMod
                        imageTypeMod operand "," operand ";";
queryimageOp           ::= "_width"
                        | "_height"
                        | "_depth"
                        | "_array"
                        | "_channelorder"
                        | "_channeltype";
querysampler           ::= "querysampler" querysamplerOp nonOpaqueTypeMod
                        operand "," operand ";";
querysamplerOp         ::= "_coord"
                        | "_filter"
                        | "_addressing";
branch                 ::= "br" TOKEN_LABEL_IDENTIFIER ";";
                        | "cbr" optWidthMod nonOpaqueTypeMod
                        TOKEN_CREGISTER "," TOKEN_LABEL_IDENTIFIER ";";
                        | "sbr" optWidthMod nonOpaqueTypeMod 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" nonOpaqueTypeMod
                        operand "," nonRegisterIdentifier ";";
crossLane              ::= "activelaneid" optWidthMod nonOpaqueTypeMod operand ";";
                        | "activelanecount" optWidthMod nonOpaqueTypeMod
                        nonOpaqueTypeMod operand "," operand ";";
                        | "activelanemask" "_v4" optWidthMod nonOpaqueTypeMod
                        nonOpaqueTypeMod vectorOperand "," operand ";";
                        | "activelanepermute" optWidthMod nonOpaqueTypeMod
                        operand "," operand "," operand "," operand "," operand
                        ";";
call                   ::= "call" TOKEN_GLOBAL_IDENTIFIER
                        callOutputActualArguments callInputActualArguments ";";
                        | "scall" optWidthMod nonOpaqueTypeMod operand
                        callOutputActualArguments callInputActualArguments
                        callTargets ";";
                        | "icall" optWidthMod nonOpaqueTypeMod operand

```

	callOutputActualArguments callInputActualArguments TOKEN_GLOBAL_IDENTIFIER ";"
callOutputActualArguments	::= callActualArguments
callInputActualArguments	::= callActualArguments
callActualArguments	::= "(" [operandList] ")"
callTargets	::= "[" { TOKEN_GLOBAL_IDENTIFIER "," } TOKEN_GLOBAL_IDENTIFIER "]"
ret	::= "ret" ";"
alloca	::= "alloca" optAlignMod nonOpaqueTypeMod operand "," operand ";"
packetcompletionSIG	::= "packetcompletionSIG" signalTypeMod operand ";"
queue	::= "addqueuewriteindex" segmentMod memOrderMod nonOpaqueTypeMod operand "," memoryOperand "," operand ";" "casqueuewriteindex" segmentMod memOrderMod nonOpaqueTypeMod operand "," memoryOperand "," operand "," operand ";" ("ldqueueuereadindex" "ldqueueuereadindex") segmentMod memOrderMod nonOpaqueTypeMod operand "," memoryOperand ";" ("stqueueuereadindex" "stqueueuereadindex") segmentMod memOrderMod nonOpaqueTypeMod memoryOperand "," operand ";"
pragmaOperand	::= TOKEN_STRING immediateOperand aggregateConstant identifierOperand TOKEN_LABEL_IDENTIFIER
operand	::= immediateOperand identifierOperand
immediateOperand	::= integerConstant floatConstant typedConstant 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 ::= optFtzMod [ floatRoundingMod ]
optFtzMod      ::= [ "_ftz" ]
floatRoundingMod ::= "_up"
                  | "_down"
                  | "_zero"
                  | "_near"

intRoundingMod ::= "_upi"
                | "_downi"
                | "_zeroi"
                | "_neari"
                | "_upi_sat"
                | "_downi_sat"
                | "_zeroi_sat"
                | "_neari_sat"
                | "_supi"
                | "_sdowni"
                | "_szeroi"
                | "_sneari"
                | "_supi_sat"
                | "_sdowni_sat"
                | "_szeroi_sat"
                | "_sneari_sat"

optPackingMod ::= [ packingMod ]
packingMod    ::= "_pp"
                | "_ps"
                | "_sp"
                | "_ss"
                | "_s"
                | "_p"
                | "_pp_sat"
                | "_ps_sat"
                | "_sp_sat"
                | "_ss_sat"
                | "_s_sat"
                | "_p_sat"

dataTypeMod   ::= "_" dataType
dataType      ::= baseType
                | packedType
                | opaqueType

nonOpaqueTypeMod ::= "_" nonOpaqueType
nonOpaqueType  ::= baseType
                | packedType

baseType       ::= integerType
                | floatType
                | bitType

integerType    ::= "u8"

```

```

| "s8"
| "u16"
| "s16"
| "u32"
| "s32"
| "u64"
| "s64"
bitType ::= "b1"
| "b8"
| "b16"
| "b32"
| "b64"
| "b128"
floatType ::= "f16"
| "f32"
| "f64"
packedType ::= integerPackedType
| halfPackedType
| singlePackedType
| doublePackedType
integerPackedType ::= "u8x4"
| "s8x4"
| "u16x2"
| "s16x2"
| "u8x8"
| "s8x8"
| "u16x4"
| "s16x4"
| "u32x2"
| "s32x2"
| "u8x16"
| "s8x16"
| "u16x8"
| "s16x8"
| "u32x4"
| "s32x4"
| "u64x2"
| "s64x2"
halfPackedType ::= "f16x2"
| "f16x4"
| "f16x8"
singlePackedType ::= "f32x2"
| "f32x4"
doublePackedType ::= "f64x2"
opaqueType ::= imageType
| samplerType
| signalType
imageTypeMod ::= " " imageType
imageType ::= "roimg"
| "woimg"
| "rwimg"
samplerTypeMod ::= " " samplerType
samplerType ::= "samp"
signalTypeMod ::= " " signalType
signalType ::= "sig32"
| "sig64"

```

APPENDIX A.

Limits

This appendix lists the maximum or minimum values that HSA implementations must support:

- **c registers:** The *c* registers in HSAIL are a single pool of resources per function scope. It is an error if the value $(c_{\max} + 1)$ exceeds 128 for any kernel or function definition, where c_{\max} is the highest *c* register number in the kernel or function code block, or -1 if no *c* registers are used. For example, if a function code block only uses registers $\$c0$ and $\$c7$, then c_{\max} is 7 not 2.
- **s, d, and q registers:** 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 2048 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.
- **Equivalence classes:** Every implementation must support exactly 256 classes.
- **Identifiers:** Every HSAIL implementation must support identifiers with names whose size ranges from 1 to 1024 characters. Implementations are allowed to support longer names.
- **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 \(fbarrier\) Instructions \(page 230\)](#)) and the exception detection operations (see [11.2. Exception Instructions \(page 260\)](#)).
- **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 \(page 114\)](#)).
- 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 \(page 244\)](#)).
- Image data type support: Every agent that supports images must support images as defined in [Chapter 7. Image Instructions \(page 194\)](#) 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

An atomic memory operation that specifies an acquire memory ordering (an `ld_acq`, `atomic_ar`, or `atomicnoreset_ar` instruction).

active work-group

A work-group executing in a compute unit.

active work-item

A work-item in an active work-group. At an instruction, an active work-item is one that executes the current instruction.

agent

A hardware or software component that participates in the HSA memory model. An agent can submit AQL packets for execution. An agent may also, but is not required, to be a kernel agent. It is possible for a system to include agents that are neither kernel agents nor host CPUs. See [1.1. What Is HSAIL? \(page 20\)](#).

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 HSA executables into which zero or more HSA code objects have been loaded for zero or more kernel agents. See [4.2. Program, Code Object, and Executable \(page 48\)](#).

Architected Queuing Language (AQL)

An AQL packet is an HSA-standard packet format. AQL kernel dispatch packets are used to dispatch kernels on the kernel agent and specify the launch dimensions, instruction code, kernel arguments, completion detection, and more. Other AQL packets control aspects of a kernel agent such as when to execute AQL packets and making the results of memory instructions visible. AQL packets are queued on User Mode Queues. See *HSA Platform System Architecture Specification Version 1.0* section 2.9 *Requirement: Architected Queuing Language (AQL)*.

arg segment

A memory segment used to pass arguments into and out of functions. See [2.8.1. Types of Segments \(page 31\)](#) and [10.2. Function Call Argument Passing \(page 244\)](#).

BRIG

The HSAIL binary format. See [Chapter 18. BRIG: HSAIL Binary Format \(page 298\)](#).

call convention

Each kernel agent can support one or more call conventions. For example, a kernel agent may have different call conventions that each use a different number of hardware registers to allow different numbers of wavefronts to execute on a compute unit. See [4.2.1. Finalization \(page 49\)](#).

compound type

A type made up of a base data type and a length. See [4.13.1. Base Data Types \(page 99\)](#).

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. A kernel agent is composed of one or more compute units. See [2.1. Overview of Grids, Work-Groups, and Work-Items \(page 23\)](#).

current work-item flattened ID

The current work-item ID flattened into one dimension. Uses current work-group size and so differs for partial work-groups than work-item flattened ID. See [2.3.2. Work-Item Flattened ID and Current Work-Item Flattened ID \(page 27\)](#).

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 \(page 23\)](#).

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 \(page 41\)](#).

fbarrier

A fine-grain barrier that applies to a subset of a work-group. See [9.2. Fine-Grain Barrier \(fbarrier\) Instructions \(page 230\)](#).

finalizer

A finalizer is part of the HSA runtime and translates HSAIL code in the form of BRIG into an HSA code object that contains the appropriate native machine code for a kernel agent that is part of an HSA system. When an application uses the HSA runtime it can optionally include the finalizer.

finalizer extension

An operation specific to a finalizer. Finalizer extensions are specified in the `extension` directive and accessed like all HSAIL instructions. See [13.1. extension Directive \(page 274\)](#).

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 \(page 27\)](#).

global segment

A memory segment in which memory is visible to all work-items in all kernel agents and to all host CPUs. See [2.8.1. Types of Segments \(page 31\)](#).

grid

A multidimensional, rectangular structure containing work-groups. A grid is formed when a program launches a kernel. See [1.2. HSAIL Virtual Language \(page 21\)](#).

group segment

A memory segment in which memory is visible to a single work-group. See [2.8.1. Types of Segments \(page 31\)](#).

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 a kernel agent using memory instructions to construct and enqueue AQL packets. In some systems, a host CPU can also act as a kernel agent (with appropriate HSAIL finalizer and AQL mechanisms). See [1.1. What Is HSAIL? \(page 20\)](#).

HSA code object

An HSAIL program can be finalized to produce an HSA code object for a specific instruction set architecture. An HSA code object can then be loaded into an HSA executable for a specific kernel agent that supports the instruction set architecture of the HSA code object. The kernels in the HSA executable can then be executed on the kernel agent on which they have been loaded. See [4.2. Program, Code Object, and Executable \(page 48\)](#).

HSA executable

An HSA executable manages the allocation of global and readonly segment variables defined by HSA code objects. It also manages linking global and readonly segment variable declarations to external definitions outside the HSA executable, such as in the host application. An application can use the HSA runtime to create zero or more HSA executables, to which zero or more HSA code objects can be loaded for different kernel agents. See [4.2. Program, Code Object, and Executable \(page 48\)](#).

HSA implementation

A combination of one or more host CPU agents able to execute the HSA runtime, one or more kernel agents 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: support for User Mode Queues, signals and memory management; optional support for images and samplers; a finalizer; and a loader. See the *HSA Runtime Programmer's Reference Manual*.

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.

HSAIL module

The unit of HSAIL generation. A single module can contain multiple declarations and definitions. It can be added to one or more HSAIL programs. See [4.2. Program, Code Object, and Executable \(page 48\)](#) and [4.3. Module \(page 53\)](#).

HSAIL program

The unit of HSAIL linkage. An application can use the HSA runtime to create zero or more HSAIL programs, and add zero or more HSAIL modules to a program. The program linkage names of a module are linked with the program linkage names in the other modules in the same program. For each program, the modules 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. The exception is that global and readonly segment variables may be declared only, in which case the HSA executable must be used to provide the definition, such as to a host application variable. See [4.2. Program, Code Object, and Executable \(page 48\)](#) and [4.3. Module \(page 53\)](#).

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 \(page 211\)](#).

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

An invalid address is a location in application global memory where an access from a kernel agent or other agent is violating system software policy established by the setup of the system page table attributes. If a kernel agent accesses an invalid address, system software shall be notified. See *HSA Platform System Architecture Specification Version 1.0* section 2.1 *Requirement: Shared Virtual Memory* and section 2.9.3 *Error handling*.

kernarg segment

A memory segment used to pass arguments into a kernel. See [2.8.1. Types of Segments \(page 31\)](#).

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? \(page 20\)](#).

kernel agent

An agent that supports the HSAIL instruction set and supports execution of AQL kernel dispatch packets. As an agent, a kernel agent can dispatch commands to any kernel agent (including itself) using memory instructions to construct and enqueue AQL packets. A kernel agent is composed of one or more compute units. See [1.1. What Is HSAIL? \(page 20\)](#).

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 \(page 29\)](#).

loader

A loader is part of the HSA runtime and can load HSA code objects onto a kernel agent that is part of the HSA system. In addition, it can provide the information required for the application to create AQL kernel dispatch packets that can execute the kernels contained in the loaded HSA code object.

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 \(page 98\)](#).

NaN

Not A Number. A class of floating-point values defined by IEEE/ANSI Standard 754-2008. Used to indicate that a value is not a valid floating-point number. Can either be a quiet NaN or a signaling NaN. See [4.19.4. Not A Number \(NaN\) \(page 111\)](#).

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 \(page 69\)](#).

packet ID

Each AQL packet has a 64-bit packet ID unique to the User Mode Queue on which it is enqueued. The packet ID is assigned as a monotonically increasing sequential number of the logical packet slot allocated in the User Mode queue. The combination of the packet ID and the queue ID is unique for a process.

packet processor

Packet processors are tightly bound to one or more agents, and provide the functionality to process AQL packets enqueued on User Mode Queues of those agents. The packet processor function may be performed by the same or by a different agent to the one with which the User Mode Queue is associated that will execute the kernel dispatch packet or agent dispatch packet function.

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 \(page 31\)](#).

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. Global and readonly segment variables with program linkage may also be linked to definitions outside the HSAIL program using the HSA executable. See [4.12.1. Program Linkage \(page 97\)](#).

queue ID

An identifier for a User Mode Queue in a process. Each queue ID is unique in the process. The combination of the queue ID and the packet ID is unique for a process.

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 \(page 31\)](#).

release synchronizing operation

A memory instruction marked with release (an `st_rel`, `atomic_ar`, or `atomicnoreset_ar` instruction).

sampler handle

An opaque handle to a sampler which specifies how coordinates are processed by an `rdimage` image instruction. See [7.1.8. Sampler Creation and Sampler Handles \(page 214\)](#).

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 \(page 31\)](#).

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 \(signal\) Instructions \(page 187\)](#).

spill segment

A memory segment used to load or store register spills. See [2.8.1. Types of Segments \(page 31\)](#).

ULP

Used to specify the precision of a floating-point operation. See [4.19.6. Unit of Least Precision \(ULP\) \(page 112\)](#).

uniform operation

An instruction 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 \(page 41\)](#).

User Mode Queue

A User Mode Queue is a memory data structure created by the HSA runtime on which AQL packets can be enqueued. The packets are processed by the packet processor associated with the User Mode Queue. For example, a User Mode Queue associated with the packet processor of a kernel agent can be used to execute kernels on that kernel agent. See *HSA Platform System Architecture Specification Version 1.0* section 2.8 *Requirement: User Mode Queuing*.

wavefront

A group of work-items executing on a single instruction pointer. See [2.6. Wavefronts, Lanes, and Wavefront Sizes \(page 29\)](#).

WAVESIZE

An implementation defined constant specifying the size of a wavefront for a kernel agent. See [2.6.2. Wavefront Size \(page 30\)](#) and [2.6. Wavefronts, Lanes, and Wavefront Sizes \(page 29\)](#).

work-group

A collection of work-items from the same kernel dispatch. See [2.2. Work-Groups \(page 25\)](#).

work-group ID

The identifier of a work-group expressed in three dimensions. See [2.2.1. Work-Group ID \(page 25\)](#).

work-group flattened ID

The work-group ID flattened into one dimension. See [2.2.2. Work-Group Flattened ID \(page 26\)](#).

work-item

The simplest element of work. Another name for a unit of execution in a kernel dispatch. See [2.3. Work-Items \(page 26\)](#).

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 \(page 27\)](#).

work-item flattened ID

The work-item ID flattened into one dimension. Uses work-group size and so differs for partial work-groups than current work-item flattened ID. See [2.3.2. Work-Item Flattened ID and Current Work-Item Flattened ID \(page 27\)](#).

work-item flattened absolute ID

The work-item absolute ID flattened into one dimension. See [2.3.4. Work-Item Flattened Absolute ID \(page 27\)](#).

work-item ID

The identifier of a work-item (within the work-group) expressed in three dimensions. See [2.3.1. Work-Item ID \(page 26\)](#).

write atomicity

A condition of a store such that it must be written in its entirety.

Index

2

24-bit integer optimization instructions 122, 349
 mad24 122, 367
 mad24hi 123, 367
 mul24 123, 280-282, 367
 mul24hi 123, 367

A

acquire memory order 376
 active work-group 28, 376
 active work-item 28, 43-44, 175, 376
 addressing mode 199, 206-211, 214-216, 346
 agent 20-21, 31-35, 38-39, 70-71, 95, 114, 167, 169-171, 173, 180, 187-188, 195-196, 211-215, 217, 253, 263-264, 267-268, 365, 375-376, 378, 380
 aggregate constant 91-92
 application program 376
 Architected Queuing Language 20, 24, 376, 378, 380
 arg segment 34, 66, 74, 78-79, 97-99, 132, 166, 212, 214, 217, 244-248, 292, 375-376
 argument scope 74, 79, 99, 243-245, 256, 328, 375
 argument scope arg segment 52
 arguments 296
 arithmetic instructions 109, 116
 atomic memory instruction 170-173, 180-181, 185, 265, 307

B

base data type 99, 101, 377
 Base profile 85, 100, 103, 108, 110-111, 130, 133, 142, 144, 162, 165, 173, 177-178, 284, 288-289
 Base profile 131
 bit conditional move instruction 140, 351
 cmov 140, 351
 bit string instructions 127
 bitextract 127
 bitinsert 128
 bitmask 128
 bitrev 128
 bitselect 128
 firstbit 128-129
 lastbit 129
 bits per pixel 198, 204, 211
 branch instructions 41, 44-45, 94, 104, 132, 227-228, 370
 brn 227
 cbr 41, 44-45, 227, 370
 sbr 41, 44, 94, 132, 228, 370
 BREAK 112, 271, 273, 279, 296-297

BRIG binary format 48, 92, 198, 275, 298, 300, 313, 317-318, 321, 348, 358, 376

C

call convention 377
 call instruction 217
 channel order 195, 198-200, 204-205, 208-209, 344
 channel type 195, 198, 200-201, 204-205, 212, 216, 345
 clock special instruction 43, 265
 code object. HSA 378
 compare instructions 80, 155, 289, 332, 365, 368
 cmp 133, 155, 289, 365, 368
 compile-time macro 30, 267
 compound type 99-100, 105, 126, 134-136, 173, 234, 314, 377
 compute 296
 compute unit 24-25, 39, 230, 267-268, 271, 374, 376-377, 379-380
 control 365
 control (c) register 80, 125, 147
 control directive 24, 61-62, 240, 262, 271, 278-283
 enablebreakexceptions 279
 enabledetectexceptions 271, 279-280, 365
 maxdynamicgroupsize 280
 maxflatgridsize 280
 maxflatworkgroupsize 281
 requireddim 281
 requiredgridsize 282
 requiredworkgroupsize 282
 requirepartialworkgroups 283
 control flow 227, 245
 control flow divergence 228, 252, 254, 293
 conversion instructions 100, 106, 159-160, 173, 176, 179, 270, 289-291, 332-333, 365-366, 368
 cvt 106, 111, 159, 173, 176, 179, 270, 289-291, 368
 coordinate normalization mode 206, 211, 214
 copy instructions 34, 71-74, 104, 115, 119, 130, 132, 160, 188, 217, 247, 275, 365, 367-368
 combine instruction 131
 expand instruction 131
 lda 34, 71-74, 104, 115, 132, 188, 217, 247, 365, 367
 mov 132, 160, 188, 217, 275, 365, 368
 current work-item flattened ID 27, 377
 currentworkgroupsize special instruction 258

D

debugtrap special instruction 266, 272
 declaration 58
 default_float_rounding 285

DETECT 108, 112, 261-262, 271, 279-280, 297
 dimension 24-27, 29, 64-65, 194, 196-197, 199, 206-207, 211, 222-223, 245, 257-260, 281-283, 328, 346, 358, 374, 377, 382
 directive 30, 55, 61-63, 82, 101, 171, 194, 240, 262, 274-283, 298, 303, 305, 320-321, 325-326, 328, 339, 342-343, 347, 356-358, 377
 extension directive 55, 63, 101, 171, 194, 274-275, 377
 dispatch 24-25, 32-35, 38, 114, 171-172, 212, 218-219, 262, 376-378, 380
 divergent control flow 41-42, 44, 293, 377

E

exception instructions 260-261, 271, 297, 366
 cleardetectexcept 261
 getdetectexcept 261, 271, 297, 366
 setdetectexcept 261, 271, 366
 exceptions 108, 110-112, 143, 165, 260-261, 269-273, 279-280, 289-291, 295-297
 executable, HSA 378
 extension directive 274

F

filter mode 206-207, 209-211, 214, 216, 218, 346
 finalization 49
 finalizer 50, 63, 168-169, 172, 228, 243, 254, 262, 271, 278, 280-283, 296-297, 377-378
 finalizer extension 63, 377
 fine-grain barrier 25, 43, 61, 63, 68, 97, 230-236, 321, 324-325, 357, 364-366, 377
 flattened absolute ID 27, 377, 382
 floating-point arithmetic instructions 29, 116, 118-120, 140, 142-144, 183, 202-203, 270, 289-290, 292, 365, 367
 add instruction 142, 270
 ceil 142
 div 118, 142, 270, 292, 367
 floor 142
 fma 142-143, 270, 289-290, 367
 fract 143
 max 116, 119, 143, 183, 202-203, 367
 min 119, 143, 183, 202-203, 367
 mul 119, 143, 270, 365, 367
 rint 143
 sqrt 144, 270
 sub 120, 144, 183, 270, 367
 trunc 144, 367
 floating-point bit instructions 111, 117-119, 146-147, 168-169, 252, 289, 331, 333-334, 366-367
 abs 147
 class 147, 168-169, 252, 331, 333-334, 367
 copysign 147
 neg 147
 floating-point optimization instruction 144
 ftz modifier 110-111, 146, 148, 156, 162, 165, 270, 290-291

Full profile 111, 142, 144, 284, 288-290
 function 34, 41, 52, 58-60, 62-63, 70, 72-74, 78-80, 96-99, 217, 243-255, 277-278, 282-283, 304, 320-321, 323-324, 328, 356, 362, 364
 declaration 98
 definition 98-99
 function declaration 58-59, 70, 74, 79, 97, 247
 function definition 58-59, 70, 74, 244-247, 323-324
 indirect function 52, 248, 250
 function instructions 41, 44, 46-47, 50, 60, 62-63, 65, 68, 74, 217, 243-256, 275, 330, 366, 370-371
 alloca 245, 255-256, 366, 371
 call instruction 62-63, 243-245, 246-248, 250, 252
 icall 41, 44, 60, 248, 253-254, 370
 ret 46-47, 74, 243-244, 247-249, 254-255, 275, 330, 366
 scall 41, 44, 50, 65, 68, 250-253, 370
 function signature 55, 60, 63, 70, 73-74, 78-79, 99, 104, 114, 246, 248, 254, 275, 322-324, 357-358, 364

G

global segment 22, 31-33, 35, 39, 70-71, 94-95, 97, 107, 114-115, 167, 170-172, 180, 188-190, 197, 213-215, 219, 253, 263-264, 268, 294, 328, 335, 377
 grid 21, 24-25, 27-28, 72-73, 167, 258-259, 282, 378, 382
 group segment 22, 24, 26, 32, 35-36, 38-39, 52, 73, 96-97, 107, 112-113, 132, 167, 170, 180, 230, 234, 237, 253, 261, 266, 280, 286-287, 335, 374, 378

H

hardware registers 34, 39, 50, 52, 73-74
 host CPU 20-21, 32-33, 38-39, 171, 253, 267, 285, 376-378
 HSA code object 378
 HSA executable 378
 HSA implementation 20-21, 23-24, 28, 38-41, 43-44, 63, 71-74, 78, 110-112, 114-115, 122, 148, 170, 172-173, 175, 179-180, 187-188, 190-191, 195, 197, 202, 204, 206-207, 211-214, 228-230, 234, 240, 244, 250, 252, 256, 261-262, 265-266, 269-270, 277, 284, 288-290, 292-293, 314, 318, 374-375, 378
 HSA runtime 268, 378
 HSAIL 378
 HSAIL module 379
 HSAIL program 379

I

illegal instruction 379
 image 82, 101, 170, 194-199, 204, 208-209, 211-214, 216-218, 220, 274-275, 294, 304-305, 333, 336, 344-345
 image access permission 211-212, 217
 image coordinates 195, 206, 209, 219-223
 image data 195-200, 211-213, 216-221, 223, 335, 375, 379
 image element 195, 197-198, 201, 206-209, 211, 218-219, 223

- image geometry 195-196, 204-206, 209, 211-213, 216, 333, 336
- image handler 217
- image memory model 169, 196, 218
- image sampler 206-207, 215-218, 220, 222, 346, 355-356, 375
- image segment 171, 218-219, 335
- image size 82, 194-195, 197, 211-212
- pixel 151, 198
- texel 209
- image format 195, 198, 204, 211-213, 223
- image instructions 112, 169, 194-198, 201-204, 206-207, 209, 211-212, 216-221, 223, 274, 294, 302, 333, 335, 355
 - imagefence 225-226
 - ldimage 169, 206, 210, 212, 216-217, 221-222, 275, 366, 370
 - queryimage 217, 224, 275, 305, 336, 366, 370
 - querysampler 217, 224-225, 275, 313, 336, 366, 370
 - rdimage 195, 197, 206-207, 210, 214, 216-220, 222, 275, 366, 370, 381
 - stimage 169, 206, 209-210, 216-217, 222-223, 275, 366, 370
- imagefence 275
- indirect function descriptor 41
- individual bit instructions 125-126
 - and instruction 125
 - not instruction 126
 - or instruction 125
 - xor instruction 125
- integer arithmetic instructions 116, 118-120
 - abs instruction 118
 - add instruction 118
 - borrow instruction 118
 - carry instruction 118
 - div instruction 118
 - max instruction 116, 119
 - min instruction 119
 - mul instruction 119
 - mulhi instruction 119
 - neg instruction 119
 - rem instruction 119
 - sub instruction 120
- integer optimization instruction 121-122
 - mad 121, 367
- integer shift instructions 123-124
 - shl 124, 367
 - shr 124, 367
- interval 190-191, 379
 - closed interval 200, 202-203, 379
 - half-open interval 207, 379
 - left-open interval 379
 - open interval 379
 - right-open interval 276, 284, 320, 330, 339, 379
- invalid address 269, 379
- ISA 20, 292, 318, 326

K

- kernarg segment 33, 35, 38, 53, 72-73, 96, 98, 114-115, 132, 166, 171, 175, 267, 375, 379
- kernel 24-25, 32-34, 38-39, 50, 52-53, 56-57, 71, 73, 95-97, 99, 112-115, 132, 171-172, 212, 214, 218-219, 252-254, 259, 262, 266, 271, 276-277, 279-281, 297, 322, 325, 362, 364, 377, 379
 - kernel descriptor 97
- kernel agent 20-21, 24-25, 31-33, 36, 38-39, 53, 71, 114-115, 170-171, 204, 212, 214-215, 218, 230, 264, 267, 272, 274, 283, 376-378, 380, 382
- kernel dispatch 376, 380
- kernel dispatch packet instructions 26, 28, 257-258, 327-328, 366, 371
 - currentworkgroupsize 258
 - dim 258, 327-328, 366
 - gridgroups 258
 - gridsize 258
 - packetcompletion sig 258, 371
 - packetid 259
 - workgroupid 259
 - workgroupsize 259
 - workitemabsid 259
 - workitemflatabsid 28, 259, 366
 - workitemflatid 259
 - workitemid 260

L

- lane 29, 241, 267, 333-334, 380
- library 286-287, 378
- limits 28, 40, 194, 197, 211, 214, 279, 288-289, 375
- linkage 56-60, 64-65, 97, 306, 323-325, 327-328
 - arg linkage 65, 70, 99
 - function linkage 57, 59, 65, 68, 70, 98-99
 - module linkage 57-58, 65, 68, 70, 98, 286, 380
 - program linkage 57-58, 65, 68, 70, 79, 97-98, 286, 381
- loader 380

M

- machine instructions 43
- machine model 39-40, 101, 130-131, 154-155, 166-167, 187, 234, 285, 296
- memory fence 166, 169-170, 192, 194, 233, 237-238, 307, 335
- memory instructions 31, 35, 43, 166, 168-173, 179, 187, 218, 292, 307, 334-335, 378, 380
 - atomic 170, 180-181, 187-188, 190, 229, 264, 302, 331, 366, 368
 - atomicnoret 180-186, 190, 212, 366, 369
 - ld 115, 169, 173-174, 176, 181-182, 216-217, 246, 275, 294, 368
 - ld | AuthoringStatus.DONE | [11] 366
 - memfence 192-193, 335, 366, 369

- signal 40, 43, 101, 187-191, 258, 260, 269, 271-272, 302, 307, 338, 355, 366, 369, 381
- signalnoret 190, 366, 369
- st 169, 177, 179, 182, 217, 246, 275, 366, 368
- memory model 20-21, 32, 169, 171, 188, 218, 295, 376, 378
 - memory order 169, 173, 181, 307, 330-331, 335, 337-338
 - memory scope 95, 170, 172-173, 180-181, 219, 307, 330-331
 - memory segment 26, 36, 153, 167, 171, 314, 376-381
 - synchronizing memory instruction 173, 294-295
- miscellaneous instructions 264-265
 - cuid 266, 366
 - groupbaseptr 266, 366
 - kernargbaseptr 114, 265, 267, 366
 - laneid 267, 366
 - maxcuid 266-267, 366
 - maxwaveid 267, 366
 - nop 267
 - nullptr 153, 267, 366
 - waveid 268, 366
- module header 40, 54, 82, 284, 288, 318
- module linkage 380
- multimedia instructions 150
 - bitalign 150-151, 367
 - bytealign 151, 367
 - lerp 151, 367
 - packcvt 151, 365, 368
 - sad 151-152, 365, 368
 - sadhi 152, 368
 - unpackcvt 151, 365, 368

N

- native floating-point instructions 111, 148, 270
 - ncos 149, 366
 - nexp2 149, 366
 - nfma 149
 - nlog2 149, 270, 366
 - nrcp 149, 292, 366
 - nrsqrt 149, 366
 - nsin 149, 367
 - nsqrt 108, 149, 367
- natural alignment 71-72, 114, 214, 328, 380

P

- packed data 80, 101, 133, 311
- packed data instructions 133
 - pack 135
 - shuffle 134, 367
 - unpack 135
 - unpackhi 135, 367
 - unpacklo 134, 367
- packet 24
- packet ID 380
- packet processor 380
- packing control 101, 311, 332, 336

- padding bytes 314, 320
- partial lane 44
- partial wavefront 44
- partial work-group 25-26, 258
- performance tuning 278
- persistence rules 38
- pixel 195
- popcount 126
- pragma directive 276-277
- private segment 22, 28, 32-34, 36, 38-39, 52, 73, 96-97, 244, 246-247, 250, 252-253, 255-256, 292, 374, 380
- profile 21, 40, 100, 111-112, 148, 202, 284, 288-289, 311, 326, 362
- profiles
 - Base profile 289
 - Full profile 290
- program linkage 381

Q

- queue ID 24, 259, 381

R

- race condition 232, 234-235, 237
- read atomicity 381
- readonly segment 33-34, 38, 64, 70, 72, 94, 95-98, 171, 175, 188, 212-215, 295, 318, 328, 381
- register pressure 292
- release synchronizing instruction 381
- runtime 20-21, 33, 48-49, 53, 112, 173, 190-191, 204, 211-212, 214, 218, 269, 271-272, 276, 279, 378
- runtime library 279

S

- sampler 214-218, 346
 - sampler handle 101, 195-196, 214-215, 217-218, 346
- sampler handle 381
- segment 31, 35-36, 38, 70-71, 100-101, 107, 132, 153-154, 166-167, 170-171, 173, 212, 214-215, 217, 267-268, 296, 327, 330-331, 334, 337-338, 380-381
- segment checking instructions 100, 153
 - segmentp 38, 100, 153, 167, 368
- segment conversion instructions 154, 368
 - ftos 132, 154, 167, 368
 - stof 132, 154-155, 167, 368
- segment modifier 107, 170
- shared virtual memory 35, 379
- shuffle instruction 137
- signature 362
- signed or unsigned 128
- small model 39-40
- special instructions 26, 42, 257, 264, 269
 - addqueuewriteindex 263, 371
 - casqueuewriteindex 263
 - ldqueuereadindex 263, 371
 - ldqueuewriteindex 263-264, 371

stqueueuereadindex 264, 371
 stqueueuwriteindex 264
 spill segment 34, 39-40, 52, 72-73, 96-97, 132, 292, 381

U

uniform instruction 381
 unit of least precision (ULP) 381
 URL 381
 User Mode Queue 382
 user mode queue instructions 262-263
 addqueueuwriteindex 263
 casqueueuwriteindex 263
 ldqueueuereadindex 263
 ldqueueuwriteindex 264
 stqueueuereadindex 264
 stqueueuwriteindex 264

V

variadic function 248
 vector operand 106
 virtual machine 19-20, 277, 378

W

wavefront 29, 39, 42, 232-233, 241, 255, 261-262, 272-273, 347, 380-382
 wavefront size 29-30, 41, 43, 45, 229, 240-241, 267, 281-282, 293, 318, 374, 380
 WAVESIZE 30, 44-45, 94, 104-106, 175, 177, 192, 235-237, 240-242, 267, 276, 284, 318, 347, 361, 382
 width modifier 42, 44-45, 175-176, 228-230, 252, 254, 318, 332, 334-335
 work-group 24-28, 32, 35, 38, 43, 52, 113, 170, 218-219, 229-230, 233-234, 236, 255, 271, 280-281, 322, 374, 377-378, 381-382
 work-group absolute ID 36, 377
 work-group flattened ID 26, 374, 382
 work-group ID 25-26, 260, 382
 work-item 21, 26-27, 32-33, 37-39, 43-44, 52, 96-97, 170, 175-176, 190-191, 218-219, 231-232, 234, 241, 243, 261, 265-267, 273, 281, 380-382
 work-item absolute ID 25, 27, 36, 260, 377, 382
 work-item flattened absolute ID 27-28, 382
 work-item flattened ID 27, 29, 382
 work-item flattened ID, current 27
 work-item ID 26-27, 260, 382
 workitemflatabsid 259
 write atomicity 382