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

Revision: Version 1.2 • Issue Date: 2 May 2018
© 2018 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 and educational institutions that they represented at the time of their contribution.

**AMD**
- Michael Bedy
- Paul Blinzer
- Boleslaw Ciesielski
- Eric Finger
- Mark Fowler
- Mike Houston
- Nikolay Haustov
- Mark Herdeg
- Bill Licea-Kane
- Leonid Lobachev
- Mike Mantor
- Vicki Meagher
- Stanislav Mekhanoshin
- Dmitry Preobrazhensky
- Valery Pykhtin
- Chris Reeve
- Phil Rogers
- Norm Rubin
- Benjamin Sander
- Elizabeth Sanville
- Oleg Semenov
- Brian Sumner
- Yaki Tebeka
- Vinod Tipparaju
- Tony Tye (Spec. Editor)
- Micah Villmow
- Konstantin Zhuravlyov

**ARM**
- Jem Davies
- Ian Devereux
- Robert Elliott
- Alexander Galazin
- Rune Holm

**Arteris**
- Kurt Shuler

**Codeplay**
- Andrew Richards (Workgroup Chair)

**General Processor Technologies**
- Paul D’Arcy
- John Glossner

**HSA Foundation**
- Greg Stoner

**Imagination Technologies**
- Yoong-Chert Foo
- John Howson
- James McCarthy
- Jason Meredith
- Mark Rankilor
- Zoran Zaric

**MediaTek**
- Rahul Agarwal
- Richard Bagley
- Roy Ju
Acknowledgments

- Trent Lo
- Chien-Ping Lu

**MulticoreWare**
- Thomas Jablin
- Chuan Na

**Qualcomm**
- Greg Bellows
- Lihan Bin
- P.J. Bostley
- Alex Bourd
- Ken Dockser
- Jamie Esliger
- Benedict Gaster
- Andrew Gruber
- Lee Howes
- Wilson Kwan
- Jack Liu
- Bob Rychlik
- Robert J. Simpson
- Sumesh Udayakumaran

**Samsung**
- Soojung Ryu

**Tampere University of Technology**
- Pekka Jääskeläinen

**Texas Instruments**
- Matthew Locke

**VTM Group**
- Chelsi Odegaard
## Contents

**Acknowledgments** ........................................................................................................... 3

**About the HSA Programmer's Reference Manual** .............................................................. 20
  - Audience .......................................................................................................................... 20
  - Document Conventions .................................................................................................... 20
  - HSA Information Sources ............................................................................................... 20

**Chapter 1. Overview** ........................................................................................................... 21
  - 1.1 What is HSAIL? ......................................................................................................... 21
  - 1.2 HSAIL virtual language ............................................................................................ 22
  - 1.3 HSAIL experimental features .................................................................................... 23

**Chapter 2. HSAIL Programming Model** ............................................................................ 24
  - 2.1 Overview of grids, work-groups, and work-items ..................................................... 24
  - 2.2 Work-groups ............................................................................................................. 26
    - 2.2.1 Work-group ID .................................................................................................... 26
    - 2.2.2 Work-group flattened ID ..................................................................................... 27
  - 2.3 Work-items ............................................................................................................... 27
    - 2.3.1 Work-item ID ...................................................................................................... 27
    - 2.3.2 Work-item flattened ID and current work-item flattened ID ............................... 28
    - 2.3.3 Work-item absolute ID ....................................................................................... 28
    - 2.3.4 Work-item flattened absolute ID ....................................................................... 28
  - 2.4 Scalable data-parallel computing .............................................................................. 29
  - 2.5 Active work-groups and active work-items .............................................................. 29
  - 2.6 Wavefronts, lanes, and wavefront sizes .................................................................... 29
    - 2.6.1 Example of contents of a wavefront .................................................................. 30
    - 2.6.2 Wavefront size .................................................................................................... 31
  - 2.7 Types of memory ........................................................................................................ 31
  - 2.8 Segments ................................................................................................................... 32
    - 2.8.1 Types of segments ............................................................................................... 32
    - 2.8.2 Shared virtual memory ....................................................................................... 36
    - 2.8.3 Addressing for segments ................................................................................... 36
    - 2.8.4 Memory segment access rules .......................................................................... 37
    - 2.8.5 Memory segment isolation ............................................................................... 39
  - 2.9 Small and large machine models .............................................................................. 40
  - 2.10 Base and full profiles .............................................................................................. 41
  - 2.11 Race conditions ....................................................................................................... 41
  - 2.12 Divergent control flow ............................................................................................ 42
    - 2.12.1 Uniform instructions ......................................................................................... 42
    - 2.12.2 Using the width modifier with control transfer instructions ............................. 45
    - 2.12.3 (Post-)dominator and immediate (post-)dominator ......................................... 46
  - 2.13 Forward progress ..................................................................................................... 46

**Chapter 3. Examples of HSAIL Programs** ....................................................................... 48
  - 3.1 Vector add translated to HSAIL .................................................................................. 48
  - 3.2 Transpose translated to HSAIL .................................................................................. 49
Chapter 4. HSAIL Syntax and Semantics .................................................. 50

4.1 Two formats ............................................................................. 50
4.2 Program, code object, and executable ........................................ 50
  4.2.1 Finalization ........................................................................ 51
  4.2.2 Loading ............................................................................. 53
  4.2.3 Execution .......................................................................... 55
4.3 Module ....................................................................................... 57
  4.3.1 Annotations ....................................................................... 59
  4.3.2 Kernel ............................................................................... 60
  4.3.3 Function ............................................................................ 61
  4.3.4 Signature .......................................................................... 64
  4.3.5 Code block ........................................................................ 65
  4.3.6 Arg block ......................................................................... 66
  4.3.7 Instruction ......................................................................... 67
  4.3.8 Variable ............................................................................ 68
  4.3.9 Fbarrier ............................................................................ 72
  4.3.10 Declaration and definition qualifiers ................................... 73
4.4 Source text format ..................................................................... 79
4.5 Strings ....................................................................................... 80
4.6 Identifiers .................................................................................. 81
  4.6.1 Syntax .............................................................................. 81
  4.6.2 Scope .............................................................................. 82
4.7 Registers .................................................................................... 83
4.8 Constants ................................................................................... 85
  4.8.1 Integer constants ................................................................. 86
  4.8.2 Floating-point constants .................................................... 89
  4.8.3 Typed constants .................................................................. 94
  4.8.4 Aggregate constants .......................................................... 99
  4.8.5 How text format constants are converted to bit string constants 101
4.9 Labels ......................................................................................... 103
4.10 Variable initializers ................................................................... 103
4.11 Storage duration ....................................................................... 105
4.12 Linkage ...................................................................................... 106
  4.12.1 Program linkage ............................................................... 106
  4.12.2 Module linkage ................................................................. 107
  4.12.3 Function linkage ............................................................... 107
  4.12.4 Arg linkage ...................................................................... 108
  4.12.5 None linkage ................................................................... 108
4.13 Data types ................................................................................ 108
  4.13.1 Base data types ................................................................. 108
  4.13.2 Packed data types ............................................................. 109
  4.13.3 Opaque data types ............................................................ 110
  4.13.4 Array data types ............................................................... 110
4.14 Packing controls for packed data .............................................. 110
  4.14.1 Ranges ............................................................................ 111
  4.14.2 Packed typed constants .................................................... 112
4.15 Subword sizes ........................................................................... 113
4.16 Operands .................................................................................. 113
  4.16.1 Operand compound type .................................................. 114
  4.16.2 Rules for operand registers .............................................. 114
4.17 Vector operands ........................................................................ 115
4.18 Address expressions ................................................................. 115
4.19 Floating point ........................................................................... 116
  4.19.1 Floating-point numbers .................................................... 118
Chapter 5. Arithmetic Instructions

5.1 Overview of arithmetic instructions ................................................................. 127
5.2 Integer arithmetic instructions ........................................................................ 127
  5.2.1 Syntax ........................................................................................................ 127
  5.2.2 Description ................................................................................................ 128
5.3 Integer optimization instruction ......................................................................... 132
  5.3.1 Syntax ........................................................................................................ 132
  5.3.2 Description ................................................................................................ 133
5.4 24-bit integer optimization instructions ........................................................... 133
  5.4.1 Syntax ........................................................................................................ 133
  5.4.2 Description ................................................................................................ 134
5.5 Integer shift instructions .................................................................................... 134
  5.5.1 Syntax ........................................................................................................ 134
  5.5.2 Description for standard form .................................................................... 135
  5.5.3 Description for packed form ...................................................................... 135
5.6 Individual bit instructions .................................................................................. 136
  5.6.1 Syntax ........................................................................................................ 136
  5.6.2 Description ................................................................................................ 136
5.7 Bit string instructions ........................................................................................ 138
  5.7.1 Syntax ........................................................................................................ 138
  5.7.2 Description ................................................................................................ 139
5.8 Copy (move) instructions .................................................................................... 141
  5.8.1 Syntax ........................................................................................................ 142
  5.8.2 Description ................................................................................................ 142
  5.8.3 Additional information about lda .............................................................. 143
5.9 Packed data instructions ..................................................................................... 144
  5.9.1 Syntax ........................................................................................................ 144
  5.9.2 Description ................................................................................................ 145
  5.9.3 Controls in src2 for shuffle instruction ..................................................... 147
  5.9.4 Common uses for shuffle instruction ......................................................... 148
  5.9.5 Examples of unpacklo and unpackhi instructions ...................................... 150
5.10 Bit conditional move (cmov) instruction ........................................................... 151
  5.10.1 Syntax ....................................................................................................... 151
  5.10.2 Description ................................................................................................ 151
5.11 Floating-point arithmetic instructions .............................................................. 151
  5.11.1 Syntax ....................................................................................................... 152
  5.11.2 Description ................................................................................................ 153
5.12 Floating-point optimization instruction ............................................................ 155
  5.12.1 Syntax ....................................................................................................... 155
  5.12.2 Description ................................................................................................ 156
5.13 Floating-point bit instructions .......................................................................... 157
  5.13.1 Syntax ....................................................................................................... 157
  5.13.2 Description ................................................................................................ 158
5.14 Native floating-point instructions ..................................................................... 159
  5.14.1 Syntax ....................................................................................................... 159
  5.14.2 Description ................................................................................................ 160
Chapter 6. Memory Instructions .............................................. 180

6.1 Memory and addressing ................................................. 180
   6.1.1 How addresses are formed .................................... 180
   6.1.2 Memory hierarchy ................................... 181
   6.1.3 Alignment ........................................ 182
   6.1.4 Equivalence classes ........................................ 182
6.2 Memory model .......................................................... 183
   6.2.1 Memory order ............................................... 183
   6.2.2 Memory scope ........................................ 184
   6.2.3 Memory synchronization segments ........................... 184
   6.2.4 Non-memory synchronization segments ...................... 185
   6.2.5 Agent allocation ........................................ 185
   6.2.6 Coarse grain allocation ..................................... 186
   6.2.7 Kernel dispatch memory synchronization .................... 186
   6.2.8 Execution barrier ........................................ 187
   6.2.9 Flat addresses ........................................... 187
6.3 Load (ld) instruction .................................................. 187
   6.3.1 Syntax ............................................... 188
   6.3.2 Description ........................................ 189
   6.3.3 Additional information .................................... 190
6.4 Store (st) instruction .................................................. 191
   6.4.1 Syntax ............................................... 192
   6.4.2 Description ........................................ 192
   6.4.3 Additional information .................................... 193
6.5 Atomic memory instructions ......................................... 194
6.6 Atomic (atomic) instructions ......................................... 195
   6.6.1 Syntax ............................................... 196
   6.6.2 Description of atomic and atomic no return instructions ........................................... 196
6.7 Atomic no return (atomicnoret) instructions ....................... 200
   6.7.1 Syntax ............................................... 200
   6.7.2 Description ........................................ 201
6.8 Notification (signal) instructions .................................... 202
   6.8.1 Syntax ............................................... 203
   6.8.2 Description ........................................ 204
Chapter 7. Image Instructions 209

7.1 Images in HSAIL 209
7.1.1 Why use images? 209
7.1.2 Image overview 210
7.1.3 Image geometry 211
7.1.4 Image format 213
   7.1.4.1 Channel order 213
      7.1.4.1.1 x-form channel orders 214
      7.1.4.1.2 Standard RGB (s-form) channel orders 214
   7.1.4.2 Channel type 216
   7.1.4.3 Bits per pixel (bpp) 219
7.1.5 Image access permission 220
7.1.6 Image coordinate 222
   7.1.6.1 Coordinate normalization mode 222
   7.1.6.2 Addressing mode 223
   7.1.6.3 Filter mode 225
7.1.7 Image creation and image handles 227
7.1.8 Sampler creation and sampler handles 232
7.1.9 Using image instructions 234
7.1.10 Image memory model 236
7.2 Read image (rdimage) instruction 237
7.2.1 Syntax 238
7.2.2 Description 239
7.3 Load image (ldimage) instruction 239
7.3.1 Syntax 239
7.3.2 Description 240
7.4 Store image (stimage) instruction 241
7.4.1 Syntax 241
7.4.2 Description 242
7.5 Query image and query sampler instructions 242
7.5.1 Syntax 243
7.5.2 Description 243
7.6 Image fence (imagefence) instruction 244
   7.6.1 Syntax 244
   7.6.2 Description 244

Chapter 8. Branch Instructions 246

8.1 Syntax 246
8.2 Description 246

Chapter 9. Parallel Synchronization and Communication Instructions 248

9.1 Barrier instructions 248
   9.1.1 Syntax 248
   9.1.2 Description 248
9.2 Fine-grain barrier (fbarrier) instructions 249
   9.2.1 Overview: What is an fbarrier? 249
   9.2.2 Syntax 250
   9.2.3 Description 250
   9.2.4 Additional information about fbarrier instructions 253
18.5.2.12 hsa_brig_inst_queue_t ........................................................................ 377
18.5.2.13 hsa_brig_inst_seg_t ........................................................................ 377
18.5.2.14 hsa_brig_inst_seg_cvt_t ................................................................... 378
18.5.2.15 hsa_brig_inst_signal_t ................................................................. 378
18.5.2.16 hsa_brig_inst_source_type_t ......................................................... 379
18.5.2.17 hsa_ext_brig_inst_image_t ............................................................. 379
18.5.2.18 hsa_ext_brig_inst_query_image_t .................................................. 380
18.5.2.19 hsa_ext_brig_inst_query_sampler_t .............................................. 380
18.6 hsa_operand section ............................................................................. 380
18.6.1 Constant operands ................................................................. 382
18.6.2 hsa_brig_operand_address_t .............................................................. 383
18.6.3 hsa_brig_operand_align_t ................................................................. 384
18.6.4 hsa_brig_operand_code_list_t .......................................................... 384
18.6.5 hsa_brig_operand_code_ref_t ............................................................ 384
18.6.6 hsa_brig_operand_constant_bytes_t .................................................. 385
18.6.7 hsa_brig_operand_constant_expression_t ........................................... 386
18.6.8 hsa_brig_operand_constant_operand_list_t ..................................... 387
18.6.9 hsa_brig_operand_operand_list_t ..................................................... 388
18.6.10 hsa_brig_operand_register_t ............................................................ 389
18.6.11 hsa_brig_operand_string_t .............................................................. 389
18.6.12 hsa_brig_operand_wavesize_t .......................................................... 390
18.6.13 hsa_brig_operand_zero_t ................................................................. 390
18.6.14 hsa_ext_brig_operand_constant_image_t ........................................ 390
18.6.15 hsa_ext_brig_operand_constant_sampler_t ...................................... 391
18.7 BRIG syntax for instructions ............................................................... 392
18.7.1 BRIG syntax for arithmetic instructions .......................................... 392
18.7.1.1 BRIG syntax for integer arithmetic instructions ............................... 392
18.7.1.2 BRIG syntax for integer optimization instruction ............................ 393
18.7.1.3 BRIG syntax for 24-bit integer optimization instructions .................. 393
18.7.1.4 BRIG syntax for integer shift instructions ....................................... 394
18.7.1.5 BRIG syntax for individual bit instructions ..................................... 394
18.7.1.6 BRIG syntax for bit string instructions .......................................... 394
18.7.1.7 BRIG syntax for copy (move) instructions ...................................... 395
18.7.1.8 BRIG syntax for packed data instructions ..................................... 395
18.7.1.9 BRIG syntax for bit conditional move (cmov) instruction ................. 395
18.7.1.10 BRIG syntax for floating-point arithmetic instructions .................. 396
18.7.1.11 BRIG syntax for floating-point optimization instruction ................ 397
18.7.1.12 BRIG syntax for floating-point bit instructions ............................. 397
18.7.1.13 BRIG syntax for native floating-point instructions .......................... 397
18.7.1.14 BRIG syntax for multimedia instructions ..................................... 398
18.7.1.15 BRIG syntax for segment checking (segmentp) instruction ............. 398
18.7.1.16 BRIG syntax for segment conversion instructions ......................... 398
18.7.1.17 BRIG syntax for compare (cmp) instruction .................................. 399
18.7.1.18 BRIG syntax for conversion (cvt) instruction .................................. 399
18.7.2 BRIG syntax for memory instructions .............................................. 399
18.7.3 BRIG syntax for image instructions ............................................... 400
18.7.4 BRIG syntax for branch instructions .............................................. 401
18.7.5 BRIG syntax for parallel synchronization and communication instructions ................................................. 401
18.7.6 BRIG syntax for function instructions ............................................. 402
18.7.7 BRIG syntax for special instructions .............................................. 403
18.7.7.1 BRIG syntax for kernel dispatch packet instructions ....................... 403
18.7.7.2 BRIG syntax for exception instructions ......................................... 403
18.7.7.3 BRIG syntax for user mode queue instructions .............................. 404
18.7.7.4 BRIG syntax for miscellaneous instructions .................................. 404
Chapter 19. HSAIL Grammar in Extended Backus-Naur Form .......... 405

19.1 HSAIL lexical grammar in Extended Backus-Naur Form (EBNF) ................................................. 405
19.2 HSAIL syntax grammar in Extended Backus-Naur Form (EBNF) .................................................. 406

Appendix A. Limits ................................................................. 420

Appendix B. Glossary of HSAIL Terms ............................... 421

Index .................................................................................. 429

Figures

Figure 2-1 A grid and its work-groups and work-items ............................................................... 24
Figure 2-2 TOKEN_WAVESIZE syntax diagram ........................................................................... 31
Figure 4-1 HSA runtime support for HSAIL life cycle ............................................................... 51
Figure 4-2 module syntax diagram .............................................................................................. 57
Figure 4-3 moduleHeader syntax diagram ................................................................................... 57
Figure 4-4 profile syntax diagram ................................................................................................ 58
Figure 4-5 machineModel syntax diagram ................................................................................... 58
Figure 4-6 defaultFloatRounding syntax diagram ................................................................. 58
Figure 4-7 moduleDirective syntax diagram ............................................................................... 58
Figure 4-8 moduleStatement syntax diagram ............................................................................. 59
Figure 4-9 annotations syntax diagram ....................................................................................... 59
Figure 4-10 annotation syntax diagram ....................................................................................... 59
Figure 4-11 TOKEN_COMMENT syntax diagram .......................................................................... 60
Figure 4-12 kernel syntax diagram .............................................................................................. 61
Figure 4-13 kernelHeader syntax diagram .................................................................................. 61
Figure 4-14 kernFormalArgumentList syntax diagram ............................................................. 61
Figure 4-15 kernFormalArgument syntax diagram ................................................................... 61
Figure 4-16 function syntax diagram .......................................................................................... 62
Figure 4-17 functionHeader syntax diagram .............................................................................. 63
Figure 4-18 funcOutputFormalArgumentList syntax diagram .................................................. 63
Figure 4-19 funcInputFormalArgumentList syntax diagram ..................................................... 63
Figure 4-20 funcFormalArgumentList syntax diagram .............................................................. 63
Figure 4-21 funcFormalArgument syntax diagram .................................................................... 63
Figure 4-22 signature syntax diagram ......................................................................................... 64
Figure 4-23 sigOutputFormalArgumentList ............................................................................. 64
Figure 4-24 sigInputFormalArgumentList syntax diagram ...................................................... 64
Figure 4-25 sigFormalArgumentList syntax diagram ............................................................... 64
Figure 4-26 sigFormalArgument syntax diagram ....................................................................... 65
Figure 4-27 codeBlock syntax diagram ...................................................................................... 65
Figure 4-28 codeBlockDirective syntax diagram ....................................................................... 65
Figure 4-29 codeBlockDefinition ............................................................................................... 65
Figure 4-30 codeBlockStatement syntax diagram ....................................................................... 66
Figure 4-31 argBlock syntax diagram ........................................................................................ 66
Figure 4-32 argBlockDefinition .................................................................................................. 66
Figure 4-33 argBlockStatement syntax diagram ........................................................................ 67
Figure 4-34 moduleVariable syntax diagram ........................................................................... 69
Figure 4-35 codeBlockVariable syntax diagram ........................................................................ 69
Figure 4-36 argBlockVariable syntax diagram .......................................................................... 69
Figure 4-37 variable syntax diagram .......................................................................................... 70
Figure 4-38 variableSegment syntax diagram ........................................................................... 70
Figure 4-39 dataTypeMod syntax diagram .................................................................................. 71
Figure 4-95 aggregateConstantZero syntax diagram ........................................ 101
Figure 4-96 optinitializer syntax diagram ......................................................... 103
Figure 4-97 packedTypedConstant syntax diagram .......................................... 112
Figure 5-1 Example of broadcast ................................................................. 149
Figure 5-2 Example of rotate ........................................................................ 149
Figure 5-3 Example of unpack ....................................................................... 150
Figure 6-1 Memory hierarchy ....................................................................... 182
Figure 13-1 extension syntax diagram ......................................................... 293
Figure 13-2 location syntax diagram ............................................................. 296
Figure 13-3 pad syntax diagram .................................................................. 296
Figure 13-4 pragma syntax diagram ............................................................ 297
Figure 13-5 pragmaOperand syntax diagram .............................................. 297
Figure 14-1 moduleHeader syntax diagram .................................................. 323

Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Wavefront 0 through 6</td>
<td>30</td>
</tr>
<tr>
<td>2-2</td>
<td>Memory segment access rules</td>
<td>37</td>
</tr>
<tr>
<td>2-3</td>
<td>Machine model data sizes</td>
<td>41</td>
</tr>
<tr>
<td>4-1</td>
<td>Text constants and results of the conversion</td>
<td>101</td>
</tr>
<tr>
<td>4-2</td>
<td>Base data types</td>
<td>108</td>
</tr>
<tr>
<td>4-3</td>
<td>Packed data types and possible lengths</td>
<td>109</td>
</tr>
<tr>
<td>4-4</td>
<td>Opaque data types</td>
<td>110</td>
</tr>
<tr>
<td>4-5</td>
<td>Packing controls for instructions with one source input</td>
<td>111</td>
</tr>
<tr>
<td>4-6</td>
<td>Packing controls for instructions with two source inputs</td>
<td>111</td>
</tr>
<tr>
<td>5-1</td>
<td>Syntax for integer arithmetic instructions</td>
<td>127</td>
</tr>
<tr>
<td>5-2</td>
<td>Syntax for packed versions of integer arithmetic instructions</td>
<td>128</td>
</tr>
<tr>
<td>5-3</td>
<td>Syntax for integer optimization instruction</td>
<td>132</td>
</tr>
<tr>
<td>5-4</td>
<td>Syntax for 24-bit integer optimization instructions</td>
<td>133</td>
</tr>
<tr>
<td>5-5</td>
<td>Syntax for integer shift instructions</td>
<td>134</td>
</tr>
<tr>
<td>5-6</td>
<td>Syntax for individual bit instructions</td>
<td>136</td>
</tr>
<tr>
<td>5-7</td>
<td>Inputs and Results for popcount Instruction</td>
<td>137</td>
</tr>
<tr>
<td>5-8</td>
<td>Syntax for bit string instructions</td>
<td>138</td>
</tr>
<tr>
<td>5-9</td>
<td>Inputs and results for firstbit and lastbit instructions</td>
<td>141</td>
</tr>
<tr>
<td>5-10</td>
<td>Syntax for copy (move) instructions</td>
<td>142</td>
</tr>
<tr>
<td>5-11</td>
<td>Syntax for shuffle and interleave instructions</td>
<td>144</td>
</tr>
<tr>
<td>5-12</td>
<td>Syntax for pack and unpack instructions</td>
<td>145</td>
</tr>
<tr>
<td>5-13</td>
<td>Bit selectors for shuffle instruction</td>
<td>147</td>
</tr>
<tr>
<td>5-14</td>
<td>Syntax for bit conditional move (cmov) instruction</td>
<td>151</td>
</tr>
<tr>
<td>5-15</td>
<td>Syntax for floating-point arithmetic instructions</td>
<td>152</td>
</tr>
<tr>
<td>5-16</td>
<td>Syntax for packed versions of floating-point arithmetic instructions</td>
<td>153</td>
</tr>
<tr>
<td>5-17</td>
<td>Syntax for floating-point optimization instruction</td>
<td>155</td>
</tr>
<tr>
<td>5-18</td>
<td>Syntax for floating-point bit instructions</td>
<td>157</td>
</tr>
<tr>
<td>5-19</td>
<td>Class instruction source operand condition bits</td>
<td>157</td>
</tr>
<tr>
<td>5-20</td>
<td>Syntax for packed versions of floating-point bit instructions</td>
<td>158</td>
</tr>
<tr>
<td>5-21</td>
<td>Syntax for native floating-point instructions</td>
<td>159</td>
</tr>
<tr>
<td>5-22</td>
<td>Syntax for multimedia instructions</td>
<td>161</td>
</tr>
<tr>
<td>5-23</td>
<td>Syntax for segment checking (segmentp) instruction</td>
<td>164</td>
</tr>
<tr>
<td>5-24</td>
<td>Syntax for segment conversion instructions</td>
<td>165</td>
</tr>
<tr>
<td>5-25</td>
<td>Syntax for compare (cmp) instruction</td>
<td>167</td>
</tr>
<tr>
<td>5-26</td>
<td>Syntax for packed version of compare (cmp) instruction</td>
<td>168</td>
</tr>
<tr>
<td>5-27</td>
<td>Floating-point comparisons</td>
<td>169</td>
</tr>
<tr>
<td>5-28</td>
<td>Conversion methods</td>
<td>171</td>
</tr>
<tr>
<td>5-29</td>
<td>Notation for conversion methods</td>
<td>173</td>
</tr>
<tr>
<td>5-30</td>
<td>Syntax for conversion (cvt) instruction</td>
<td>173</td>
</tr>
</tbody>
</table>
Table 5–31 Syntax for packed versions of conversion (cvt) instructions .................................................. 174
Table 5–32 Rules for rounding for conversions ......................................................................................... 175
Table 5–33 Integer rounding modes ........................................................................................................ 177
Table 6–1 Syntax for load (ld) instruction .................................................................................................. 188
Table 6–2 Syntax for store (st) instruction ................................................................................................. 192
Table 6–3 Syntax for atomic instructions .................................................................................................. 196
Table 6–4 Syntax for atomic no return instructions .................................................................................. 200
Table 6–5 Syntax for signal instructions .................................................................................................. 203
Table 6–6 Syntax for memfence instruction ............................................................................................... 208
Table 7–1 Image geometry properties .................................................................................................... 211
Table 7–2 Channel order properties ........................................................................................................ 214
Table 7–3 Channel type properties ......................................................................................................... 216
Table 7–4 Channel order, channel type, and image geometry combination ............................................. 221
Table 7–5 Image handle properties ......................................................................................................... 230
Table 7–6 Image instruction combinations ............................................................................................. 234
Table 7–7 Syntax for read image instruction ........................................................................................... 238
Table 7–8 Syntax for load image instruction ............................................................................................ 239
Table 7–9 Syntax for store image instruction ........................................................................................... 241
Table 7–10 Syntax for query image and query sampler instructions ....................................................... 243
Table 7–11 Explanation of imageProperty modifier .............................................................................. 243
Table 7–12 Explanation of samplerProperty modifier ............................................................................ 244
Table 7–13 Syntax for imagefence instruction .......................................................................................... 244
Table 8–1 Syntax for branch instructions ................................................................................................ 246
Table 9–1 Syntax for barrier instructions ................................................................................................ 248
Table 9–2 Syntax for fbar instructions ..................................................................................................... 250
Table 9–3 Syntax for cross-lane instructions ........................................................................................... 259
Table 10–1 Syntax for direct call instruction ........................................................................................... 269
Table 10–2 Syntax for switch call instruction ........................................................................................... 270
Table 10–3 Syntax for indirect call instruction ......................................................................................... 272
Table 10–4 Syntax for ret instruction ...................................................................................................... 274
Table 10–5 Syntax for allocate memory (alloca) instruction .................................................................. 274
Table 11–1 Syntax for kernel dispatch packet instructions .................................................................... 276
Table 11–2 Syntax for exception instructions ......................................................................................... 279
Table 11–3 Syntax for exception instructions ........................................................................................ 281
Table 11–4 Syntax for miscellaneous instructions ................................................................................... 284
Table 13–1 Control directives for low-level performance tuning ............................................................. 317
Table 18–1 Formats of directives in the has_code section ...................................................................... 361
Table 18–2 Formats of instructions in the has_code section .................................................................. 370
Table 18–3 Formats of operands in the has_operand section .................................................................. 381
Table 18–4 BRIG syntax for integer arithmetic instructions .................................................................. 392
Table 18–5 BRIG syntax for integer optimization instruction ............................................................... 393
Table 18–6 BRIG syntax for 24-bit integer optimization instructions ..................................................... 393
Table 18–7 BRIG syntax for integer shift instructions ............................................................................ 394
Table 18–8 BRIG syntax for individual bit instructions .......................................................................... 394
Table 18–9 BRIG syntax for bit string instructions ................................................................................ 394
Table 18–10 BRIG syntax for copy (move) instructions .......................................................................... 395
Table 18–11 BRIG syntax for packed data instructions .......................................................................... 395
Table 18–12 BRIG syntax for bit conditional move (cmov) instruction ................................................... 395
Table 18–13 BRIG syntax for floating-point arithmetic instructions ....................................................... 396
Table 18–14 BRIG syntax for floating-point optimization instruction .................................................... 397
Table 18–15 BRIG syntax for floating-point bit instructions .................................................................. 397
Table 18–16 BRIG syntax for native floating-point instructions ............................................................. 397
Table 18–17 BRIG syntax for multimedia instructions .......................................................................... 398
Table 18–18 BRIG syntax for segment checking (segmentp) instruction ............................................... 398
Table 18–19 BRIG syntax for segment conversion instructions ............................................................... 398
Table 18–20 BRIG syntax for compare (cmp) instruction .................................................. 399
Table 18–21 BRIG syntax for conversion (cvt) instruction ............................................. 399
Table 18–22 BRIG syntax for memory instructions ......................................................... 399
Table 18–23 BRIG syntax for image instructions ............................................................ 400
Table 18–24 BRIG syntax for branch instructions .......................................................... 401
Table 18–25 BRIG syntax for parallel synchronization and communication instructions .............................. 401
Table 18–26 BRIG syntax for instructions related to functions ........................................ 402
Table 18–27 BRIG syntax for kernel dispatch packet instructions .................................... 403
Table 18–28 BRIG syntax for exception instructions ..................................................... 403
Table 18–29 BRIG syntax for user mode queue instructions ........................................... 404
Table 18–30 BRIG syntax for miscellaneous instructions .............................................. 404
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

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Boldface</strong></td>
<td>In syntax tables, indicates a required item.</td>
</tr>
<tr>
<td><em>Italics</em></td>
<td>In text, indicates the name of a document or a new term that is described in the Appendix B Glossary of HSAIL Terms (on page 421). In syntax tables, indicates a variable representation of a modifier or operand.</td>
</tr>
<tr>
<td><strong>Monospace text</strong></td>
<td>Indicates actual syntax.</td>
</tr>
<tr>
<td><em>n</em></td>
<td>Indicates the generic use of a number.</td>
</tr>
</tbody>
</table>

HSA Information Sources

- 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 machine code if the hardware components cannot support HSAIL natively.
  - A loader loads machine 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 Architectured 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 machine code.

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 (on page 40).

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 (on page 327). 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 (on page 48).

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.
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.

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 a code object that contains the kernel 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 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 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 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 machine 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 (on page 276) and the HSA Platform System Architecture Specification Version 1.2, section 2.8 Requirement: User mode queuing and section 2.9 Requirement: Architected Queuing Language (AQL).

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 (on page 27).

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 (on the next 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 (on 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 Figure 2–1 (on page 24), the grid is composed of 24 work-groups (dimension X = 2, dimension Y = 4, and dimension Z = 3).

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 (on 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 (see 2.13 Forward progress (on page 46)).

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 (see 2.3.3 Work-item absolute ID (on page 28) by the corresponding work-group size component and ignoring the remainder. The value of the work-group ID is returned by the workgroupid instruction.
Work-group size is the product of the three dimensions:

\[
\text{work-group size} = \text{workgroupsize}_0 \times \text{workgroupsize}_1 \times \text{workgroupsize}_2
\]

The size of the work-group specified when the kernel was dispatched is returned by the \text{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 \text{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 \text{workgroupsize} instruction if the current work-item belongs to a partial work-group.

Every grid has a multidimensional count of the number of work-groups for the corresponding grid dimension which is called the \text{grid work-group count}. The grid work-group count is calculated by dividing each component of the grid work-item size by the corresponding work-group size component and rounding up to an integral value if there is any remainder. The value of the grid work-group count is returned by the \text{gridgroups} instruction.

### 2.2.2 Work-group flattened ID

Each work-group has a \text{work-group flattened ID}.

The work-group flattened ID is defined as:

\[
\text{work-group flattened ID} = \text{workgroupid}_0 + \\
\quad \text{workgroupid}_1 \times \text{gridgroups}_0 + \\
\quad \text{workgroupid}_2 \times \text{gridgroups}_0 \times \text{gridgroups}_1
\]

### 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, work-group size, grid work-group count, and so forth through the use of dispatch packet instructions. See 11.1 Kernel dispatch packet instructions (on page 276).

Work-items are able to have local data through a memory segment called the \text{private segment}. Memory in a private segment is accessed using loads and stores. This memory is not accessible outside its associated work-item (that is, it is not seen by other work-items or agents).

Work-items are able to share data with other work-items in the same work-group through a memory segment called the \text{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 (on page 32).

### 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 \text{work-item ID}.

\text{workgroupsize}_i \text{ is the size of the work-group for dimension } i, \text{ or 1 if the work-group has fewer dimensions.}

For each dimension \( i \), the set of values of work-item ID\(_i\) is the dense set \([0, 1, 2, \ldots, \text{workgroupsize}_i - 1]\).

The work-group size can be accessed by means of the dispatch packet instruction \text{workgroupsize}.

The work-item ID can be accessed by means of the dispatch packet instruction \text{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:

\[
\text{work-item flattened ID} = \text{workitemid}_0 + \\
(\text{workitemid}_1 \times \text{workgroupsize}_0) + \\
(\text{workitemid}_2 \times \text{workgroupsize}_0 \times \text{workgroupsize}_1)
\]

The work-item flattened ID can be accessed by means of the dispatch packet 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 (on page 24)) 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:

\[
\text{current work-item flattened ID} = \text{workitemid}_0 + \\
(\text{workitemid}_1 \times \text{currentworkgroupsize}_0) + \\
(\text{workitemid}_2 \times \text{currentworkgroupsize}_0 \times \text{currentworkgroupsize}_1)
\]

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 dispatch packet 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. `gridsize_i` is the size of the grid for dimension i, or 1 if the grid has fewer dimensions.

For each dimension i, the set of values of work-item absolute ID_i is the dense set \([0, 1, 2, \ldots, \text{gridsize}_i - 1]\).

The grid size can be accessed by means of the dispatch packet instruction `gridsize`.

The work-item absolute ID can be accessed by means of the dispatch packet 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:

\[
\text{work-item flattened absolute ID} = \text{workitemabsid}_0 + \\
(\text{workitemabsid}_1 \times \text{gridsize}_0) + \\
(\text{workitemabsid}_2 \times \text{gridsize}_0 \times \text{gridsize}_1)
\]

The work-item flattened absolute ID can be accessed by means of the dispatch packet 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:

```cpp
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 \((i + 1)\) 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 (on the previous page)).
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:

\[
\text{current work-item flattened ID} \mod \text{wavefront size}
\]

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 (on page 24).

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 \(13 \times 3 \times 11 = 429\) 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

<table>
<thead>
<tr>
<th>Wavefront 0</th>
<th>Wavefront 1</th>
<th>Wavefront 2</th>
<th>Wavefront 3</th>
<th>Wavefront 4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dimensions X, Y, Z</strong></td>
<td>0-12, 0, 0</td>
<td>0-12, 1, 1</td>
<td>11-12, 0, 3</td>
<td>10-12, 1, 6</td>
</tr>
<tr>
<td><strong>Work-Item Absolute Flattened IDs</strong></td>
<td>0-12</td>
<td>64</td>
<td>128-129</td>
<td>192-194</td>
</tr>
<tr>
<td><strong>Lane IDs</strong></td>
<td>0-12</td>
<td>12-12</td>
<td>10-12</td>
<td>0-12</td>
</tr>
<tr>
<td><strong>Dimensions X, Y, Z</strong></td>
<td>0-12, 2, 0</td>
<td>0-12</td>
<td>0-12</td>
<td>0-12</td>
</tr>
<tr>
<td><strong>Work-Item Absolute Flattened IDs</strong></td>
<td>0-12</td>
<td>13-25</td>
<td>130-142</td>
<td>192-235</td>
</tr>
<tr>
<td><strong>Lane IDs</strong></td>
<td>0-12</td>
<td>12-12</td>
<td>10-12</td>
<td>0-12</td>
</tr>
<tr>
<td><strong>Dimensions X, Y, Z</strong></td>
<td>0-12, 0, 1</td>
<td>0-12</td>
<td>0-12</td>
<td>0-12</td>
</tr>
<tr>
<td><strong>Work-Item Absolute Flattened IDs</strong></td>
<td>0-12</td>
<td>26-38</td>
<td>143-155</td>
<td>208-220</td>
</tr>
<tr>
<td><strong>Lane IDs</strong></td>
<td>0-12</td>
<td>12-12</td>
<td>10-12</td>
<td>0-12</td>
</tr>
<tr>
<td><strong>Dimensions X, Y, Z</strong></td>
<td>0-12, 1, 1</td>
<td>0-12</td>
<td>0-12</td>
<td>0-12</td>
</tr>
<tr>
<td><strong>Work-Item Absolute Flattened IDs</strong></td>
<td>0-12</td>
<td>39-51</td>
<td>156-168</td>
<td>221-233</td>
</tr>
<tr>
<td><strong>Lane IDs</strong></td>
<td>0-12</td>
<td>12-12</td>
<td>10-12</td>
<td>0-12</td>
</tr>
<tr>
<td><strong>Dimensions X, Y, Z</strong></td>
<td>0-12, 0, 1</td>
<td>0-12</td>
<td>0-12</td>
<td>0-12</td>
</tr>
<tr>
<td><strong>Work-Item Absolute Flattened IDs</strong></td>
<td>0-12</td>
<td>52-63</td>
<td>169-181</td>
<td>234-246</td>
</tr>
<tr>
<td><strong>Lane IDs</strong></td>
<td>0-12</td>
<td>12-12</td>
<td>10-12</td>
<td>0-12</td>
</tr>
</tbody>
</table>
2.6.2 Wavefront size

On some implementations, a kernel might be more efficient if it is written with knowledge of the wavefront size. Thus, HSAIL includes a compile-time macro, WAVESIZE. 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 \textbf{2.12 Divergent control flow (on page 42)}.

WAVESIZE is only available inside the HSAIL code.

In Extended Backus-Naur Form, WAVESIZE is called \texttt{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 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 \textbf{2.8 Segments (on the next page)}.

- Registers
There are four register sizes:
- 1-bit
- 32-bit
- 64-bit
- 128-bit

Registers are untyped.

For more information, see 4.7 Registers (on page 83).

- 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 (on page 209).

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 (on 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 (on page 73).

  - 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 machine 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 memory copy API.

The visibility of global memory is further constrained by the memory model (see 6.2 Memory model (on page 183)). For a description of the visibility of variable initializers, see 4.10 Variable initializers (on page 103).

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 (on page 73).

Standard page protections (for example, read-only, read-write, and protected) apply to global memory. See the HSA Platform System Architecture Specification Version 1.2, 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 (on page 183).

  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 (on page 36).

- **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 (on page 36).

- 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 machine 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 (on page 73).

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 (on page 103).

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 (on page 73).

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 machine 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 (on page 263).

Also see:

• 4.6.2 Scope (on page 82)
• 4.11 Storage duration (on page 105)
• 4.3.10 Declaration and definition qualifiers (on page 73)
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 \((P − \text{start of the segment})\) 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 \((P − \text{start of the group segment})\) 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 (on page 180).

See also:
- 5.16 Segment checking (segmentp) instruction (on page 164)
- 5.17 Segment conversion instructions (on page 165)

### 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>
<thead>
<tr>
<th>Segment</th>
<th>HSA Kernel Agent interaction (HSAIL)</th>
<th>Non-HSA Kernel Agent interaction</th>
<th>Persistence</th>
<th>Allocation</th>
<th>Definition can be initialized?</th>
<th>Where can variables be defined?</th>
<th>Can be accessed by a flat address?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>General global space; non-const variables read-write; const variables read-only and value must not change during storage duration of variable.</td>
<td>Read-write by all agents.</td>
<td>Application</td>
<td>Program or agent</td>
<td>Optional for non-const variables; required for const variables</td>
<td>module; kernel or function code block</td>
<td>Yes</td>
</tr>
<tr>
<td>Readonly</td>
<td>Read-only; value must not change during execution of kernel dispatch.</td>
<td>Can be written by host CPU agent using HSA runtime, provided no kernel dispatch is executing that is using variable.</td>
<td>Application</td>
<td>Agent</td>
<td>Optional</td>
<td>module; kernel or function code block</td>
<td>No</td>
</tr>
<tr>
<td>Segment</td>
<td>HSA Kernel Agent interaction (HSAIL)</td>
<td>Non-HSA Kernel Agent interaction</td>
<td>Persistence</td>
<td>Allocation</td>
<td>Definition can be initialized?</td>
<td>Where can variables be defined?</td>
<td>Can be accessed by a flat address?</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------</td>
<td>----------------------------------</td>
<td>-------------</td>
<td>------------</td>
<td>-------------------------------</td>
<td>-------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>Kernarg</td>
<td>Holds kernel arguments; read-only; value must not change during execution of kernel dispatch.</td>
<td>Initial values provided by the agent when the kernel dispatch is queued. Initial values must not be changed while kernel dispatch is executing.</td>
<td>Kernel</td>
<td>Automatic</td>
<td>No</td>
<td>kernel formal argument list</td>
<td>No</td>
</tr>
<tr>
<td>Group</td>
<td>Read-write.</td>
<td>Inaccessible.</td>
<td>Work-group</td>
<td>Automatic</td>
<td>No</td>
<td>module; kernel or function code block</td>
<td>Yes</td>
</tr>
<tr>
<td>Arg</td>
<td>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 lda instruction.</td>
<td>Inaccessible.</td>
<td>Work-item</td>
<td>Automatic</td>
<td>No</td>
<td>kernel or function arg block; function formal arguments</td>
<td>No</td>
</tr>
<tr>
<td>Private</td>
<td>Holds work-item local variables; read-write.</td>
<td>Inaccessible.</td>
<td>Work-item</td>
<td>Automatic</td>
<td>No</td>
<td>module; kernel or function code block</td>
<td>Yes</td>
</tr>
<tr>
<td>Spill</td>
<td>Holds spilled register values; read-write; cannot have address taken with lda instruction.</td>
<td>Inaccessible.</td>
<td>Work-item</td>
<td>Automatic</td>
<td>No</td>
<td>kernel or function code block</td>
<td>No</td>
</tr>
</tbody>
</table>

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 (on page 82) and 4.11 Storage duration (on page 105).

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 result of a conversion back to a segment address is only defined if the conversion is to the same segment kind as the original segment address. This allows a `segmentp` 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 (segmentp) instruction (on page 164).

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 (on page 73).

• Initializers, see 4.10 Variable initializers (on page 103).

### 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 (on the facing 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 (on page 323).
Table 2–3 Machine model data sizes

<table>
<thead>
<tr>
<th></th>
<th>Small</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flat address</td>
<td>32-bit</td>
<td>64-bit</td>
</tr>
<tr>
<td>Global segment address</td>
<td>32-bit</td>
<td>64-bit</td>
</tr>
<tr>
<td>Readonly segment address</td>
<td>32-bit</td>
<td>64-bit</td>
</tr>
<tr>
<td>Kernarg segment address</td>
<td>32-bit</td>
<td>64-bit</td>
</tr>
<tr>
<td>Group segment address</td>
<td>32-bit</td>
<td>32-bit</td>
</tr>
<tr>
<td>Arg segment address</td>
<td>32-bit</td>
<td>32-bit</td>
</tr>
<tr>
<td>Private segment address</td>
<td>32-bit</td>
<td>32-bit</td>
</tr>
<tr>
<td>Spill segment address</td>
<td>32-bit</td>
<td>32-bit</td>
</tr>
<tr>
<td>Fbarrier address</td>
<td>32-bit</td>
<td>32-bit</td>
</tr>
<tr>
<td>Address expression offset</td>
<td>32-bit</td>
<td>64-bit</td>
</tr>
<tr>
<td>Atomic value</td>
<td>32-bit</td>
<td>32-bit &amp; 64-bit</td>
</tr>
<tr>
<td>Signal value</td>
<td>32-bit</td>
<td>64-bit</td>
</tr>
<tr>
<td>Kernel code handle</td>
<td>64-bit</td>
<td>64-bit</td>
</tr>
<tr>
<td>Indirect function code handle</td>
<td>32-bit</td>
<td>64-bit</td>
</tr>
</tbody>
</table>

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 (on page 323).

For more information, see Chapter 16 Profiles (on page 327).

2.11 Race conditions

The program execution is undefined if it has a race condition.

A program has a race condition if:

- there are two memory accesses that are performed by different work-items or agent threads
- both access the same location in group segment, global segment, or image data memory,
at least one of the accesses is a store,
at least one of the accesses is not atomic, and
the accesses are unordered according to the memory ordering rules.

The memory ordering rules are defined in the HSA Platform System Architecture Specification Version 1.2,
Chapter 3 HSA Memory Consistency Model and section 2.15 Requirement: Images. Section 6.2.1 Memory order
(on page 183) defines how memory ordering is expressed in HSAIL.

### 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 (on page 246)), or a function call that can invoke
multiple functions (namely a switch call $scall$ or indirect call $icall$, see Chapter 10 Function Instructions
(on page 262)), 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 pseudo code as:

```c
if (condition) {
    // then statements
} else {
    // else statements
}
```

and might be translated into HSAIL as:

```hsail
// 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 (on page 46)). Communication operations comprise:

- atomic memory (see 6.5 Atomic memory instructions (on page 194))
- memfence (see 6.9 Memory fence (memfence) instruction (on page 207))
- signals (see 6.8 Notification (signal) instructions (on page 202))
- imagefence (see 7.6 Image fence (imagefence) instruction (on page 244))
- cross-lane (see 9.4 Cross-lane instructions (on page 259))
- barrier and wavebarrier (see 9.1 Barrier instructions (on page 248))
- fbarriers (see 9.2 Fine-grain barrier (fbarrier) instructions (on page 249))
- clock (see 11.4 Miscellaneous instructions (on page 283))
- clearedetectexcept, getdetectexcept and setdetectexcept (see 11.2 Exception instructions (on page 279))

or calls to functions that contain any of these (see Chapter 10 Function Instructions (on page 262))

In addition, control flow involving cross-lane instructions (see 9.4 Cross-lane instructions (on page 259)) 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 (on page 332)). 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 (on page 183).

### 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 machine 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 (on the facing 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 machine 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 (on page 28)) 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 (on 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 (on 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 machine 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 pseudo code (part of a reduction):

```c
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 segment memory allocation be a function of the wavefront size. The wavefront size for a particular kernel can be obtained by an HSA runtime query on the executable. 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:

```c
@x:  // all code that leaves the cbr must eventually reach @x
@y:  // and that code must reach @y
```

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

### 2.13 Forward progress

HSAIL implementations need to ensure forward progress. The forward progress guarantees for the execution of kernels are defined in the *HSA Platform System Architecture Specification Version 1.2*, section 2.10 *Requirement: Agent scheduling* and section 2.11 *Requirement: Kernel dispatch forward progress*.

Informally, any program can count on one-way communication such that:
- Work-group A can wait for values written by work-group B without deadlock provided either:
  - A and B belong to the same kernel dispatch, and A comes after B in work-group flattened ID order, or
  - A and B belong to different kernel dispatches in the same user mode queue, and the kernel dispatch of A comes after the kernel dispatch of B in packet ID order.
- In the base profile at least one work-group of at least one kernel dispatch of at least one user mode queue will make forward progress.
- In the full profile at least one work-group of at least one kernel dispatch will make forward progress independently for each user mode queue.

See 2.1 Overview of grids, work-groups, and work-items (on page 24), 2.2.2 Work-group flattened ID (on page 27), and Chapter 16 Profiles (on page 327).
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:

```c
__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:

```hsail
module &VectorAdd:1:1:$full:$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; // %if.end
  @BB0_1:
  ret;
  @BB0_2:
  // %if.then
  shl_u32 $s1, $s1, 2;
  add_u32 $s2, $s2, $s1;
  ld_global_f32 $s2, ($s2);
  add_u32 $s3, $s3, $s1;
  ld_global_f32 $s3, ($s3);
  add_f32 $s2, $s3, $s2;
  add_u32 $s0, $s0, $s1;
```
3.2 Transpose translated to HSAIL

The code below shows one way to write a transpose.

```hsail
module &Transpose:1:$full:$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:
    // BB0:
    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 (on page 337).

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 objects from HSAIL that can be executed on kernel agents. The life cycle is split into three stages:

- Finalization: Creates code objects from HSAIL. (See 4.2.1 Finalization (on the facing page).)
- Loading: Uses code objects to manage the allocation of global and readonly segment variables and installing of the machine code onto specific kernel agents. (See 4.2.2 Loading (on page 53).)
- Execution: Creates kernel dispatch packets that cause the execution of loaded machine code on a kernel agent, together with managing the allocation of associated resources such as group and private segment variables. (See 4.2.3 Execution (on page 55).)

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

Finalization and loading can be performed within the same application, or can be done by independent applications by saving the code objects. 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.
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 (on page 40)), profile (see Chapter 16 Profiles (on page 327)), and default floating-point rounding mode (see 4.19.2 Floating-point rounding (on page 118)) must be specified. All HSAIL modules have a module header (see Chapter 14 module Header (on page 323)) 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 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 (on page 323)). 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 (on page 106)).

The HSA runtime finalizer can be used to generate code objects from a program. At the time of finalization, all variables, fbarriers, 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 (on page 262).

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 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.

The HSA runtime finalizer can be used to generate two kinds of code objects: program code object and agent code object.

A program code object contains information about resources that are accessible by all kernel agents that execute the program. These include all defined program allocation global segment variables.

An agent code object contains information about resources that are only accessible by a single kernel agent that executes the program. These include all defined agent allocation global segment variables, readonly segment variables, and machine code for all the kernels and indirect functions defined in the modules added to the program for a specific instruction set architecture.

The default floating-point rounding mode affects agent code object generation as follows:

- If the program specifies the default floating-point rounding mode as default, then the default floating-point rounding mode of the instruction set architecture specified will be used. The instruction set architecture default floating-point rounding mode may be default, in which case the same agent code object can be executed with either zero or near default floating-point rounding mode.

- If the program specifies the default floating-point rounding mode as zero or near, then an error will be reported if the specified instruction set architecture does not support that default floating-point rounding mode.

- Otherwise, the default floating-point rounding mode of the program will be used.

Note that:

- The default floating-point rounding mode of the program, not that specified by the modules added to the program, is used. 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.
If the instruction set architecture default floating-point rounding mode is not default, then the agent code object generated always has a default floating-point rounding mode of zero or near, even if the default floating-point rounding mode of the program is default. In the case of a program with a default floating-point rounding mode of default, it is when the agent code object is generated for a specific instruction set architecture that the actual floating-point rounding mode used as the default is determined.

If the instruction set architecture default floating-point rounding mode is default, then the agent code object generated has a default floating-point rounding mode that matches the program default floating-point rounding mode, which can include default. If the agent code object has a default floating-point rounding mode of default, it is not until the agent code object is loaded into a specific kernel agent that the actual floating-point rounding mode used as the default is determined.

An HSA runtime query can be used to determine the instruction set architectures supported by the finalizer. A kernel agent can support multiple instruction set architectures. HSA runtime queries can be used to determine the instruction set architectures supported by a kernel agent and the properties of each instruction set architecture.

The HSA runtime finalizer may return an error if any kernel contained in the program requires a kernarg segment with a size that exceeds the limit of the instruction set architecture for which it is generating an agent code object. There is an HSA runtime query to obtain the kernarg segment size limit for an instruction set architecture. See HSA Platform System Architecture Specification Version 1.2, Appendix A Limits.

Once the program code object and agent 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 finalizer can generate code objects to a contiguous memory blob or optionally to a file. It can therefore be used to support both online and offline finalization, allowing caching of finalized results.

### 4.2.2 Loading

A code object can either be created directly using the HSA runtime finalizer into memory, or a previously saved code object can be loaded into memory or optionally accessed directly from a file.

To execute the kernels of a program, the code objects generated for the program must first be loaded into an executable. An application can use the HSA runtime to create zero or more executables.

When an executable is created, the profile and default floating-point rounding mode (either zero or near) must be specified.

The HSA runtime can be used to load at most one program code object, and at most one agent code object per kernel agent, into an executable. The instruction set architecture of an agent code object must match one of the instruction set architectures supported by the kernel agent.

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

The machine model of the code objects must match the address size used by the application (see Table 2-3 (on page 41)).

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

The extensions (including their version) used by the code objects must be supported by the HSA runtime, and for agent code objects must also be supported by the kernel agent. See 13.1 extension directive (on page 293). The HSA runtime can be used to query which extensions are supported.
The default floating-point rounding mode of the agent code objects either must match the default floating-point rounding mode of the executable or be default. In the latter case, the agent code object will be executed using the default floating-point rounding mode of the executable.

Note that an executable contains only agent code objects that use the same default floating-point rounding mode. Therefore, if there are kernel agents that do not support the same default floating-point rounding mode, it is necessary to use separate executables for their agent code objects.

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

All code objects 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 must not be destroyed until the executable has been destroyed. The code object and executable are not 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 HSA runtime may return an error if an agent code object contains a kernel that requires a kernarg segment with a size that exceeds the limit of the kernel agent on which it is being loaded for the instruction set architecture of the agent code object. This may happen when loading an agent code object into an executable, or when freezing an executable. There is an HSA runtime query to obtain the kernarg segment size limit for a kernel agent for a given instruction set architecture. See HSA Platform System Architecture Specification Version 1.2, Appendix A Limits.

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 (on page 106)) and allocation (see 4.3.10 Declaration and definition qualifiers (on page 73) and 6.2.5 Agent allocation (on page 185)) of the variable. For example:

- If a program code object is loaded into an executable, the program allocation global segment variables it defines will result in a single allocation that is shared by all kernel agents.

- If multiple agent code objects are loaded in the same executable for different kernel agents that define an agent allocation global segment variable or readonly segment variable, then each kernel 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 machine code has been allocated;

- the loaded machine code has been relocated to reference the variable allocations and external definitions;

- the machine code has been installed in the kernel agents on which it is loaded, copying to kernel
agent local memory if necessary;

- and any machine code instruction caches have been flushed.

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

### 4.2.3 Execution

Once an agent 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 executable for the specific kernel on the specific kernel agent. 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 fbarrings.

Any additional space required for dynamically allocated group segment memory must be added to this group segment static size (see 4.20 Dynamic group segment memory allocation (on page 123)).

The behavior is undefined if the group segment size specified in a kernel dispatch packet is smaller than required by the kernel's execution, or exceeds the amount of group segment memory available on the associated kernel agent to execute one work-group.

Specifying a group segment size in a kernel dispatch packet larger than required is allowed. However, it may result in reduced performance. For example, the number of work-groups that can execute concurrently may be reduced.

- 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 kernel dispatch packet must have the size of the call stack added to the reported private segment static size. It is implementation defined as to how to determine a suitable value to add and may depend on the data used by the particular kernel dispatch.

The behavior is undefined if the private segment size specified in a kernel dispatch packet is smaller than required by the kernel's execution, exceeds the amount of private segment memory available on the associated kernel agent to execute one work-group, or exceeds the amount of private segment memory supported by a work-item. See HSA Platform System Architecture Specification Version 1.2, Appendix A Limits.

Specifying a private segment size in a kernel dispatch packet larger than required is allowed. However, it may result in reduced performance. For example, the number of work-groups that can execute concurrently may be reduced, and memory access performance may be reduced due to reduced cache locality or increased page faults.

- The kernel code handle that specifies the machine code contained in an agent code object loaded on the kernel agent. It can be used for the kernel dispatch packet kernel object address field up until the executable is destroyed. A kernel code handle is an opaque 64-bit value for small and large machine model (see 2.9 Small and large machine models (on page 40)).

A kernel code handle is also available using a kernel integer symbolic expression constant (see 4.8.1 Integer constants (on page 86)).

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 kernel's kernarg segment.

The kernarg address in the kernel dispatch packet must be obtained as defined in 4.21 Kernarg segment (on page 125).

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.

Completion of a kernel dispatch packet's execution may be determined by optionally setting the completion signal in the kernel dispatch packet. Packets can specify execution ordering through the barrier field, and control memory visibility through the acquire and release fence fields. For more information, see HSA Platform System Architecture Specification Version 1.2, section 2.8 Requirement: User mode queuing and section 2.9 Requirement: Architected Queuing Language (AQL).

HSA runtime queries on an executable can be used to obtain an indirect function code handle for an indirect function in an agent code object loaded in an executable on a kernel agent. An indirect function code handle is also available using an indirect function integer symbolic expression constant (see 4.8.1 Integer constants (on page 86)). 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 (on page 40)).

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 (on page 271)). The icall instruction is not supported by the base profile (see 16.2.1 Base profile requirements (on page 328)).

The machine 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.
The machine code for indirect functions and the kernels that call them must be loaded in the same kernel agent of the same executable.

The machine code for kernels and indirect functions will remain available to execute until the HSA runtime is used to destroy the executable in which the machine code is loaded. All 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.

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 (on page 323).
A module directive can be the extension directive which must precede other HSAIL statements and applies to the whole module. See 13.1 extension directive (on page 293).

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

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

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.

For more information on location, pad, and pragma directives, see Chapter 13 Directives (on page 293).

### 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 (on page 106).

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**

```
kernel
delQual -> linkageQual -> kernelHeader -> ";"
linkageQual -> kernelHeader -> codeBlock -> ";"
```

**Figure 4–13 kernelHeader syntax diagram**

```
kernelHeader
"kernel" -> TOKEN_GLOBAL_IDENTIFIER -> kernFormalArgumentList
```

**Figure 4–14 kernFormalArgumentList syntax diagram**

```
kernFormalArgumentList
"(" -> kernFormalArgument -> ")
",” -> kernFormalArgument -> kernFormalArgument
```

**Figure 4–15 kernFormalArgument syntax diagram**

```
kernFormalArgument
optAlignQual -> "kemarg" -> dataTypeMod
TOKEN_LOCAL_IDENTIFIER -> optArrayDimension
```

### 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 (on page 106).

An indirect function has limitations on the symbols it can reference. See 10.8 Indirect call (icall) instruction (on page 271).

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.
For more information, see Chapter 10 Function Instructions (on page 262).
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

```syntax
signature
   "signature" TOKEN_GLOBAL_IDENTIFIER sigOutputFormalArgumentList sigInputFormalArgumentList `;`
```

Figure 4–23 sigOutputFormalArgumentList

```syntax
sigOutputFormalArgumentList
   sigFormalArgumentList
```

Figure 4–24 sigInputFormalArgumentList syntax diagram

```syntax
sigInputFormalArgumentList
   sigFormalArgumentList
```

Figure 4–25 sigFormalArgumentList syntax diagram

```syntax
sigFormalArgumentList
   "(" sigFormalArgument (".")
   sigFormalArgumentList
group
   ")")
```
For more information, see Chapter 10 Function Instructions (on page 262).

### 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 ({}).

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.

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

For more information on:
- Control directives, see Chapter 13 Directives (on page 293).
- Labels, see 4.9 Labels (on page 103).

### 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 (on page 263).

An arg block definition can be an arg block variable.

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 (on page 263).

4.3.7 Instruction

An instruction is an executable HSAIL statement.

The example below shows four instructions:

```hsail
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 (on page 113).

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 extension directive (on page 293).

For more information, see:

- Chapter 5 Arithmetic Instructions (on page 127)
- Chapter 6 Memory Instructions (on page 180)
- Chapter 7 Image Instructions (on page 209)
- Chapter 8 Branch Instructions (on page 246)
- Chapter 9 Parallel Synchronization and Communication Instructions (on page 248)
- Chapter 10 Function Instructions (on page 262)
- Chapter 11 Special Instructions (on page 276)
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 defines 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 \texttt{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 \texttt{4.6.2 Scope (on page 82)}.

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 (on page 106).

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 module variable name must be a global identifier.

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. A code block or arg block variable name must be a local identifier.

---

**Figure 4–34 moduleVariable syntax diagram**

```
moduleVariable

  optDeclQual → linkageQual → variable
```

**Figure 4–35 codeBlockVariable syntax diagram**

```
codeBlockVariable

  variable
```

**Figure 4–36 argBlockVariable syntax diagram**

```
argBlockVariable

  variable
```
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 (on page 23)).
- **Private**: Only allowed for code block variables. In addition, allowed for module variables as an experimental feature (see 1.3 HSAIL experimental features (on page 23)).
- **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 (on page 60) and 4.3.3 Function (on page 61) respectively.

The variable data type can be one of the data types described in 4.13 Data types (on page 108), except for b1.
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 (on page 40)).

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.

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 (on page 267).

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 (on page 103).
- Identifier scopes, see 4.6.2 Scope (on page 82).
- Variable storage duration, see 4.11 Storage duration (on page 105).

### 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 name must be a global identifier. A code block fbarrier name must be a local identifier.

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 (on page 106).

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 (on page 82).

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

```
moduleFbarrier
    ▸ optDeclQual ▸ linkageQual ▸ fbarrier ▸
```

Figure 4–42 codeBlockFbarrier syntax diagram

```
codeBlockFbarrier
    ▸ fbarrier ▸
```

Figure 4–43 fbarrier syntax diagram

```
fbarrier
    ▸ "fbarrier" ▸ nonRegisterIdentifier ▸ ":" ▸
```

For more information, see 9.2 Fine-grain barrier (fbarrier) instructions (on page 249).

### 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

```
optDeclQual
    ▸ declQual ▸
```

Figure 4–45 declQual syntax diagram

```
declQual
    ▸ "decl" ▸
```
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 fbaryers. If omitted:

- Kernels, functions, module variables, and module fbaryers default to module linkage.
- Code block variables, code block fbaryers, 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 (on page 106).

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 for each executable on which a program code object that defines the variable is loaded (see 4.2 Program, code object, and executable (on page 50)).

In HSAIL all references to the variable within a single executable 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 (on page 227) and 7.1.8 Sampler creation and sampler handles (on page 232)).

agent allocation

Causes the HSA runtime to perform a separate allocation for the variable for each kernel agent of each executable on which an agent code object that defines the variable is loaded (see 4.2 Program, code object, and executable (on page 50) and 6.2.5 Agent allocation (on page 185)). Each separate allocation will have a unique global segment address. The program execution is 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 \texttt{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 program execution is undefined if any other agent accesses that address. The variable's global segment address can be converted to a flat address, but the program execution is 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 (on page 227) and 7.1.8 Sampler creation and sampler handles (on page 232)).

\textit{automatic allocation}

Causes variables to be automatically allocated at the start of the variable's storage duration. See 4.11 Storage duration (on page 105).

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 \texttt{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, \texttt{global_f64 \&x[10]} must be aligned on a 64-bit (8-byte) boundary. For example, \texttt{align(8)}\texttt{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 \texttt{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. Kernel arg 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 (on page 105).

The program execution is 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 (on page 334).

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 (on page 82).

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 (on page 82).

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 (on page 82).

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 (on page 82).

The memory layout of variables in the kernarg segment is defined in 4.21 Kernarg segment (on page 125).
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 (on page 82).

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 (on page 82).

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 (on page 105) for a description of when storage is allocated for variables.

Also see:

- 7.1.7 Image creation and image handles (on page 227)
- 7.1.8 Sampler creation and sampler handles (on page 232)

Here is an example:

```hsail
function &fib(&&arg, &byte) (&&arg, &byte)
{
    private &&arg, byte;
    // allocate a private variable used internally

    ld_arg &&arg, [%byte];
    cmp_lt &&arg, %byte, %byte;  // if n < 3 go to return
    br %byte, @return;
    
    &&arg, %byte;
    &&arg, %byte;
    sub &&arg, %byte, %byte, 2;  // compute fib (n-2)
    st_arg &&arg, [%byte];
    call &fib(%byte);(%byte);  // store the result in %byte

    st_private &&arg, [%byte];
    
    &byte;
    &byte;
    sub &byte, %byte, %byte, 1;  // compute fib (n-1)
    st_arg &&arg, [%byte];
    call &fib(%byte);(%byte);  // store the result in %byte
    ld_arg &&arg, %byte;  // restore the result
}
```
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:

- TOKENCOMMENT (see 4.3.1 Annotations (on page 59))
- TOKENGLOBALIDENTIFIER (see 4.6 Identifiers (on page 81))
- TOKENLOCALIDENTIFIER (see 4.6 Identifiers (on page 81))
- TOKENLABELIDENTIFIER (see 4.6 Identifiers (on page 81))
- TOKENCREGISTER (see 4.7 Registers (on page 83))
- TOKENSREGISTER (see 4.7 Registers (on page 83))
- TOKENDREGISTER (see 4.7 Registers (on page 83))
- TOKENQREGISTER (see 4.7 Registers (on page 83))
- TOKENINTEGERLITERAL (see 4.8.1 Integer constants (on page 86))
- TOKENHALFLITERAL (see 4.8.2 Floating-point constants (on page 89))
- TOKENSINGLELITERAL (see 4.8.2 Floating-point constants (on page 89))
- TOKENDOUBLELITERAL (see 4.8.2 Floating-point constants (on page 89))
- TOKENWAVESIZE (see 2.6.2 Wavefront size (on page 31))
- TOKENSTRINGLITERAL (see 4.5 Strings (on the next page))
- Tokens for all the terminal symbols of the HSAIL syntax (see 19.2 HSAIL syntax grammar in Extended Backus-Naur Form (EBNF) (on page 406))

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) (on page 405).
4.5 Strings

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_LITERAL**.

### 4.6 Identifiers

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

**Figure 4–51** TOKEN_GLOBAL_IDENTIFIER syntax diagram

\[
\text{TOKEN_GLOBAL_IDENTIFIER} \\
\text{"\&" identifier}
\]

**Figure 4–52** TOKEN_LOCAL_IDENTIFIER syntax diagram

\[
\text{TOKEN_LOCAL_IDENTIFIER} \\
\text{"%" identifier}
\]

**Figure 4–53** TOKEN_LABEL_IDENTIFIER syntax diagram

\[
\text{TOKEN_LABEL_IDENTIFIER} \\
\text{"@" identifier}
\]

**Figure 4–54** identifier syntax diagram

\[
\text{identifier} \\
\text{[a-zA-Z_]} \rightarrow \text{[a-zA-Z0-9_]} \\
\]

#### 4.6.1 Syntax

Identifiers that are register names must start with a dollar sign ($). See **4.7 Registers (on page 83)**.

Identifiers that are labels must start with an at sign (@). See **4.9 Labels (on page 103)**.

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 starting with the characters __hsa are reserved for use by the HSA standard.

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 (on the 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 (on page 68).

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 (on page 263).

- 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

```
TOKEN_CREGISTER
  ("SC")"RegisterNumber"
```

Figure 4–56 TOKEN_SREGISTER syntax diagram

```
TOKEN_SREGISTER
  ("SS")"RegisterNumber"
```

Figure 4–57 TOKEN_DREGISTER syntax diagram

```
TOKEN_DREGISTER
  ("SD")"RegisterNumber"
```
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](image-url)
4.8 Constants

All constants must be compatible with the data type of the expected value according to the rules in Table 4-1 (on page 101). 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.
- Typed constant arguments: the expected value type is the type implied by the type of the typed constant.
- Instruction address expressions: the expected value type is an unsigned integer of the address size. See Table 2-3 (on page 41). 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

Integer constants are 64-bit unsigned values. They are only valid if the expected value type is an integer type, or a bit type less than or equal to 64 bits. The expected value 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. See 4.13.1 Base data types (on page 108)) and 4.8.5 How text format constants are converted to bit string constants (on page 101).

Note that 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:

```hsa
global_s32 %someident = 24;
add_s32 $s1, 24, 25;
```

An integer constant can be specified as an optionally signed integer literal, an integer symbolic expression, or as a null pointer address.
An integer constant can be specified as an integer literal. Some uses of integer literals allow an optional + and − sign before the integer literal. For −, the integer literal value is treated as a two’s complement 64-bit value and negated, regardless of whether the expected value type is a signed integer type, and the resulting bits used as the value. In the Extended Backus-Naur Form syntax, an integer literal is referred to as a TOKEN_INTEGER_LITERAL. Integer literals 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 4–64 (on the next page).
- A hexadecimal integer constant starts with 0x or 0X. See Figure 4–65 (on the next page).
- An octal integer constant starts with 0. See Figure 4–66 (on the next page).

In BRIG, the size of the data for an integer literal value must be the number of bytes needed by the expected value type. For b1, a single byte is used and must be 0 or 1.
An integer constant can also be specified as an integer symbolic expression constant which consists of “addr”, followed by a parenthesized global symbol identifier and optional signed integer literal displacement. The global symbol identifier can be that of a global segment variable, readonly segment variable, kernel, or indirect function. However, an agent allocation global segment variable, readonly segment variable, or indirect function is not allowed if the integer constant is used in a variable initializer for a program allocation global segment variable (see 4.3.10 Declaration and definition qualifiers (on page 73)). An integer symbolic expression constant is not allowed when the data type of the expected value is b1.

The value of a variable integer symbolic expression constant is the address of the variable denoted by the global symbol identifier plus the signed integer literal displacement performed using 64-bit two's complement arithmetic. In the small machine model the 32-bit address is zero extended to 64 bits (see 2.9 Small and large machine models (on page 40)). The address used is the address at which the variable was allocated when the code object containing the definition was loaded into an executable. For agent allocation global segment variables and readonly segment variables, the address corresponds to the variable allocation for the specific kernel agent specified when the agent code object was loaded into the executable. See 4.2.2 Loading (on page 53).
global_u64 &x[4];
global_u64 &x_p = addr(&x);
global_u64 &x_p4 = addr(&x + 4);
global_u64 &y1 = addr(&y1);
global_b8 &y2 = addr(&y2); // least significant 8 bits of address of &y2
mov_u64 $d0, addr(&x + 4);

The value of a kernel integer symbolic expression constant is a 64-bit kernel code handle of the kernel denoted by the global symbol identifier. A signed integer literal displacement is not allowed. The kernel code handle can be used in a kernel dispatch packet of a queue for the kernel agent that the agent code object containing the kernel definition was loaded up until the associated executable is destroyed. See 4.2.2 Loading (on page 53).

decl prog kernel &kern(kernarg_u64 %in, kernarg_u64 %res);
alloc(agent) global_u64 &k_p = addr(&kern);
mov_u64 $d0, addr(&kern);

The value of an indirect function integer symbolic expression constant is the indirect function code handle of the indirect function denoted by the global symbol identifier. A signed integer literal displacement is not allowed. In the small machine model the 32-bit indirect function code handle is zero extended to 64 bits (see 2.9 Small and large machine models (on page 40)). The indirect function code handle can be used in an icall instruction (see 10.8 Indirect call (icall) instruction (on page 271)) for machine code executing on the kernel agent in the executable that the agent code object containing the indirect function definition was loaded up until the associated executable is destroyed (see 4.2.2 Loading (on page 53)). The icall instruction is not supported by the base profile (see 16.2.1 Base profile requirements (on page 328)).

decl indirect function &ifunc() (arg_u64 %in, arg_u64 %res);
alloc(agent) global_u64 &if_p = addr(&ifunc);
mov_u64 $d0, addr(&ifunc); // $d0 can be used as code handle of icall instruction

Finally, an integer constant can be specified as the value of the null pointer address for either a flat address, a group segment address, a private segment address, or a kernarg segment address. The syntax is nullptr, nullptr_group, nullptr_private, and nullptr_kernarg respectively. However, nullptr_group, nullptr_private, and nullptr_kernarg are not allowed if the integer constant is used in a variable initializer for a program allocation global segment variable. A null pointer constant is not allowed when the data type of the expected value is b1. The value the null pointer constant corresponds to the value returned by the nullptr instruction (see 11.4 Miscellaneous instructions (on page 283)), zero extended to 64 bits if necessary. For the group, private and kernarg segments, the address corresponds to the null pointer address for the specific kernel agent specified when the code object was loaded into the executable.

global_u64 &f_p1 = nullptr;
global_u64 &f_p2 = u64(nullptr);
alloc(agent) global_u32 &g_p = nullptr_group;
alloc(agent) global_u32 &g_p = nullptr_private;
alloc(agent) global_u64 &k_p = nullptr_kernarg;
global_b16 &s1 = nullptr; // least significant 16 bits of flat null pointer
global_b16 &s2 = u16(nullptr); // least significant 16 bits of flat null pointer
mov_u32 $s0, nullptr_group;

4.8.2 Floating-point constants

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 (on page 101). In Extended Backus-Naur Form syntax, a half-precision float literal is referred to as a TOKEN_HALF_LITERAL.

• 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 (on page 101). In Extended Backus-Naur Form syntax, a single-precision float literal is referred to as a TOKEN_SINGLE_LITERAL.

• 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 (on page 101). In Extended Backus-Naur Form syntax, a double-precision float literal is referred to as a TOKEN_DOUBLE_LITERAL. 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 (on page 328)).

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 4–67 floatConstant syntax diagram

Figure 4–68 halfConstant syntax diagram
In BRIG, the size of the data for a floating-point literal value must be the number of bytes needed by the expected value type.

Floating-point literals 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 literal 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 literals, but is required for half-precision and single-precision decimal float literals. The decimal floating-point literal is converted to the memory representation using convert to nearest even (see 4.19.2 Floating-point rounding (on page 118)) without flushing subnormal values to zero (see 4.19.3 Flush to zero (ftz) (on page 119)) even in base profile (see 16.2.1 Base profile requirements (on page 328)). See Figure 4-74 (on page 93).

- A hexadecimal floating-point literal 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 literals, but is required for half-precision and single-precision hexadecimal float literals. The hexadecimal floating-point literal is converted to the memory representation using convert to nearest even (see 4.19.2 Floating-point rounding (on page 118)).
representation using convert to nearest even (see 4.19.2 Floating-point rounding (on page 118)) without flushing subnormal values to zero (see 4.19.3 Flush to zero (ftz) (on page 119)) even in base profile (see 16.2.1 Base profile requirements (on page 328)). See Figure 4–75 (on the facing page).

- An IEEE/ANSI Standard 754-2008 binary interchange double-precision floating-point literal begins with 0d or 0D followed by 16 hexadecimal digits. A single-precision floating-point literal begins with 0f or 0F followed by eight hexadecimal digits. A half-precision floating-point literal begins with 0h or 0H followed by four hexadecimal digits.

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

Figure 4–71 TOKEN_HALF_LITERAL syntax diagram

Figure 4–72 TOKEN_SINGLE_LITERAL syntax diagram
Chapter 4. HSAIL Syntax and Semantics  4.8 Constants

Figure 4–73 TOKEN_DOUBLE_LITERAL syntax diagram

Figure 4–74 decimalFloatLiteral syntax diagram

Figure 4–75 hexFloatLiteral syntax diagram
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.
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.

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 (f64) is not supported by the base profile (see 16.2.1 Base profile requirements (on page 328)).
For information on packed typed constants, see 4.14.2 Packed typed constants (on page 112).

For information on image handle typed constants, see 7.1.7 Image creation and image handles (on page 227). 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 13.4 pragma directive (on page 297)).

For information on sampler handle typed constants, see 7.1.8 Sampler creation and sampler handles (on page 232). 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 13.4 pragma directive (on page 297)).

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

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 (on page 101). 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

arrayTypedConstant

- integerArrayTypedConstant
- doubleArrayTypedConstant
- singleArrayTypedConstant
- halfArrayTypedConstant
- packedArrayTypedConstant
- imageArrayTypedConstant
- samplerArrayTypedConstant
- signalArrayTypedConstant

Figure 4–84 integerArrayTypedConstant syntax diagram

integerArrayTypedConstant

- integerType
- "[" + integerConstant + "]"
- integerTypedConstant

integerConstant

integerTypedConstant

"("
4.8.4 Aggregate constants

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

```
global_s8 &ar1[] = s32[](0, -1, nullptr, addr(&ar1 + 8));
global_s8 &ar2[] = s8[](0, -1, s8(nullptr), s8(addr(&ar2 + 2)));  
```
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: \texttt{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. \(n\) is treated as a \(u64\) type and valid values 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.
Finally, an aggregate constant element can be a zero request: \texttt{zero(n)}. This causes exactly \( n \) zero bytes to be generated. \( n \) is treated as a \texttt{u64} type and can be 0 in which case no bytes are generated. A zero request is more compact to express in HSAIL and BRIG than the corresponding sequence of 0 literal values. On some implementations, it may result in a more compact code object if used to specify the entire variable initializer as it allows the variable to be placed in the (zeroed) uninitialized data section.

\begin{verbatim}
aggregateConstantZero

"zero" "(" TOKEN_INTEGER_LITERAL ")" 

\end{verbatim}

global_b16 \&ag1[2] = {s8(0), u16(1), s8(2)};
global_b8 \&ag2[] = {u16(1), align(8), sig64(0), zero(128)};
global_b8 \&ag3[] = {s8(0), u16[]{1, 2, 3, 4, u16(9), f16(2.0h)};
global_b32 \&ag4[] = {u32 (nullptr), u32 (addr (\&ag4))};
\end{verbatim}

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>
<thead>
<tr>
<th>Kind of text format constant provided (see 4.8 Constants (on page 85))</th>
<th>Data type of expected value (see 4.13 Data types (on page 108))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer constant</td>
<td>Bit type</td>
</tr>
<tr>
<td>Float typed constant</td>
<td>Length-only rule</td>
</tr>
<tr>
<td>Integer typed constant</td>
<td>Length-only rule</td>
</tr>
<tr>
<td>Packed typed constant</td>
<td>Length-only rule</td>
</tr>
</tbody>
</table>

Table 4-1 Text constants and results of the conversion

Chapter 4. HSAIL Syntax and Semantics  4.8 Constants
<table>
<thead>
<tr>
<th>Kind of text format constant provided (see 4.8 Constants (on page 85))</th>
<th>Data type of expected value (see 4.13 Data types (on page 108))</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bit type</td>
</tr>
<tr>
<td>Image typed constant</td>
<td>Error</td>
</tr>
<tr>
<td>Sampler typed constant</td>
<td>Error</td>
</tr>
<tr>
<td>Signal typed constant</td>
<td>Error</td>
</tr>
<tr>
<td>Array typed constant</td>
<td>Error</td>
</tr>
<tr>
<td>Aggregate constant</td>
<td>Error</td>
</tr>
</tbody>
</table>

Truncation for an integer value in the text is as follows: the value is input as 64 bits in 2s compliment, 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, 0xfffffffff; // Legal: 9 f's stored as 0x0000000000000000f
```

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 0x0000000000000000f.

```
add $s32, $s0, 0xfffffffff; // 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, 0xfffffffff; // Illegal: integer constant is evaluated as 64 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 (on page 31).

4.9 Labels

Label identifiers consist of an at sign (@) followed by the name of the identifier (see 4.6 Identifiers (on page 81)).

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

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

Label identifiers cannot appear in an address expression.

See Chapter 8 Branch Instructions (on page 246).

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.

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 (on page 50)), 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 program execution is undefined if a kernel dispatch does not use appropriate memory synchronization to access the variable after it has been initialized.
A readonly segment variable has agent allocation, and so has distinct memory allocations for each agent (see 4.3.10 Declaration and definition qualifiers (on page 73)). 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 program execution is 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 (on page 85)):

- 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 (on page 101) 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 (on page 101) 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 (on page 101).

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 (on page 227) and 7.1.8 Sampler creation and sampler handles (on page 232).

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

```c
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(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, 31, 30, 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[] = {-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 &sl = sig64(0);  // Signal handles should only be initialized with 0.
global_sig64 &sa[2] = sig64[](sig64(0), sig64(0x000));
global_sig64 &se[] = sig64[](sig64(0), sig64(0x000), 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 code objects that have been finalized from an HSAIL program that includes an HSAIL module containing the definition are loaded into an HSA runtime executable and lasts until the executable is destroyed (see 4.2 Program, code object, and executable (on page 50)). 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 segment memory allocation (on page 123)).

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 (on page 50) and 4.3 Module (on page 57).

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 (on page 82).

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 runtime 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 (on page 50)). For agent allocation variables, it is required to define the name for each agent onto which the agent code object is loaded (see 6.2.5 Agent allocation (on page 185)).

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:

```c
// 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:

```c
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:

```hsail
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:

```hsail
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:

```hsail
// 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

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Possible lengths in bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>Bit type</td>
<td>1, 8, 16, 32, 64, 128</td>
</tr>
<tr>
<td>s</td>
<td>Signed integer type</td>
<td>8, 16, 32, 64</td>
</tr>
<tr>
<td>u</td>
<td>Unsigned integer type</td>
<td>8, 16, 32, 64</td>
</tr>
<tr>
<td>f</td>
<td>Floating-point type</td>
<td>16, 32, 64</td>
</tr>
</tbody>
</table>

A *compound type* is made up of a base data type and a length.
The 64-bit floating-point type ($\mathbf{f64}$) is not supported by the base profile (see 16.2.1 Base profile requirements (on page 328)). 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 (\texttt{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 $\times$ followed by a count. For example, $8\times4$ 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

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Lengths for 32-bit types</th>
<th>Lengths for 64-bit types</th>
<th>Lengths for 128-bit types</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>Signed integer</td>
<td>$8\times4$, $16\times2$</td>
<td>$8\times8$, $16\times4$, $32\times2$</td>
<td>$8\times16$, $16\times8$, $32\times4$, $64\times2$</td>
</tr>
<tr>
<td>u</td>
<td>Unsigned integer</td>
<td>$8\times4$, $16\times2$</td>
<td>$8\times8$, $16\times4$, $32\times2$</td>
<td>$8\times16$, $16\times8$, $32\times4$, $64\times2$</td>
</tr>
<tr>
<td>f</td>
<td>Floating-point</td>
<td>$16\times2$</td>
<td>$16\times4$, $32\times2$</td>
<td>$16\times8$, $32\times4$, $64\times2$</td>
</tr>
</tbody>
</table>

32-bit sizes are:

- $8\times4$ — four bytes; can be used with $s$ and $u$ types
- $16\times2$ — two shorts or half-floats; can be used with $s$, $u$, and $f$ types

64-bit sizes are:

- $8\times8$ — eight bytes; can be used with $s$ and $u$ types
- $16\times4$ — four shorts or half-floats; can be used with $s$, $u$, and $f$ types
- $32\times2$ — two integers or floats; can be used with $s$, $u$, and $f$ types

128-bit sizes are:

- $8\times16$ — 16 bytes; can be used with $s$ and $u$ types
- $16\times8$ — eight shorts or half-floats; can be used with $s$ or $u$, and $f$ types
- $32\times4$ — four integers or floats; can be used with $s$, $u$, and $f$ types
- $64\times2$ — two 64-bit integers or two doubles; can be used with $s$, $u$, and $f$ types

The 64-bit floating-point packed type ($\mathbf{f64x2}$) is not supported by the base profile (see 16.2.1 Base profile requirements (on page 328)). 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

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

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 (on page 294)). 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 (on page 40)). This includes segment variable declarations, segment variable definitions and instructions.

For more information see:
- 7.1.7 Image creation and image handles (on page 227)
- 7.1.8 Sampler creation and sampler handles (on page 232)
- 6.8 Notification (signal) instructions (on page 202)

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 (on the 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

<table>
<thead>
<tr>
<th>Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>The single source is treated as packed. The instruction is applied to each element separately.</td>
</tr>
<tr>
<td>p_sat</td>
<td>Same as p, except that each result is saturated. (Cannot be used with floating-point values.)</td>
</tr>
<tr>
<td>a</td>
<td>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.</td>
</tr>
<tr>
<td>a_sat</td>
<td>Same as a, except that the result is saturated. (Cannot be used with floating-point values.)</td>
</tr>
</tbody>
</table>

Table 4-6 Packing controls for instructions with two source inputs

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

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 typed constants

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 (on page 89). The 64-bit packed floating-point type (\( f 64x2 \)) is not supported by the base profile (see 16.2.1 Base profile requirements (on page 328)).

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 (on page 101).

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

```hsail
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, 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_ps_s8x4 $s1, $s2, $s3;
```

\[ s1 = \text{the logical OR of:} \]
\[ s2[0-7] + s3[0-7] \]
\[ s2[8-15] + s3[8-15] \]
\[ s2[16-23] + s3[16-23] \]
\[ s2[24-31] + s3[24-31] \]

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 \( \text{b8}, \text{b16}, \text{s8}, \text{s16}, \text{u8}, \text{and u16} \) 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 (on page 101) based on the type of the operand. Note that image, sampler, and array typed constants are not allowed. See 4.8 Constants (on page 85) and 2.6.2 Wavefront size (on page 31).

- Memory, image, segment checking, segment conversion, and \( \text{lda} \) instructions take an address expression as a source operand. See 4.18 Address expressions (on page 115).

- 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 (on page 115).

- Branch instructions can take a label and list of labels as a source operand. See Chapter 8 Branch Instructions (on page 246).
Call instructions can take a function identifier, list of function identifiers, and signature identifier as a source operand. See Chapter 10 Function Instructions (on page 262).

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 (on page 108). 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 (on page 41).

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 (on page 101).

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 (on page 118).

- If the operand is an integer type 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 operation is performed and the size of the operand compound type dictates the number of least significant bits that are used as the result. 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 bl 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 bl value or conversion to a signed or unsigned integer value or a float value. See 5.19 Conversion (cvt) instruction (on page 171).

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 (on page 101).

In BRIG, the type of the vector operand is the type of each component operand. See 4.16 Operands (on page 113).

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:

```hsail
  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 (on page 187).

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 (on page 68).

Addresses are always in bytes. For information about how addresses are formed from an address expression, see 6.1.1 How addresses are formed (on page 180).

Some examples of addresses are:

```hsail
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+4}; // group segment-relative address
ld_group_f16 $s5, [100];      // read 16 bits at absolute global segment address 100
```

See 6.3 Load (ld) instruction (on page 187).

### 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 (on page 328)) has been specified:
  - The 64-bit floating-point type (float 64) is not supported (see 16.2.1 Base profile requirements (on page 328)).
  - The DETECT and BREAK exception policies are not supported for the five floating point exceptions specified in 12.2 Hardware exceptions (on page 288), therefore instructions do not have to generate them as they have no observable effect (see 4.19.5 Floating-point exceptions (on page 121)).

Floating-point values are stored in IEEE/ANSI Standard 754-2008 binary interchange format encoding. See 4.19.1 Floating-point numbers (on the next 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 (on page 288).
  - 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 (on page 279).

Four IEEE/ANSI Standard 754-2008 rounding modes are supported for some floating-point instructions. See 4.19.2 Floating-point rounding (on the next page).

The ftz (flush to zero) modifier, which forces subnormal values to zero, is supported on most instructions. See 4.19.3 Flush to zero (ftz) (on page 119).

Instructions that produce NaN results have certain requirements. See 4.19.4 Not a Number (NaN) (on page 120).

Some instructions are fast approximations (the nsqrt instruction is an example). See 5.14 Native floating-point instructions (on page 159).

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 (on page 175).

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 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) (on page 121).
### 4.19.1 Floating-point numbers

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

- An \( f16 \) number is stored in memory 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 \( f16 \) value stored in an \( s \) register occupies the least significant 16 bits of the register, and the most significant 16 bits are undefined.
- An \( f32 \) number is stored in memory and in an \( s \) register as 1 bit of sign, 8 bits of exponent, and 23 bits of mantissa. The exponent is biased with an excess value of 127.
- An \( f64 \) number is stored in memory and in a \( d \) register as 1 bit of sign, 11 bits of exponent, and 52 bits of mantissa. The exponent is biased with an excess value of 1023.

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

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

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

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

Signaling NaNs are never the result of arithmetic 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 toward positive infinity.
- **down** specifies that the result of the instruction should be rounded toward negative infinity.
- **zero** specifies that result of the instruction should be rounded toward zero.
- **near** specifies that result of the instruction should be rounded toward 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 (on page 323) and 16.2.1 Base profile requirements (on page 328).
Floating-point operations that support the rounding modifier first compute the infinitely accurate 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 (on page 288)):

- 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) (on the next 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 finite numbers then the overflow and inexact exceptions are generated. The destination is set to either an appropriately signed infinity or appropriately signed largest representable finite number according to the rounding mode. See 5.19.5 Description of floating-point rounding modes (on page 177).
- Else if tininess is detected and \texttt{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 (see 5.19.5 Description of floating-point rounding modes (on page 177). 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 (\texttt{ftz})

HSAIL supports the flush to zero \texttt{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 \texttt{ftz} modifier then:

- If the base profile has been specified then the \texttt{ftz} modifier must be specified. See 16.2.1 Base profile requirements (on page 328).
- Otherwise, the \texttt{ftz} modifier is optional.

If \texttt{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 with the same sign 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 \texttt{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 (on the previous page)), then the
destination operand must be set to 0.0 with the same sign 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 (on page 118).

### 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 (on page 157)) and native floating-point instructions (see 5.14 Native floating-point instructions (on page 159)):

- 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 (on page 166)) 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 (on page 288).
- 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 (on page 328).
- 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 (on page 329).
  - 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 smaller floating-point type to a larger one and then back to the original smaller 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 have some exceptions to these rules, both when converting component values (see 7.1.4.2 Channel type (on page 216)), and when using a sampler with normalized coordinates (see 7.1.6.1 Coordinate normalization mode (on page 222)) or a linear filter (see 7.1.6.3 Filter mode (on page 225)).
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 (on page 288)). It also provides a mechanism to control these exceptions by means of the DETECT and BREAK exception policies (see 12.3 Hardware exception policies (on page 290)). The exception policies are specified when a kernel is finalized and cannot be changed at runtime (see 13.5 Control directives for low-level performance tuning (on page 317)). 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 (on page 328)).

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 instructions that return floating-point values have an accuracy defined in terms of ULP rather than in terms of the correctly rounded result specified by IEEE/ANSI Standard 754-2008. In addition, the accuracy of some instructions varies according to the profile specified. See 16.2 Profile-specific requirements (on page 328).

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 is defined in terms of a floating-point representation rep which specifies an ordered list \( F_{rep} \) of consecutive floating-point numbers that can be represented. These values are termed the finite numbers of the representation. For IEEE/ANSI Standard 754-2008 representations, \( F_{rep} \) comprises:

- all finite normal and subnormal floating-point numbers,
- does not include plus infinity and minus infinity, and
- only includes a single value for 0.0.

ULP is defined as follows:

\[
\text{ulp}_{rep}(\text{expected}) = |b - a|
\]

where the calculation is performed with infinite accuracy and:

- rep is a floating-point representation.
- expected is an infinitely accurate numeric value. Note that the distinguished values of plus infinity, minus infinity, and NaN are not allowed.
- If expected lies between two finite consecutive floating-point numbers in \( F_{rep} \), without being equal to one of them, then let \( a \) and \( b \) be those numbers. Note that \( a \) and \( b \) may not be the nearest finite floating-point numbers due to changes in the spacing of the finite numbers in \( F_{rep} \).
- Otherwise, let \( a \) and \( b \) be the two non-equal finite floating-point numbers nearest \( x \) in \( F_{rep} \). Note that either expected equals \( a \) or \( b \), or expected is greater/less than the largest/smallest finite number in \( F_{rep} \) and both \( a \) and \( b \) are less/greater than expected with one of them being the largest/smallest finite number in \( F_{rep} \).

The relative error of a computation is expressed in terms of ULP and is defined as follows:

\[
\text{ulp}_{error}_{rep}(\text{actual}, \text{expected})
\begin{align*}
&\text{if } (\text{actual}_\text{is_nan} \&\& \text{expected}_\text{is_nan}) \text{ return } 0.0; \\
&\text{if } (\text{actual}_\text{is_nan} \| \text{expected}_\text{is_nan}) \text{ return } +\infty; \\
&\text{if } (\text{actual}_\text{is_plus_infinity} \&\& (\text{expected}_\text{is_plus_infinity} \| (\text{expected} > \text{rep}_\text{max_float}))) \text{ return } 0.0;
\end{align*}
\]
if (actual_is_minus_infinity && (expected_is_minus_infinity || (expected < -rep_max_float))) return 0.0;
if ((actual_is_plus_infinity || actual_is_minus_infinity) && (expected_is_plus_infinity || expected_is_minus_infinity))
return ±∞;
if (actual_is_finite && (expected <= rep_max_float))
    return |actual - expected| / ulp(rep(expected));
if (actual_is_finite && (expected_is_plus_infinity || expected > rep_max_float))
    return (|actual - rep_max_float| / ulp(rep(rep_max_float))) + 1;
if (actual_is_finite && (expected_is_minus_infinity || expected < -rep_max_float))
    return (|actual - (-rep_max_float)| / ulp(rep(-rep_max_float))) + 1;
if (actual_is_plus_infinity) return (|rep_max_float - expected| / ulp(rep(expected))) + 1;
if (actual_is_minus_infinity) return (|-rep_max_float - expected| / ulp(rep(expected))) + 1;

where all calculations are performed with infinite accuracy and:

- **rep** is a floating point representation.
- **actual** is a finite numeric value specified by rep or the distinguished value of plus infinity, minus infinity, or NaN.
- **Expected** is an infinitely accurate numeric value or the distinguished value of plus infinity, minus infinity, or NaN.

The implementation of an instruction meets a maximum relative error bound of \( n \) ULP if, for all possible source values of the instruction, either:

\[
\text{round}_\text{rep}(\text{expected}) = \text{actual}
\]
or:

\[
\text{ulp}_\text{error}_\text{rep}(\text{actual}, \text{expected}) \leq n
\]

where:

- **rep** is the representation of the result of the instruction.
- **actual** is the result returned by the instruction’s implementation. It is either a finite numeric value specified by **rep** or the distinguished values plus infinity, minus infinity, and NaN. For IEEE/ANSI Standard 754-2008 representations both representations of zero are treated as 0.0, and all NaN values are treated the same.
- **expected** is the infinitely accurate result of the instruction’s definition without performing any floating-point rounding of the result. This can include the distinguished values plus infinity, minus infinity, and NaN. Except, if the operation specifies the \( \text{ftz} \) modifier (see 4.19.3 *Flush to zero (ftz)* (on page 119)):
  - If a source operand has a subnormal floating-point value according to the representation specified by the instruction for that operand, then the value 0.0 is used for that operand when computing **expected**.
  - If the infinitely accurate result is a tiny non-zero value according to the implementation definition of tininess performed with infinite accuracy for the floating-point rounding mode and result representation specified by the instruction (see 4.19.2 *Floating-point rounding* (on page 118)), then 0.0 is returned for **expected**.
- \( \text{round}_\text{rep}(\text{expected}) \) converts the infinitely accurate **expected** to either a finite numeric value specified by **rep** or the distinguished value plus infinity, minus infinity, or NaN. The conversion is done with infinite accuracy for the floating-point rounding mode and result representation specified by the instruction (see 4.19.2 *Floating-point rounding* (on page 118)). For IEEE/ANSI Standard 754-2008...
representations both representations of zero are treated as 0.0, and all NaN values are treated the same.

### 4.20 Dynamic group segment memory allocation

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

```hsail
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 segment 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 machine 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 segment memory can be specified by increasing the value of the group segment size specified in a kernel dispatch packet (see **4.2.3 Execution (on page 55)**). It can be accessed in a kernel using the `groupbaseptr`, `groupstaticsize`, and `grouptotalsize` instructions (see **11.4 Miscellaneous instructions (on page 283)**) and aligned using the `requiredgroupbaseptralign` control directive (see **13.5 Control directives for low-level performance tuning (on page 317)**).

Dynamic group segment memory can be accessed directly in the kernel using these steps:

1. The kernel specifies the alignment required for the group segment base address using the `requiredgroupbaseptralign` control directive. The group segment static size is obtained by the `groupstaticsize` instruction, rounded up to the required alignment, and added to the group segment base address returned by the `groupbaseptr` instruction to obtain the aligned base of the dynamic group segment memory. The base of each dynamic variable can then be computed by adding its aligned offset within the dynamically allocated space.

2. The finalizer determines the amount of group segment memory used by the group segment variables accessed by each kernel and the functions they call directly or indirectly, and records this static size in the agent code object generated. The finalizer also uses this information to generate
machine code for the \texttt{groupstaticsize} instruction.

3. The application obtains the group segment static size using HSA runtime queries on the executable for the specific kernel. The application dispatches the kernel, and specifies the amount of group segment memory as the sum of the static size (rounded up to the alignment of the dynamic group segment variables) plus the amount required for the dynamic group segment memory.

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

```hsail
kernel &k1(kernarg_u32 %dynamic_size)
{
    requiredgroupbaseptralign 4;
    group_u32 %known[2];
    group_u32 %more_known[4];
    groupbaseptr_u32 $a0;
    groupstaticsize_u32 $s1;
    add_u32 $s1, $s1, 3; // Align group segment static size to 4.
    add_u32 $s2, $s0, $s1; // Aligned base of dynamic group segment.
    ld_kernarg_u32 $s3, [%dynamic_size];
    add_u32 $s3, $s3, $s2;
    st_group_f32 1.0f, [$s2 + 8];
    st_group_f32 2.0f, [$s3];
    //...
}
```

Dynamic group segment memory can be accessed using kernel arguments using these steps:

1. The application declares the HSAIL kernel with additional arguments, which are group segment offsets for the dynamically sized group segment memory. The kernel adds these offsets to the group segment base address returned by the \texttt{groupbaseptr} instruction, and uses the result to access the dynamically sized group segment memory. The kernel specifies the alignment required for the group segment base address using the \texttt{requiredgroupbaseptralign} control directive.

2. The finalizer determines the amount of group segment memory used by the group segment variables accessed by each kernel and the functions they call directly or indirectly, and records this static size in the agent code object generated.

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 static size. The group segment static size is obtained using HSA runtime queries on the executable for the specific 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 segment memory as the sum of the static size plus the amount required for the dynamic group segment memory.

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

```hsail
kernel &k1(kernarg_u32 %dynamic_offset, kernarg_u32 %more_dynamic_offset)
{
    requiredgroupbaseptralign 4;
    group_u32 %known[2];
    group_u32 %more_known[4];
    groupbaseptr_u32 $a0;
    ld_kernarg_u32 $s1, [%dynamic_offset];
    add_u32 $s1, $s0, $s1;
    st_group_f32 1.0f, [$s2 + 8];
    st_group_f32 2.0f, [$s3];
    //...
}
```
ld_kernarg_u32 $s2, [%more_dynamic_offset];
add_u32 $s2, $s0, $a2;
st_group_f32 1.0f, [$s1 + 8];
st_group_f32 2.0f, [$s2];
//...
};

The amount of dynamic group segment memory available to the work-group of an executing kernel dispatch work-item can be determined by subtracting the static size obtained using the groupstaticsize instruction from the total size obtained from the grouptotalsize instruction. This may be useful for applications that determine the dynamic group segment size at kernel dispatch time based on the limits supported by the kernel agent.

### 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 (on page 73).

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. This is required to support allocations at least as large as the maximum kernarg segment size limit of the instruction set architectures supported by kernel agent.

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 (on page 68)).

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 \( \$s1 \):

```hsail
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:

```hsail
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 (on page 187) and 5.8 Copy (move) instructions (on page 141).
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 (on page 113)).

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 (on page 113).

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.

5.2.1 Syntax

Table 5–1 Syntax for integer arithmetic instructions

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs_sLength</td>
<td>dest, src0</td>
</tr>
<tr>
<td>add_sat_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>borrow_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>carry_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>div_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>max_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>min_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>mul_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>mulhi_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>neg_sLength</td>
<td>dest, src0</td>
</tr>
<tr>
<td>rem_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>sub_sat_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
</tbody>
</table>
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(−2^{31})` returns $−2^{31}$ for 32-bit operands. `abs(−2^{63})` returns $−2^{63}$ for 64-bit operands.
add

The `add` instruction computes the sum of the two sources `src0` and `src1`, with optional saturation, and stores the result into the destination `dest`.

The `add` instruction supports both signed and unsigned forms, which affect the result if saturation is specified, otherwise both forms compute the same result and simply aid readers of the code.

The non-saturating form might execute faster than the saturating form.

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.

The result of the remainder of `-2^{31}` for `s32` types, or `-2^{63}` for `s64` types, divided by `-1` is undefined, and it is implementation defined whether no exception is generated or an implementation defined exception is generated.

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.

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

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 \times 1`, the upper 32 bits of the signed integer product are all 1's while the upper 32 bits of the unsigned product are all 0's.

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

To generate a 128-bit product from 64-bit sources, compilers can generate both 64-bit half results using `mul_u64/mul_s64` and `mulhi_u64/mulhi_s64` and then combine the partial results using a `combine` instruction. See 5.8 Copy (move) instructions (on page 141).

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_length` is valid.

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

sub

The `sub` instruction subtracts source `src1` from source `src0`, with optional saturation, and places the result in the destination `dest`.

The `sub` instruction supports both signed and unsigned forms, which affect the result if saturation is specified, otherwise both forms compute the same result and simply aid readers of the code.

The non-saturating form might execute faster than the saturating form.
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;
add_sat_s8 $s1, 42, $s2;
add_sat_u8 $s1, $s2, 0x23;
add_sat_s16 $s1, 42, $s2;
add_sat_u16 $s1, $s2, 0x23;
add_sat_s32 $s1, $s2;
add_sat_u32 $s1, $s2, 0x23;
add_sat_s64 $d1, $d2, 0x23;
add_sat_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;
sub_sat_s8 $s1, 100, 10;
sub_sat_u8 $s1, $s2, 0x23;
sub_sat_s16 $s1, 100, 10;
sub_sat_u16 $s1, $s2, 0x23;
sub_sat_s32 $s1, 100, 10;
sub_sat_u32 $s1, $s2, 0x23;
```
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 (on the facing page).

5.3.1 Syntax

Table 5–3 Syntax for integer optimization instruction

<table>
<thead>
<tr>
<th>Opcode and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>mad_sat_TypeLength</td>
<td>dest, src0, src1, src2</td>
</tr>
</tbody>
</table>

Explanation of Modifiers (see Table 4–2 (on page 108))

sat: Optional. If present, then if the result value is greater than or less than the range of TypeLength it is set to the maximum or minimum representable value of TypeLength respectively. The result is sign-extended if Type is s, and zero-extended if Type is u, to the size of the dest register. See 4.16.2 Rules for operand registers (on page 114).

Type: s, u.

Length: 8, 16, 32, 64. 8 and 16 are only allowed if sat is specified.

Explanation of Operands (see 4.16 Operands (on page 113))

dest: Destination register.

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

Exceptions (see Chapter 12 Exceptions (on page 288))

No exceptions are allowed.

For BRIG syntax, see 18.7.1.2 BRIG syntax for integer optimization instruction (on page 393).
5.3.2 Description

The integer `mad` (multiply add) instruction multiplies source `src0` times source `src1` and then adds source `src2`, with optional saturation, and stores the result in the destination `dest`.

Integer `mad` supports both signed and unsigned forms, which affect the result if saturation is specified, otherwise both forms compute the same result and simply aid readers of the code.

The non-saturating form might execute faster than the saturating form.

Examples

```assembly
mad_s32 $s1, $s2, $s3, $s5;
mad_s64 $d1, $d2, $d3, $d2;
mad_u32 $s1, $s2, $s3, $s3;
mad_u64 $d1, $d2, $d3, $d1;
mad_sat_s8 $s1, $s2, $s3, $s5;
mad_sat_s16 $s1, $s2, $s3, $s5;
mad_sat_s32 $s1, $s2, $s3, $s5;
mad_sat_s64 $d1, $d2, $d3, $d2;
mad_sat_u8 $s1, $s2, $s3, $s5;
mad_sat_u16 $s1, $s2, $s3, $s5;
mad_sat_u32 $s1, $s2, $s3, $s5;
mad_sat_u64 $d1, $d2, $d3, $d2;
```

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 (on the previous page).

5.4.1 Syntax

Table 5–4 Syntax for 24-bit integer optimization instructions

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mad24_TypeLength</code></td>
<td><code>dest, src0, src1, src2</code></td>
</tr>
<tr>
<td><code>mad24hi_TypeLength</code></td>
<td><code>dest, src0, src1, src2</code></td>
</tr>
<tr>
<td><code>mul24_TypeLength</code></td>
<td><code>dest, src0, src1</code></td>
</tr>
<tr>
<td><code>mul24hi_TypeLength</code></td>
<td><code>dest, src0, src1</code></td>
</tr>
</tbody>
</table>

**Explanation of Modifiers (see Table 4–2 (on page 108))**

- Type: `s`, `u`
- Length: `32`
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

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>shl_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>shr_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
</tbody>
</table>
### Explanation of Modifiers (see Table 4-2 (on page 108))

<table>
<thead>
<tr>
<th>Type: a, u</th>
</tr>
</thead>
</table>

**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 (on page 113))

<table>
<thead>
<tr>
<th>dest: Destination register.</th>
</tr>
</thead>
<tbody>
<tr>
<td>src0, src1: Sources. Can be a register or immediate value. Regardless of <strong>TypeLength</strong>, src1 is always u32.</td>
</tr>
</tbody>
</table>

### Exceptions (see Chapter 12 Exceptions (on page 288))

| No exceptions are allowed. |

For BRIG syntax, see 18.7.1.4 BRIG syntax for integer shift instructions (on page 394).

#### 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

- \(\text{shl}_u32\) \(\$s1, \$s2, 2\);
- \(\text{shl}_u64\) \(\$d1, \$d2, 2\);
- \(\text{shl}_s32\) \(\$s1, \$s2, 2\);
- \(\text{shl}_s64\) \(\$d1, \$d2, 2\);
- \(\text{shr}_u32\) \(\$s1, \$s2, 2\);
- \(\text{shr}_u64\) \(\$d1, \$d2, 2\);
- \(\text{shr}_s32\) \(\$s1, \$s2, 2\);
- \(\text{shr}_s64\) \(\$d1, \$d2, 2\);
- \(\text{shl}_{u8x8}\) \(\$d0, \$d1, 2\);
- \(\text{shl}_{u8x4}\) \(\$s1, \$s2, 2\);
- \(\text{shr}_{u8x4}\) \(\$s1, \$s2, 1\);
- \(\text{shr}_{u8x8}\) \(\$d1, \$d2, 2\);

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

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>and_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>or_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>xor_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>not_TypeLength</td>
<td>dest, src0</td>
</tr>
<tr>
<td>popcount_u32_TypeLength</td>
<td>dest, src0</td>
</tr>
</tbody>
</table>

Explanation of Modifiers (see Table 4–2 (on page 108))

Type: b

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

Explanation of Operands (see 4.16 Operands (on page 113))

dest: Destination register.

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

Exceptions (see Chapter 12 Exceptions (on page 288))

No exceptions are allowed.

For BRIG syntax, see 18.7.1.5 BRIG syntax for individual bit instructions (on page 394).

5.6.2 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 \texttt{src0} and \texttt{src1} and places the result in the destination \texttt{dest}. The \texttt{and} instruction can be applied to 1-, 32-, and 64-bit values.

or
Performs the bitwise OR operation on the two sources \texttt{src0} and \texttt{src1} and places the result in the destination \texttt{dest}. The \texttt{or} instruction can be applied to 1-, 32-, and 64-bit values.

xor
Performs the bitwise XOR operation on the two sources \texttt{src0} and \texttt{src1} and places the result in the destination \texttt{dest}. The \texttt{xor} instruction can be applied to 1-, 32-, and 64-bit values.

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

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

See this pseudo code:

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

See Table 5–7 (below).

Table 5–7 Inputs and Results for popcount Instruction

<table>
<thead>
<tr>
<th>Input</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>0</td>
</tr>
<tr>
<td>00ffffff</td>
<td>24</td>
</tr>
<tr>
<td>7fffffff</td>
<td>31</td>
</tr>
<tr>
<td>01ffffff</td>
<td>25</td>
</tr>
<tr>
<td>fffffff</td>
<td>32</td>
</tr>
<tr>
<td>ffffffff</td>
<td>20</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitextract_TypeLength</td>
<td>dest, src0, src1, src2</td>
</tr>
<tr>
<td>bitinsert_TypeLength</td>
<td>dest, src0, src1, src2, src3</td>
</tr>
<tr>
<td>bitmask_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>bitrev_TypeLength</td>
<td>dest, src0</td>
</tr>
<tr>
<td>bitselect_TypeLength</td>
<td>dest, src0, src1, src2</td>
</tr>
<tr>
<td>firstbit_u32_TypeLength</td>
<td>dest, src0</td>
</tr>
<tr>
<td>lastbit_u32_TypeLength</td>
<td>dest, src0</td>
</tr>
</tbody>
</table>

Explanation of Modifiers (see Table 4–2 (on page 108))
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 (on page 113))
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 (on page 288))
No exceptions are allowed.

For BRIG syntax, see 18.7.1.6 BRIG syntax for bit string instructions (on page 394).

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;
5.7.2 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:

\[
\text{dest} = (\text{src1} \& \text{src0}) \mid (\text{src2} \& \neg \text{src0})
\]

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 pseudo code:

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

See Table 5–9 (on the facing 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 pseudo code:

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

See Table 5–9 (on the facing 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 pseudo code:

```c
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

<table>
<thead>
<tr>
<th>Input</th>
<th>Result for firstbit</th>
<th>Result for lastbit</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>00111111</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>71111111</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01111111</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>11111111</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11111100</td>
<td>0</td>
<td>8</td>
</tr>
</tbody>
</table>

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 (on page 328)).

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 (on page 40)).
5.8.1 Syntax

Table 5–10 Syntax for copy (move) instructions

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>combine_v2_b64_b32</td>
<td>dest, (src0, src1)</td>
</tr>
<tr>
<td>combine_v4_b128_b32</td>
<td>dest, (src0, src1, src2, src3)</td>
</tr>
<tr>
<td>combine_v2_b128_b64</td>
<td>dest, (src0, src1)</td>
</tr>
<tr>
<td>expand_v2_b32_b64</td>
<td>(dest0, dest1), src0</td>
</tr>
<tr>
<td>expand_v4_b32_b128</td>
<td>(dest0, dest1, dest2, dest3), src0</td>
</tr>
<tr>
<td>lda_segment_uLength</td>
<td>dest, address</td>
</tr>
<tr>
<td>mov_moveType</td>
<td>dest, src0</td>
</tr>
</tbody>
</table>

**Explanation of Modifiers**

- **segment**: Optional segment: global, group, private, kernarg or readonly. If omitted, flat is used. See 2.8 Segments (on page 32).

- **Length**: 1, 32, 64, 128 (see Table 4–2 (on page 108)). For lda must match the address size (see Table 2–3 (on page 41)).

- **moveType**: b1, b32, b64, b128, u32, u64, s32, s64, f16, f32, r0img, w0img, 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 (on page 40) and 16.2.1 Base profile requirements (on page 328).

**Explanation of Operands (see 4.16 Operands (on page 113))**

- **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 (on page 115).

**Exceptions (see Chapter 12 Exceptions (on page 288))**

No exceptions are allowed.

For BRIG syntax, see 18.7.1.7 BRIG syntax for copy (move) instructions (on page 395).

5.8.2 Description

**combine**

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

This 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 (on page 180). The size of `dest` must match the address size of the segment. See Table 2–3 (on page 41).

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 (on page 234) and §6.8 Notification (signal) instructions (on page 202)).

If `moveType` is `f16`, 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 (on page 118).

### 5.8.3 Additional information about `lda`

Assume the following:

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

If the address contains an identifier, then the segment for the identifier must agree with the segment used in the 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)
```
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 (2xf64) is not supported (see 16.2.1 Base profile requirements (on page 328)).

See also:

- 5.2 Integer arithmetic instructions (on page 127)
- 5.11 Floating-point arithmetic instructions (on page 151)
- 5.18 Compare (cmp) instruction (on page 166)

See Table 5–11 (below) and Table 5–12 (on the facing page).

5.9.1 Syntax

Table 5–11 Syntax for shuffle and interleave instructions

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>shuffle_TypeLength</code></td>
<td><code>dest, src0, src1, src2</code></td>
</tr>
<tr>
<td><code>unpacklo_TypeLength</code></td>
<td><code>dest, src0, src1</code></td>
</tr>
<tr>
<td><code>unpackhi_TypeLength</code></td>
<td><code>dest, src0, src1</code></td>
</tr>
</tbody>
</table>

Explanation of Modifiers (see 4.13.2 Packed data types (on page 109))

Type: s, u, f.
Length: 8x4, 8x8, 16x2, 16x4, 32x2

Explanation of Operands (see 4.16 Operands (on page 113))

*dest*: Destination. See the Description below.

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

shuffle

Selects half of the elements of src0 based on controls in src2 and copies them into the lower half of the dest. It then selects half of the elements of src1 based on controls in src2 and copies them into the upper half of the dest. src2 has the fixed compound type of b32. See 5.9.3 Controls in src2 for shuffle instruction (on page 147).

unpacklo

Copies and interleaves the lower half of the elements from each source into the destination. See 5.9.5 Examples of unpacklo and unpackhi instructions (on page 150).
**unpackhi**

Copies and interleaves the upper half of the elements from each source into the destination. See 5.9.5 Examples of unpacklo and unpackhi instructions (on page 150).

**pack**

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

src0 is the same packed type as dest.

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

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

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

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

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

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

src1 has the compound type srcTypesrcLength.

See 4.16 Operands (on page 113). 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 (on page 118).

**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 *srcTypesrcLength*.

See 4.16 Operands (on page 113). 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 (on page 118).

### 5.9.3 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**

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

<table>
<thead>
<tr>
<th>src2 Bits for Packed Data Types s8x8 and u8x8</th>
<th>Copied to</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-0 selects one of eight bytes from src0</td>
<td>dest bits 7-0</td>
</tr>
<tr>
<td>5-3 selects one of eight bytes from src0</td>
<td>dest bits 15-8</td>
</tr>
<tr>
<td>8-6 selects one of eight bytes from src0</td>
<td>dest bits 23-16</td>
</tr>
<tr>
<td>11-9 selects one of eight bytes from src0</td>
<td>dest bits 31-24</td>
</tr>
<tr>
<td>14-12 selects one of eight bytes from src1</td>
<td>dest bits 39-32</td>
</tr>
<tr>
<td>17-15 selects one of eight bytes from src1</td>
<td>dest bits 47-40</td>
</tr>
<tr>
<td>20-18 selects one of eight bytes from src1</td>
<td>dest bits 55-48</td>
</tr>
<tr>
<td>23-21 selects one of eight bytes from src1</td>
<td>dest bits 63-56</td>
</tr>
</tbody>
</table>
5.9.4 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.
Swap

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

\[
\text{shuffle}_u8x4 \text{ dest, src0, src0, 0x1b;}
\]

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.
5.9.5 Examples of unpacklo and unpackhi instructions

See the figure below.

Figure 5–3 Example of unpack

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmov_TypeLength</td>
<td>dest, src0, src1, src2</td>
</tr>
</tbody>
</table>

Explanation of Modifiers (see Table 4–2 (on page 108))

- **Type**: For the regular form: b. For the packed form: s, u, f.
- **Length**: For the regular form, Length can be 1, 32, 64, 128. For the packed form, Length can be 16x2, 16x4, 16x8, 32x2, 32x4; 64x2 unless Type is t and the base profile is specified (see 16.2.1 Base profile requirements (on page 328)); or 8x4, 8x8, 8x16, if Type is s or u. TypeLength applies to dest, src1, and src2. The type that applies to src0 for the regular form is b1, and for the packed form is Length.

Explanation of Operands (see 4.16 Operands (on page 113))

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

Exceptions (see Chapter 12 Exceptions (on page 288))

- No exceptions are allowed.

For BRIG syntax, see 18.7.1.9 BRIG syntax for bit conditional move (cmov) instruction (on page 395).

5.10.2 Description

The regular form of cmov conditionally moves either of two 1-bit, 32-bit, 64-bit, or 128-bit values into the destination register dest. If the source src0 is false (0), the destination is set to the value of src2; otherwise, the destination is set to the value of src1.

The packed form of cmov conditionally moves each element of the packed type independently. If the element in src0 is false (0), the corresponding destination element is set to the corresponding element of src2; otherwise, the destination is set to the corresponding element of src1.

Examples

- cmov_b32 $s1, $c3, $s1, $s2;
- cmov_b64 $d1, $c3, $d1, $d2;
- cmov_b32 $s1, $c0, $s1, $s2;
- cmov_u8x4 $s1, $s0, $s1, $s2;
- cmov_u8x4 $s1, $s0, $s1, $s2;
- cmov_u8x8 $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 (on page 116).
5.11.1 Syntax

Table 5-15 Syntax for floating-point arithmetic instructions

<table>
<thead>
<tr>
<th>Opcode and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_ftz_round_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>ceil_ftz_TypeLength</td>
<td>dest, src0</td>
</tr>
<tr>
<td>div_ftz_round_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>floor_ftz_TypeLength</td>
<td>dest, src0</td>
</tr>
<tr>
<td>fma_ftz_round_TypeLength</td>
<td>dest, src0, src1, src2</td>
</tr>
<tr>
<td>fract_ftz_round_TypeLength</td>
<td>dest, src0</td>
</tr>
<tr>
<td>max_ftz_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>min_ftz_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>mul_ftz_round_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>rint_ftz_TypeLength</td>
<td>dest, src0</td>
</tr>
<tr>
<td>sqrt_ftz_round_TypeLength</td>
<td>dest, src0</td>
</tr>
<tr>
<td>sub_ftz_round_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>trunc_ftz_TypeLength</td>
<td>dest, src0</td>
</tr>
</tbody>
</table>

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) (on page 119).

*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 (on page 323).

*Type*: f. See Table 4-2 (on page 108).

*Length*: 16, 32, and, if the base profile has not been specified, 64. See Table 4-2 (on page 108) and 16.2.1 Base profile requirements (on page 328).

Explanation of Operands (see 4.16 Operands (on page 113))

*dest*: Destination register.

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

Exceptions (see Chapter 12 Exceptions (on page 288))

Floating-point exceptions are allowed.
Table 5-16 Syntax for packed versions of floating-point arithmetic instructions

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_ftz_round_Control_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>ceil_ftz_Control_TypeLength</td>
<td>dest, src0</td>
</tr>
<tr>
<td>div_ftz_round_Control_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>floor_ftz_Control_TypeLength</td>
<td>dest, src0</td>
</tr>
<tr>
<td>fract_ftz_round_Control_TypeLength</td>
<td>dest, src0</td>
</tr>
<tr>
<td>max_ftz_Control_TypeLength</td>
<td>dest, src0</td>
</tr>
<tr>
<td>min_ftz_Control_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>mul_ftz_round_Control_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>rint_ftz_Control_TypeLength</td>
<td>dest, src0</td>
</tr>
<tr>
<td>sqrt_ftz_round_Control_TypeLength</td>
<td>dest, src0</td>
</tr>
<tr>
<td>sub_ftz_round_Control_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>trunc_ftz_Control_TypeLength</td>
<td>dest, src0</td>
</tr>
</tbody>
</table>

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) (on page 119).

round: 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 (on page 323).

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

See 4.14 Packing controls for packed data (on page 110).

TypeLength: f16x2, f16x4, f16x8, f32x2, f32x4, and, if the base profile has not been specified, f64x2. See 4.13.2 Packed data types (on page 109) and 16.2.1 Base profile requirements (on page 328).

Explanation of Operands (see 4.16 Operands (on page 113))

dest: Destination register.

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

Exceptions (see Chapter 12 Exceptions (on page 288))

Floating-point exceptions are allowed.

For BRIG syntax, see 18.7.1.10 BRIG syntax for floating-point arithmetic instructions (on page 396).

5.11.2 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) (on page 121)) of the mathematically accurate value in the base
profile. See Chapter 16 Profiles (on page 327).

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.

\[
\begin{align*}
\text{src0}' &= \text{ftz} ? \text{flush}_{\text{subnormal}}_{\text{to}}_{\text{zero}}(\text{src0}) : \text{src0} \\
\text{dest} &= \begin{cases} 
\text{src0}' & \text{if } \text{src0}' = +0.0 \\
\text{src0}' & \text{if } \text{src0}' = -0.0 \\
+\infty & \text{if } \text{src0}' = +\infty \\
-\infty & \text{if } \text{src0}' = -\infty \\
\text{NaN} & \text{if } \text{isNaN}(\text{src0}') \\
\end{cases} \end{align*}
\]

where:

\[
\begin{align*}
\text{smallest}_{\text{numeric}}_{f16} &= 0x1.f\text{f}-1h \\
\text{smallest}_{\text{numeric}}_{f32} &= 0x1.ffffffffep-1f \\
\text{smallest}_{\text{numeric}}_{f64} &= 0x1.fffffffffffffffep-1d
\end{align*}
\]

The \text{min} is used to ensure that the result of the \text{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) (on page 120).

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) (on page 120).

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) (on page 120).

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.

sqrt
Sets the destination `dest` to the square root of source `src0`.
If `src0` is negative, must return a quiet NaN and generate the invalid operation exception.
sqrt returns the correctly rounded result for the full profile and a result less than or equal to 1 ULP (see 4.19.6 Unit of least precision (ULP) (on page 121)) of the mathematically accurate value for the base profile. See Chapter 16 Profiles (on page 327).

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 $d3,1.0,5d0;`
- `min_f32 $s3,1.0f,$s1;`
- `min_f64 $d3,1.0,5d0;`
- `mul_f32 $s3,1.0f,$s1;`
- `mul_f64 $d3,1.0,5d0;`
- `sub_f32 $s3,1.0f,$s1;`
- `sub_f64 $d3,1.0,5d0;`
- `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

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mad_f32_round_TypeLength</code></td>
<td><code>dest, src0, src1, src2</code></td>
</tr>
</tbody>
</table>
5.12.2 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**:
  \[
  \text{fma	extunderscore ftz	extunderscore round	extunderscore fLength dest, src0, src1, src2};
  \]

- **Double Round Method**:
  \[
  \text{mul	extunderscore ftz	extunderscore round	extunderscore fLength temp, src0, src1};
  \text{add	extunderscore ftz	extunderscore round	extunderscore fLength dest, temp, src2};
  \]

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 (on page 116).

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 \texttt{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

Table 5–18 Syntax for floating-point bit instructions

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{abs_TypeLength}</td>
<td>\texttt{dest, src0}</td>
</tr>
<tr>
<td>\texttt{class_b1_TypeLength}</td>
<td>\texttt{dest, src0, cond}</td>
</tr>
<tr>
<td>\texttt{copysign_TypeLength}</td>
<td>\texttt{dest, src0, src1}</td>
</tr>
<tr>
<td>\texttt{neg_TypeLength}</td>
<td>\texttt{dest, src0}</td>
</tr>
</tbody>
</table>

Explanation of Modifiers (see 4.16 Operands (on page 113)) (see Table 4–2 (on page 108))

Type: f. See Table 4–2 (on page 108).

Length: 16, 32, and, if the base profile has not been specified, 64. See Table 4–2 (on page 108) and 16.2.1 Base profile requirements (on page 328).

Explanation of Operands (see 4.16 Operands (on page 113))

\texttt{dest}: Destination register.

\texttt{src0, src1}: Sources. Can be a register or immediate value.

\texttt{cond}: Source bit set specifying the conditions being tested. Must be a register or immediate value of compound type \texttt{u32}. See Table 5–19 (below).

Table 5–19 Class instruction source operand condition bits

<table>
<thead>
<tr>
<th>Condition being tested</th>
<th>Bit value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signaling NaN</td>
<td>0x001</td>
</tr>
<tr>
<td>Quiet NaN</td>
<td>0x002</td>
</tr>
<tr>
<td>Negative infinity</td>
<td>0x004</td>
</tr>
<tr>
<td>Negative normal</td>
<td>0x008</td>
</tr>
<tr>
<td>Negative subnormal</td>
<td>0x010</td>
</tr>
<tr>
<td>Negative zero</td>
<td>0x020</td>
</tr>
<tr>
<td>Positive zero</td>
<td>0x040</td>
</tr>
<tr>
<td>Positive subnormal</td>
<td>0x080</td>
</tr>
<tr>
<td>Positive normal</td>
<td>0x100</td>
</tr>
<tr>
<td>Positive infinity</td>
<td>0x200</td>
</tr>
</tbody>
</table>

Exceptions (see Chapter 12 Exceptions (on page 288))

No exceptions are allowed.
Table 5–20 Syntax for packed versions of floating-point bit instructions

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs_Control_TypeLength</td>
<td>dest, src0</td>
</tr>
<tr>
<td>copysign_Control_TypeLength</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>neg_Control_TypeLength</td>
<td>dest, src0</td>
</tr>
</tbody>
</table>

**Explanation of Modifiers**
- Control for `abs, and neg`: p or s.
- Control for `copysign`: pp, ps, sp, or ss.
- See 4.14 Packing controls for packed data (on page 110).

**TypeLength:** f16x2, f16x4, f16x8, f32x2, f32x4, and, if the base profile has not been specified, f64x2. See 4.13.2 Packed data types (on page 109) and 16.2.1 Base profile requirements (on page 328).

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

**Exceptions**
- No exceptions are allowed.

For BRIG syntax, see 18.7.1.5 BRIG syntax for individual bit instructions (on page 394).

### 5.13.2 Description

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

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

`cond` is interpreted using the values of Table 5–19 (on the 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:

```plaintext
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_b1_f32 $c1, $s1, 3;
class_b1_f32 $c1, $s1, $s2;
class_b1_f64 $c1, $d1, $s2;
class_b1_f64 $c1, $d1, $s2;
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 (on page 116).

5.14.1 Syntax

Table 5–21 Syntax for native floating-point instructions

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>ncos_f32</td>
<td>dest, src</td>
</tr>
<tr>
<td>nexp2_f32</td>
<td>dest, src</td>
</tr>
<tr>
<td>nfma_TypeLength</td>
<td>dest, src0, src1, src2</td>
</tr>
<tr>
<td>nlog2_f32</td>
<td>dest, src</td>
</tr>
<tr>
<td>nrcp_TypeLength</td>
<td>dest, src</td>
</tr>
<tr>
<td>nrsqrt_TypeLength</td>
<td>dest, src</td>
</tr>
<tr>
<td>nsin_f32</td>
<td>dest, src</td>
</tr>
<tr>
<td>nsqrt_TypeLength</td>
<td>dest, src</td>
</tr>
</tbody>
</table>

Explanation of Modifiers (see Table 4–2 (on page 108))

| Type: f. | Length: 16, 32, and, if the base profile has not been specified, 64. See 16.2.1 Base profile requirements (on page 328). |
Chapter 5. Arithmetic Instructions  5.14 Native floating-point instructions

<table>
<thead>
<tr>
<th>Explanation of Operands (see 4.16 Operands (on page 113))</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{dest}: Destination register.</td>
</tr>
<tr>
<td>\textit{src, src0, src1, src2}: Sources. Can be a register or immediate value.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exceptions (see 12.2 Hardware exceptions (on page 288))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard floating-point exceptions are allowed.</td>
</tr>
</tbody>
</table>

For BRIG syntax, see 18.7.1.13 BRIG syntax for native floating-point instructions (on page 397).

5.14.2 Description

\texttt{ncos}

Computes the cosine of the angle in source \textit{src} and stores the result in the destination \textit{dest}. The angle \textit{src} is in radians.

\texttt{nexp2}

Computes the base-2 exponential of a value.

\texttt{nfma}

The floating-point \texttt{nfma} (native fused multiply add) computes \textit{src0 * src1 + src2} and stores the result in the destination \textit{dest}.

\texttt{nlog2}

Finds the base-2 logarithm of a value.

\texttt{nrnp}

Computes the floating-point reciprocal.

\texttt{nrsqrt}

Computes the reciprocal of the square root.

\texttt{nsin}

Computes the sine of the angle in source \textit{src} and stores the result in the destination \textit{dest}. The angle \textit{src} is in radians.

\texttt{nsqrt}

Computes the square root.

Examples

\begin{verbatim}
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;
\end{verbatim}
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 5–22 Syntax for multimedia instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitalign_b32</td>
<td>dest, src0, src1, src2</td>
</tr>
<tr>
<td>bytealign_b32</td>
<td>dest, src0, src1, src2</td>
</tr>
<tr>
<td>lerp_u8x4</td>
<td>dest, src0, src1, src2</td>
</tr>
<tr>
<td>packcvt_u8x4_f32</td>
<td>dest, src0, src1, src2, src3</td>
</tr>
<tr>
<td>unpackcvt_f32_u8x4</td>
<td>dest, src0, src1</td>
</tr>
<tr>
<td>sad_u32_u32</td>
<td>dest, src0, src1, src2</td>
</tr>
<tr>
<td>sad_u32_u16x2</td>
<td>dest, src0, src1, src2</td>
</tr>
<tr>
<td>sad_u32_u8x4</td>
<td>dest, src0, src1, src2</td>
</tr>
<tr>
<td>sadhi_u16x2_u8x4</td>
<td>dest, src0, src1, src2</td>
</tr>
</tbody>
</table>

Explanation of Operands (see 4.16 Operands (on page 113))

- 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 (on page 288))

No exceptions are allowed.

For BRIG syntax, see 18.7.1.14 BRIG syntax for multimedia instructions (on page 398).

5.15.2 Description

bitalign

Used to align 32 bits within 64 bits of data on an arbitrary bit boundary. src2 is treated as a u32 value and the least significant 5 bits used to specify a shift amount. The 32-bit src0 and src1 are treated as the least significant and most significant bits of a 64-bit value respectively, which is shifted right by the shift amount of bits, and the least significant 32 bits returned.

```c
uint32 shift = src2 & 31;
uint64_t value = (((uint64_t)src1) << 32) | ((uint64_t)src0);
uint32_t dest = (uint32_t)((value >> shift) & 0xffffffff);
```

If src0 contains 0xA3A2A1A0 and src1 contains 0xB3B2B1B0, then:

- bitalign dest, src0, src1, 8 results in destination dest containing 0xB0A3A2A1.
- bitalign dest, src0, src1, 16 results in destination dest containing 0xB1B0A3A2.
- bitalign dest, src0, src1, 24 results in destination dest containing 0xB2B1B0A3.
bytealign

Used to align 32 bits within 64 bits of data on an arbitrary byte boundary. src2 is treated as a u32 value and the least significant 2 bits used to specify a shift amount. The 32-bit src0 and src1 are treated as the least significant and most significant bits of a 64-bit value respectively, which is shifted right by the shift amount of bytes, and the least significant 32 bits returned.

uint32 shift = (src2 & 3) * 8;
uint64_t value = (((uint64_t)src1) << 32) | ((uint64_t)src0);
uint32_t dest = (uint32_t)((value >> shift) & 0xffffffff);

If src0 contains 0xA3A2A1A0 and src1 contains 0xB2B1B0, 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

Opens four floating-point numbers, converts them to unsigned integer values, and packs them into a packed u8x4 value. Conversion is performed using round to nearest even. Values greater than 255.0 are converted to 255. Values less than 0.0 are converted to 0.

dest = (((uint32_t)(cvt_neari_sat_u8_f32(src0))) & 0) |
(((uint32_t)(cvt_neari_sat_u8_f32(src1))) & 0) |
(((uint32_t)(cvt_neari_sat_u8_f32(src2))) & 0) |
(((uint32_t)(cvt_neari_sat_u8_f32(src3))) & 0);

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:
  
  ```c
  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:
  
  ```c
  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

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:
  
  ```c
  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;
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 (on the facing page).

5.16.1 Syntax

Table 5–23 Syntax for segment checking (segmentp) instruction

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>segmentp_segment_nonull_b1_srcTypesrcLength</td>
<td>dest, src</td>
</tr>
</tbody>
</table>

Explanation of Modifiers (see Table 4–2 (on page 108))

segment: Can be global, group or private. See 2.8 Segments (on page 32).

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

srcType: u.

srcLength: 32, 64. The size of the source address. Must match the address size of flat addresses. See Table 2–3 (on page 41).

Explanation of Operands (see 4.16 Operands (on page 113))

dest: Destination register. Must be a control (c) register.

src: Source for the flat address that is being checked. Can be a register or immediate value. See Table 2–3 (on page 41).

Exceptions (see Chapter 12 Exceptions (on page 288))

No exceptions are allowed.

For BRIG syntax, see 18.7.1.15 BRIG syntax for segment checking (segmentp) instruction (on page 398).
5.16.2 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 (on page 40).

If it is known that the src operand can never have the flat address null pointer value, then the nonull modifier can be specified. On some implementations this might be more efficient. The result is undefined if the nonull 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 (on page 335).

See 2.8.4 Memory segment access rules (on page 37).

Examples

```
segmentp_private_b1_u32 $c1, $s0;     // small machine model
segmentp_global_b1_u32 $c1, $s0;       // small machine model
segmentp_global_nonull_b1_u32 $c1, $s0; // small machine model
segmentp_group_b1_u64 $c1, $d0;       // large machine model
```

5.17 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 (on the previous page).

5.17.1 Syntax

Table 5–24 Syntax for segment conversion instructions

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>ftos_segment_nonull_destTypedLength_srcTypesrcLength</td>
<td>dest, src</td>
</tr>
<tr>
<td>stof_segment_nonull_destTypedLength_srcTypesrcLength</td>
<td>dest, src</td>
</tr>
</tbody>
</table>

Explanation of Modifiers

segment: group or private. See 2.8 Segments (on page 32).

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

destType: u. See Table 4–2 (on page 108).

destLength: 32, 64. The size of the destination address. For ftos, must be the address size of segment; for stof, must be the flat address size. See Table 2–3 (on page 41).

srcType: u. See Table 4–2 (on page 108).

srcLength: 32, 64. The size of the source address. For ftos, must be the flat address size; for stof, must be the address size of segment. See Table 2–3 (on page 41).

Explanation of Operands (see 4.16 Operands (on page 113))

dest: Destination register.

src: Source to be converted. Can be a register or immediate value.

Exceptions (see Chapter 12 Exceptions (on page 288))

No exceptions are allowed.
For BRIG syntax, see 18.7.1.16 BRIG syntax for segment conversion instructions (on page 398).

5.17.2 Description

ftos

Converts the flat address specified by src into a segment address and stores the result in the destination register dest. If src is the flat address nullptr value, then dest is set to the segment address nullptr value. The destination register size must match the size of the segment address. If the source is a register, it must match the size of a flat address. See 2.9 Small and large machine models (on page 40).

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 (on page 36).

If the source is not in the specified segment, the result is undefined. See 2.8.4 Memory segment access rules (on page 37).

If it is known that the src operand can never have the flat address null pointer value, then the nonull modifier can be specified. On some implementations this might be more efficient. The result is undefined if the nonull 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 (on page 335).

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 (on page 40).

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 (on page 36).

If it is known that the src operand can never have the segment address null pointer value, then the nonull modifier can be specified. On some implementations this might be more efficient. The result is undefined if the nonull 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 (on page 335).

Examples

```c
// large machine model conversions
stof_private_u64_u32 $d1, $s1;
stof_private_nonull_u64_u32 $d1, $s1;
ftos_group_u32_u64 $s1, $d2;
ftos_group_nonull_u32_u64 $s1, $d2;
```

```c
// small machine model conversions
stof_private_u32_u32 $s1, $s2;
stof_private_nonull_u32_u32 $s1, $s2;
ftos_group_u32_u32 $s1, $s2;
ftos_group_nonull_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 (on page 118).

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 (on page 116).

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 (on page 288).

The ftz modifier is supported if the source operand type is floating-point. See 4.19.3 Flush to zero (ftz) (on page 119).

See Table 5-25 (below) and Table 5-26 (on the next page).

5.18.1 Syntax

Table 5-25 Syntax for compare (cmp) instruction

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp_op_ftz destType destLength srcTypesrcLength</td>
<td>dest, src0, src1</td>
</tr>
</tbody>
</table>

**Explanation of Modifiers (see Table 4-2 (on page 108))**

- op for bit types: eq and ne.
- op for integer source types: eq, ne, lt, le, gt, ge.
- op for floating-point source types: eq, ne, lt, le, gt, ge, equ, neu, ltu, leu, gtu, geu, num, nan and signaling NaN forms seq, sne, slt, sle, sgt, sge, sequ, sneu, sltu, sleu, sgtu, sgeu, snum, snan.
- ftz: Only valid for floating-point source types. Required if the base profile has been specified, otherwise optional. If specified, subnormal source values are flushed to zero. See 4.19.3 Flush to zero (ftz) (on page 119).

**destType destLength** Describes the destination.

- destType: u, s, f; b if destLength is 1.
- destLength: 32, 64; 1 if source type is b; 16 if source type is f. If the base profile has been specified, 64 is not supported if destType is f. See 16.2.1 Base profile requirements (on page 328).

**srcTypesrcLength** Describes the two sources.

- srcType: b, u, s, f.
- srcLength: 32, 64; 1 if source type is b; 16 if source type is f. If the base profile has been specified, 64 is not supported if srcType is f. See 16.2.1 Base profile requirements (on page 328).

**Explanation of Operands (see 4.16 Operands (on page 113))**

- dest: Destination register.
- src0, src1: Sources. Can be a register or immediate value.

**Exceptions (see Chapter 12 Exceptions (on page 288))**

Signaling NaN floating-point numbers generate the invalid operation exception. The a comparison forms also generate the invalid operation exception for quiet NaN floating-point numbers.
### 5.18 Compare (cmp) instruction

#### Table 5–26 Syntax for packed version of compare (cmp) instruction

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cmp_op ftz pp uLength TypeLength</code></td>
<td><code>dest, src0, src1</code></td>
</tr>
</tbody>
</table>

#### Explanation of Modifiers (see 4.13.2 Packed data types (on page 109))

- **op**: See Explanation of Modifiers table above.
- **ftz**: 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) (on page 119).
- **Type**: `a`, `u`, `f`.
- **Length**: `8x4`, `8x8`, `8x16`, `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 (on page 328).

#### Explanation of Operands (see 4.16 Operands (on page 113))

- **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 (on page 288))

- Signaling NaN floating-point numbers generate the invalid operation exception. A 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 (on page 399).

### 5.18.2 Description

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

<table>
<thead>
<tr>
<th>Type of dest</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>f16</code>, <code>f32</code>, <code>f64</code></td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td><code>u8</code></td>
<td><code>0xff</code></td>
<td><code>0x00</code></td>
</tr>
<tr>
<td><code>u16</code></td>
<td><code>0xffff</code></td>
<td><code>0x0000</code></td>
</tr>
<tr>
<td><code>u32</code>, <code>s32</code></td>
<td><code>0xffffffff</code></td>
<td><code>0x00000000</code></td>
</tr>
<tr>
<td><code>u64</code>, <code>s64</code></td>
<td><code>0xffffffffffffffff</code></td>
<td><code>0x0000000000000000</code></td>
</tr>
<tr>
<td><code>b1</code></td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**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.
Ordered comparisons. These correspond to \textit{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, \texttt{eq} and \texttt{ne} support the \texttt{bl} 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.

Unordered comparisons. There are unordered forms of all the ordered comparisons. For example, \texttt{leu} is the unordered form of \texttt{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.

Signalizing comparisons. There are signalizing forms of the ordered, unordered, \texttt{num} and \texttt{nan} comparisons. For example, \texttt{sle} is the signaling form of \texttt{le}. Only supported for floating point source operand types. Returns the same result as the corresponding non-signalizing 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 5–27 (below):

- The table gives a mapping from the HSAIL floating-point comparisons to the corresponding IEEE/ANSI Standard 754-2008 four mutually exclusive relations \textit{less than} (LT), \textit{equal} (EQ), \textit{greater than} (GT) and \textit{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 \textit{equal} to -0.0.
- Infinite operands of the same sign compare as \textit{equal}.
- Every NaN compares \textit{unordered} with everything, including itself.
- The table also gives the IEEE/ANSI Standard 754-2008 equivalent operation name if available.

Table 5–27 Floating-point comparisons

<table>
<thead>
<tr>
<th>HSAIL</th>
<th>IEEE/ANSI Standard 754-2008</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{num}</td>
<td>EQ, LT, GT</td>
</tr>
<tr>
<td>\texttt{nan}</td>
<td>UN</td>
</tr>
<tr>
<td>\texttt{eq}</td>
<td>EQ</td>
</tr>
<tr>
<td>\texttt{ne}</td>
<td>LT, GT</td>
</tr>
<tr>
<td>\texttt{lt}</td>
<td>LT</td>
</tr>
<tr>
<td>\texttt{le}</td>
<td>EQ, LT</td>
</tr>
<tr>
<td>\texttt{gt}</td>
<td>GT</td>
</tr>
<tr>
<td>\texttt{ge}</td>
<td>EQ, GT</td>
</tr>
<tr>
<td>\texttt{equ}</td>
<td>EQ, UN</td>
</tr>
<tr>
<td>\texttt{neu}</td>
<td>LT, GT, UN</td>
</tr>
<tr>
<td>\texttt{ltu}</td>
<td>LT, UN</td>
</tr>
<tr>
<td>\texttt{leu}</td>
<td>EQ, LT, UN</td>
</tr>
<tr>
<td>\texttt{gtu}</td>
<td>GT, UN</td>
</tr>
</tbody>
</table>
### 5.18 Compare (cmp) instruction

**HSAIL**

<table>
<thead>
<tr>
<th>Comparison Operation</th>
<th>True Relations</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>geu</td>
<td>EQ, GT, UN</td>
<td>compareQuietNotLess</td>
</tr>
<tr>
<td>snum</td>
<td>EQ, LT, GT</td>
<td></td>
</tr>
<tr>
<td>anan</td>
<td>UN</td>
<td></td>
</tr>
<tr>
<td>seq</td>
<td>EQ</td>
<td>compareSignalingEqual</td>
</tr>
<tr>
<td>sne</td>
<td>LT, GT</td>
<td>compareSignalingLess</td>
</tr>
<tr>
<td>sle</td>
<td>EQ, LT</td>
<td>compareSignalingLessEqual</td>
</tr>
<tr>
<td>sgt</td>
<td>GT</td>
<td>compareSignalingGreater</td>
</tr>
<tr>
<td>sge</td>
<td>EQ, GT</td>
<td>compareSignalingGreaterEqual</td>
</tr>
<tr>
<td>sequ</td>
<td>EQ, UN</td>
<td></td>
</tr>
<tr>
<td>sneu</td>
<td>LT, GT, UN</td>
<td>compareSignalingNotEqual</td>
</tr>
<tr>
<td>sltu</td>
<td>LT, UN</td>
<td>compareSignalingLessUnordered</td>
</tr>
<tr>
<td>sleu</td>
<td>EQ, LT, UN</td>
<td>compareSignalingNotGreater</td>
</tr>
<tr>
<td>sgtu</td>
<td>GT, UN</td>
<td>compareSignalingGreaterUnordered</td>
</tr>
<tr>
<td>sgeu</td>
<td>EQ, GT, UN</td>
<td>compareSignalingNotLess</td>
</tr>
</tbody>
</table>

**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 (on page 116).

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.

For packed formats the source and destination types must have the same number of elements, and the conversion is applied pairwise to corresponding elements independently.

Table 5–28 (below) shows how the first step of the conversion instruction does the transformation. The table uses the notation defined in Table 5–29 (on page 173).

<table>
<thead>
<tr>
<th>Destination</th>
<th>Source b1</th>
<th>Source u8</th>
<th>Source s8</th>
<th>Source u16</th>
<th>Source s16</th>
<th>Source f16</th>
<th>Source u32</th>
<th>Source s32</th>
<th>Source f32</th>
<th>Source u64</th>
<th>Source s64</th>
<th>Source f64</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1</td>
<td></td>
<td></td>
<td></td>
<td>ztest</td>
<td>ztest</td>
<td>ztest</td>
<td>ztest</td>
<td>ztest</td>
<td>ztest</td>
<td>ztest</td>
<td>ztest</td>
<td>ztest</td>
</tr>
<tr>
<td>u8</td>
<td>zext</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ztest</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s8</td>
<td>zext</td>
<td>isat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ztest</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u16</td>
<td>zext</td>
<td>zext</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s16</td>
<td>zext</td>
<td>zext</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f16</td>
<td>u2h</td>
<td>u2h</td>
<td>s2h</td>
<td>u2h</td>
<td>s2h</td>
<td>-</td>
<td>u2h</td>
<td>s2h</td>
<td>f2h</td>
<td>u2h</td>
<td>s2h</td>
<td>d2h</td>
</tr>
<tr>
<td>u32</td>
<td>zext</td>
<td>zext</td>
<td>zext</td>
<td>zext</td>
<td>zext</td>
<td>h2u</td>
<td>-</td>
<td>isat</td>
<td>f2u</td>
<td>-</td>
<td></td>
<td>d2u</td>
</tr>
<tr>
<td>s32</td>
<td>b2s</td>
<td>zext</td>
<td>zext</td>
<td>zext</td>
<td>zext</td>
<td>h2s</td>
<td>isat</td>
<td>-</td>
<td>f2s</td>
<td></td>
<td></td>
<td>d2s</td>
</tr>
<tr>
<td>f32</td>
<td>u2f</td>
<td>u2f</td>
<td>s2f</td>
<td>u2f</td>
<td>s2f</td>
<td>h2f</td>
<td>u2f</td>
<td>s2f</td>
<td>-</td>
<td>u2f</td>
<td>s2f</td>
<td>d2f</td>
</tr>
<tr>
<td>Source</td>
<td>u1</td>
<td>u8</td>
<td>s8</td>
<td>u16</td>
<td>s16</td>
<td>f16</td>
<td>u32</td>
<td>s32</td>
<td>f32</td>
<td>u64</td>
<td>s64</td>
<td>f64</td>
</tr>
<tr>
<td>--------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>Dest.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u64</td>
<td>zext</td>
<td>zext</td>
<td>sext</td>
<td>sext</td>
<td>sext</td>
<td>h2u</td>
<td>zext</td>
<td>sext</td>
<td>f2u</td>
<td>-</td>
<td>isat</td>
<td>d2u</td>
</tr>
<tr>
<td>s64</td>
<td>b2s</td>
<td>zext</td>
<td>sext</td>
<td>sext</td>
<td>sext</td>
<td>h2s</td>
<td>zext</td>
<td>sext</td>
<td>f2s</td>
<td>isat</td>
<td>-</td>
<td>d2s</td>
</tr>
<tr>
<td>f64</td>
<td>u2d</td>
<td>u2d</td>
<td>s2d</td>
<td>u2d</td>
<td>s2d</td>
<td>h2d</td>
<td>u2d</td>
<td>s2d</td>
<td>f2d</td>
<td>u2d</td>
<td>s2d</td>
<td>-</td>
</tr>
</tbody>
</table>

### 5.19.2 Syntax

Table 5–30 Syntax for conversion (cvt) instruction

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>cvt_ftz_round_sat_destTypedestLength_srcTypesrcLength</td>
<td>dest, src</td>
</tr>
</tbody>
</table>

**Explanation of Modifiers (see Table 4–2 (on page 108))**

- **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) (on page 119).
Chapter 5. Arithmetic Instructions  5.19 Conversion (cvt) instruction

### Explanation of Modifiers (see Table 4-2 on page 108)

**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, sdowi, szeroi, sneari, supi_sat, sdowi_sat, zzeroi_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 (on page 118), 16.2.1 Base profile requirements (on page 328), 5.19.3 Rules for rounding for conversions (on the facing page), 5.19.4 Description of integer rounding modes (on the facing page), and 5.19.5 Description of floating-point rounding modes (on page 177).

**sat**: Optional. Only valid if `destType` and `srcType` are integer, `destLength` is less than or equal to `srcLength`, and `destTypedestLength` is not the same as `srcTypesrcLength`. See 5.19.6 Description of integer saturating mode (on page 178).

**destType**: b, u, s, f.

**destLength**: 1, 8, 16, 32, 64. 1 is only allowed for `destType` of b. 1 and 8 are not allowed for `destType` of f. If the base profile has been specified, 64 is not supported if `destType` is f. See 16.2.1 Base profile requirements (on page 328).

**srcType**: b, u, s, f.

**srcLength**: 1, 8, 16, 32, 64. 1 is only allowed for `srcType` of b. 1 and 8 are not allowed for `srcType` of f. If the base profile has been specified, 64 is not supported if `srcType` is f. See 16.2.1 Base profile requirements (on page 328).

### Explanation of Operands (see 4.16 Operands on page 113)

**dest**: Destination register.

**src**: Source. Can be a register or immediate value.

### Exceptions (see Chapter 12 Exceptions on page 288)

Floating-point exceptions are allowed.

### Table 5–31 Syntax for packed versions of conversion (cvt) instructions

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cvt_ftz_round_Control</code></td>
<td><code>destTypedestLength srcTypesrcLength</code></td>
</tr>
<tr>
<td><code>dest, src</code></td>
<td></td>
</tr>
</tbody>
</table>

### Explanation of Modifiers (see 4.14 Packing controls for packed data on page 110)

**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) (on page 119).

**round**: Optional rounding mode. Only valid if `destType` and/or `srcType` is floating-point, unless both are floating-point types and `destType` element size is larger than `srcType` element size. Possible values are up, down, zero, near, upi, downi, zeroi, neari, upi_sat, downi_sat, zeroi_sat, neari_sat, supi, sdowi, szeroi, sneari, supi_sat, sdowi_sat, zzeroi_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 (on page 118), 16.2.1 Base profile requirements (on page 328), 5.19.3 Rules for rounding for conversions (on the facing page), 5.19.4 Description of integer rounding modes (on the facing page), and 5.19.5 Description of floating-point rounding modes (on page 177).

**Control**: pp, or pp_sat if `destType` and `srcType` are both integer types.

**destType, srcType**: s, u, f.

**destLength, srcLength**: If `integerType` 8x4, 8x8, 8x16, 16x2, 16x4, 16x8, 32x2, 32x4, or 64x2; if floating-point type f16x2, f16x4, f16x8, f32x2, f32x4, and, if the base profile has not been specified, f64x2. `destLength` and `srcLength` must have the same number of packed elements. See 4.13.2 Packed data types (on page 109) and 16.2.1 Base profile requirements (on page 328).
### 5.19.3 Rules for rounding for conversions

Rounding for conversions follows the rules shown in Table 5–32 (below).

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

Table 5–32 Rules for rounding for conversions

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Type of rounding</th>
<th>Default rounding</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>f (smaller size)</td>
<td>floating-point</td>
<td>default rounding mode (specified by the module header)</td>
</tr>
<tr>
<td>f</td>
<td>f (larger size)</td>
<td>none (must not specify rounding)</td>
<td>none (no rounding performed)</td>
</tr>
<tr>
<td>s or u</td>
<td>f</td>
<td>floating-point</td>
<td>default rounding mode (specified by the module header)</td>
</tr>
<tr>
<td>f</td>
<td>s or u</td>
<td>integer</td>
<td>zeroi</td>
</tr>
<tr>
<td>f</td>
<td>bl</td>
<td>none (must not specify rounding)</td>
<td>none (always converts using ztest)</td>
</tr>
<tr>
<td>bl</td>
<td>f</td>
<td>none (must not specify rounding)</td>
<td>none (always converts to 0.0 or 1.0)</td>
</tr>
<tr>
<td>bl, s, or u</td>
<td>bl, s, or u</td>
<td>none (must not specify rounding)</td>
<td>none (no rounding performed)</td>
</tr>
</tbody>
</table>

### 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 5–33 (on page 177).

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) (on page 120).

The ftz modifier is supported. See 4.19.3 Flush to zero (ftz) (on page 119).

- 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 generated.
  - If the destination integer type is unsigned and the rounded result is $-\infty$ 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 $+\infty$ or greater than $2^{\text{destLength}-1}$:
    - The result is $2^{\text{destLength}-1}$. 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 $-\infty$ or less than $-2^{\text{destLength}-1}$:
    - The result is $-2^{\text{destLength}-1}$. 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 $+\infty$ 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.
generated. Otherwise, no exceptions must generated.

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

Table 5–33 Integer rounding modes

<table>
<thead>
<tr>
<th>Regular Integer Rounding Modes</th>
<th>Saturating Integer Rounding Modes</th>
<th>Regular Integer Rounding Mode Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Signaling Form</td>
<td>Signaling Form</td>
<td></td>
</tr>
<tr>
<td>upi</td>
<td>supi</td>
<td>Rounds up to the nearest integer greater than or equal to the exact result.</td>
</tr>
<tr>
<td>downi</td>
<td>sdowni</td>
<td>Rounds down to the nearest integer less than or equal to the exact result.</td>
</tr>
<tr>
<td>zeroi</td>
<td>szeroi</td>
<td>Rounds to the nearest integer toward zero.</td>
</tr>
<tr>
<td>neari</td>
<td>sneari</td>
<td>Rounds to the nearest integer. If there is a tie, chooses an even integer.</td>
</tr>
</tbody>
</table>

Examples are:

If $s1$ has the value 1.6, then:

```
cvt_upi_s32_f32 $s2, $s1; // sets $s2 = 2
```  

If $s1$ has the value -1.6, then:

```
cvt_upi_s32_f32 $s2, $s1; // sets $s2 = -1
```  

5.19.5 Description of floating-point rounding modes

The floating-point rounding modes are (see 4.19.2 Floating-point rounding (on page 118)):

- **up** — Rounds up toward the nearest representable value that is greater than the infinitely accurate result. Note that this cannot result in minus infinity.

- **down** — Rounds down toward the nearest representable value that is less than the infinitely accurate result. Note that this cannot result in plus infinity.

- **zero** — Rounds toward the nearest representable value that is no greater in magnitude than the infinitely accurate result. Note that this cannot result an infinity.

- **near** — If the magnitude of the infinitely accurate result is at least $\text{destTypeddestLength}_{\text{max}} + \text{sulp}(\text{destTypeddestLength}_{\text{max, float}})$ then rounds to the infinity with the same sign (see 4.19.6 Unit of least precision (ULP) (on page 121)). Otherwise, rounds toward the nearest finite 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 (on page 323)). If the base profile has been specified, then it must be omitted. See 4.19.2 Floating-point rounding (on page 118).

The \texttt{ftz} modifier is supported. See 4.19.3 Flush to zero (ftz) (on page 119).

If the source operand is a NaN, then the result must be a quiet NaN. 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) (on page 120) for information on how the NaN payload must be handled.

Otherwise, the infinitely accurate 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 (on page 118).

- 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 (on page 323)) if the base profile has been specified, then it must be omitted. See 4.19.2 Floating-point rounding (on page 118).

The \texttt{ftz} modifier is not supported. See 4.19.3 Flush to zero (ftz) (on page 119).

Otherwise, the infinitely accurate 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 (on page 118).

Floating-point rounding modes are invalid in all other cases.

### 5.19.6 Description of integer saturating mode

Integer saturating mode is allowed if all the following apply:

- Destination type and source type are integer.
- Destination type size is less than or equal to the source type size.
- Destination type is not the same as source type.

If the source value is greater than or less than the range of the destination type it is set to the maximum or minimum representable value of the destination type respectively.

The result is sign-extended if destination type is $s$, and zero-extended if destination type is $u$, to the size of the destination register. See 4.16.2 Rules for operand registers (on page 114).

Integer saturating mode is invalid in all other cases.

The saturating form might execute slower than the non-saturating form.
**Examples of regular (nonpacked) instructions**

```c
  cvt_s32_u16 $s1, $s1;
  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;
  cvt_sat_u8_s32 $s1, $s2;
  cvt_sat_s16_u16 $s1, $s2;
  cvt_sat_s32_u64 $s1, $d2;
  cvt_sat_u8_s8 $s1, $s2;
```

**Examples of packed instructions**

```c
  cvt_pp_s32x2_u16x2 $d1, $s1;
  cvt_downi_pp_u32x2_f32x2 $d1, $d2;
  cvt_pp_f16x2_u32x2 $s1, $d3;
  cvt_ftz_up_pp_f16x2_f32x2 $s1, $d0;
  cvt_pp_sat_u16x2_s32x2 $s1, $d3;
  cvt_pp_sat_u16x2_s16x2 $s1, $s0;
```
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 (on page 40). 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 (on page 36).

6.1.1 How addresses are formed

The format of an address expression is described in 4.18 Address expressions (on page 115).

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 (on page 41).

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 (on page 86).

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 (on page 36).

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 (on page 115).

### 6.1.2 Memory hierarchy

Figure 6-1 (on the 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.
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 1.2*, 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 (on page 209)* which use a separate image memory model defined in 7.1.10 *Image memory model (on page 236)*.

#### 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 Version 1.2*, 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 1.2, 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 (see 7.6 image fence (imagefence) instruction (on page 244)).
- `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 (on page 328)).

See 6.2.6 Coarse grain allocation (on page 186) 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.
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 (on page 187).

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 (on page 32).

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 (on the 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 (on page 103).

See 2.8 Segments (on page 32).

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. Program execution is undefined if a location is accessed that is part of an agent allocation, unless the access is performed by:

- the kernel agent that the allocation is associated
- a host agent using the HSA runtime copy operation

The global segment allows variables to have agent allocation. See 4.3.10 Declaration and definition qualifiers (on page 73). 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 program execution is 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 program execution is 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 (on the previous page)).
6.2.6 Coarse grain allocation

NOTE: Current coarse grain memory ownership definition is deprecated and likely to change in an incompatible way in the next release.

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 coarse grain allocation does not support system scope (see 6.2.2 Memory scope (on page 184)). 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 segment, and can specify either agent or system scope. They also make accesses to image data by image instructions coherent with accesses by memory instructions using the global segment (see 7.1.10 Image memory model (on page 236)).

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 1.2, 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 3.3.8 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 (on page 257)](chapter6-memoryinstructions-section-9-3).

### 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 (on page 115)](chapter-4-address-expressions)) and places the result into one or more registers. It is an ordinary non-synchronizing memory instruction (see [6.2 Memory model (on page 183)](chapter6-memoryinstructions-section-6-2)).

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 (on page 113)](chapter-4-operands). Integer values are sign-extended or zero-extended to fit the destination register size. f16 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 (on page 118)](chapter-4-floating-point-numbers)). 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 (f64) is not supported (see [16.2.1 Base profile requirements (on page 328)](chapter16-base-profile-requirements)).
6.3.1 Syntax

Table 6-1 Syntax for load (ld) instruction

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ld_segment_align(n)</code></td>
<td><code>dest0, address</code></td>
</tr>
<tr>
<td><code>ld_v2_segment_align(n)</code></td>
<td><code>(dest0,dest1), address</code></td>
</tr>
<tr>
<td><code>ld_v3_segment_align(n)</code></td>
<td><code>(dest0,dest1,dest2), address</code></td>
</tr>
<tr>
<td><code>ld_v4_segment_align(n)</code></td>
<td><code>(dest0,dest1,dest2,dest3), address</code></td>
</tr>
</tbody>
</table>

Explanation of Modifiers

- `v2, v3, and v4`: Optional vector element count. Used to specify that multiple contiguous memory locations, each of type `TypeLength`, are being loaded. See the Description below.
- `segment`: Optional segment: global, group, private, kernarg, readonly, spill, or arg. If omitted, a flat address is used. See 2.8 Segments (on page 32).
- `align(n)`: Optional. Used to specify the byte alignment of the base of the memory being loaded. If omitted, 1 is used indicating no alignment. See the Description below.
- `const`: Optional. Used to indicate if the memory loaded is constant. Only allowed if `segment` specifies the global segment or flat address. See the Description below.
- `equiv(n)`: Optional: `n` is an equivalence class. Used to specify the equivalence class of the memory locations being accessed. If omitted, class 0 is used, which indicates that any memory location may be aliased. See 6.1.4 Equivalent classes (on page 182).
- `width`: Optional: `width(n)`, `width(WEVSIZE)`, or `width(all)`. Used to specify the result uniformity of the loaded values. All active work-items in the same slice are guaranteed to load the same value(s). If the width modifier is omitted, it defaults to `width(1)`, indicating each active work-item can load different value(s). See the Description below.
- `nt`: Optional. Used to indicate if the memory loaded is non-temporal. See the 6.3.2 Description (on the facing page) below.
- `Type`: `u, s, f`. The `Type` specifies how the value is expanded to the size of the destination. See Table 4-2 (on page 108).
- `Length`: 8, 16, 32, 64. If the base profile has been specified, 64 is not supported if `Type` is `f`. The `Length` specifies the amount of data fetched from memory, and the amount to increment the address when the destination is a vector operand. See Table 4-2 (on page 108) and 16.2.1 Base profile requirements (on page 328).
- `TypeLength` can also be `b128`, in which case `destn` must be a `q` register; or `roimg, woimg, rzimg, samp, sig32, or sig64`, in which case `destn` must be a `d` register.

Explanation of Operands (see 4.16 Operands (on page 113))

- `dest0, dest1, dest2, dest3`: Destination registers.
- `address`: Address to be loaded from. Must be an address expression for an address in `segment` (see 4.18 Address expressions (on page 115)).

Exceptions (see Chapter 12 Exceptions (on page 288))

Invalid address exceptions are allowed. May generate a memory exception if address is unaligned and the aligned modifier has been specified.

For BRIG syntax, see 18.7.2 BRIG syntax for memory instructions (on page 399).
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 (on page 334).

const
If specified, indicates that the value of the location accessed by the load will be the same value as when the kernel dispatch started execution. An implementation can rely on the memory locations loaded not being written to since the start of the associated kernel dispatch. Note that the locations may be written after a load that is marked const, in which case any subsequent load instructions for those locations must not be marked const. Only global and readonly segment loads, and flat addresses that refer to constant global segment memory, can be marked const.

On some implementations, knowing a load is accessing memory that has not changed since the start of the kernel dispatch might be more efficient. The behavior is undefined if a memory load marked const is changed between the start of the kernel dispatch and the load instruction: on some implementations this might result in incorrect values being loaded. See 17.9 Constant access (on page 334).

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 (on page 42)). 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.

If specified, indicates that the location accessed by the load is non-temporal and will not likely be accessed again in the near term.

On some implementations, knowing a load is non-temporal might be more efficient. For example, the value loaded into a cache can be marked for early eviction. This avoids non-temporal loads from polluting the cache and causing temporal loads and stores, that will be accessed again in the near term, to be evicted prematurely. This is only a hint and can be ignored by an implementation. However, on some implementations incorrectly marking loads could result in reduced performance.

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 (on page 180).

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

```
ld_s32 $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 (on page 118).

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 (on page 234) and 6.8 Notification (signal) instructions (on page 202)).
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 (on page 183).

### Examples

```assembly
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_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_n32 ($s1,$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, [signal164];
```

### 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 (on page 113)) into memory using a segment or flat address expression (see 4.18 Address expressions (on page 115)). It is an ordinary non-synchronizing memory instruction (see 6.2 Memory model (on page 183)).

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 (on page 328)).
6.4.1 Syntax

Table 6-2 Syntax for store (st) instruction

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>st_segment_align($n) _equiv($n) _nt_TypeLength</td>
<td>src0, address</td>
</tr>
<tr>
<td>st_v2_segment_align($n) _equiv($n) _nt_TypeLength</td>
<td>(src0, src1), address</td>
</tr>
<tr>
<td>st_v3_segment_align($n) _equiv($n) _nt_TypeLength</td>
<td>(src0, src1, src2), address</td>
</tr>
<tr>
<td>st_v4_segment_align($n) _equiv($n) _nt_TypeLength</td>
<td>(src0, src1, src2, src3), address</td>
</tr>
</tbody>
</table>

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 (on page 32).

align($n$): 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.

equiv($n$): Optional: $n$ is an equivalence class. Used to specify the equivalence class of the memory locations being accessed. If omitted, class 0 is used, which indicates that any memory location may be aliased. See 6.1.4 Equivalence classes (on page 182).

nt: Optional. Used to indicate if the memory loaded is non-temporal. See the 6.4.2 Description (below) below.

Type: u, s, f. The Type specifies how the value is extracted from the source to match the size of the destination. See Table 4-2 (on page 108).

Length: 8, 16, 32, 64. If the base profile has been specified, 64 is not supported if Type is f. The Length specifies the amount of data stored, and the amount to increment the address when the destination is a vector operand. See Table 4-2 (on page 108) and 16.2.1 Base profile requirements (on page 328).

TypeLength can also be b128, in which case srcn must be a q register, or roimg, woimg, rwimg, samp, sig32, or sig64, in which case srcn must be a d register. If roimg, woimg, rwimg or samp then segment must be arg.

Explanation of Operands (see 4.16 Operands (on page 113))

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

address: Address to be stored into. Must be an address expression for an address in segment (see 4.18 Address expressions (on page 115)).

Exceptions (see Chapter 12 Exceptions (on page 288))

Invalid address expressions are allowed. May generate a memory exception if address is unaligned and the aligned modifier has been specified.

For BRIG syntax, see 18.7.2 BRIG syntax for memory instructions (on page 399).

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:

\[
\text{st}_v2\_s32\ (\$s0,\ \$s1),\ \{\$s2\};
\]

For example, this code:

\[
\text{st}_v4\_u8\ (\$s1,\ \$s2,\ \$s3,\ \$s4),\ \{120\};
\]

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.

\textbf{align (n)}

If specified, indicates that the implementation can rely on the \textit{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 program execution is 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 \textbf{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 \textit{TypeLength}. See 17.8 Unaligned access (on page 334).

\textbf{nt}

If specified, indicates that the location accessed by the store is non-temporal and will not likely be accessed again in the near term.

On some implementations, knowing a store is non-temporal might be more efficient. For example, the value in the cache that was stored can be marked for early eviction. This avoids non-temporal stores from polluting the cache and causing temporal loads and stores, that will be accessed again in the near term, to be evicted prematurely. This is only a hint and can be ignored by an implementation. However, on some implementations incorrectly marking stores could result in reduced performance.

### 6.4.3 Additional information

If \textbf{segment} is present, the address expression must be a \textit{segment} address of the same kind. If \textbf{segment} is omitted, the address expression must be a flat address. See 6.1.1 How addresses are formed (on page 180).

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

\[
\text{st}_b64\ \$s1,\ \{\&g\};\ //\ \text{not\ valid\ because\ address\ expression\ is\ a\ segment}\n\]
\[
//\ \text{address,\ but\ a\ flat\ address\ is\ required.}
\]

If \text{st}_v2, \text{st}_v3, \text{or} \text{st}_v4 is used, then all the registers must be the same size.

Subword integer type values (\text{\$s8, \$u8, \$s16 and \$u16}) are extracted from the least significant bits of the source \text{s} register. For example, storing a 256 with a \text{st}_\text{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 (on page 118).

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 (on page 234) and 6.8 Notification (signal) instructions (on page 202)).

The `roimg`, `woimg`, `rwimg` and `samp` value types are only allowed if `segment` is `arg` (see 7.1.7 Image creation and image handles (on page 227) and 7.1.8 Sampler creation and sampler handles (on page 232)).

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 (on page 183).

### Examples

```assembly
set_global_f32 $s1, [$r4];
set_global_align(4)_f32 $s1, [$r4];
set_global_u8 $s1, [$r4];
set_global_u16 $s1, [$r4];
set_global_u32 $s1, [$r4];
set_global_u32 WAVESIZE, [$r4];
set_global_f16 $s1, [$r4];
set_global_f64 $d1, [$r4];
set_global_align(8)_f64 $d1, [$r4];
set_private_f32 $s1, [$r3+4];
set_global_f32 $s1, [$r3+4];
set_spill_f32 $s1, [$r3+4];
set_arg_f32 $s1, [$r3+4];
set_f32 $s1, [$r3+4];
set_align(4)_f32 $s1, [$r3+4];
set_v4_f32 ($s1,$s1,$s1,$s1), [$r3+4];
set_v2_align(8)_equiv(9)_f32 ($s1,$s2), [$r3+4];
set_v3_s32 ($s1,$s1,$s1), [$r3+4];
set_group_equiv(0)_u32 $s0, [$r2];
set_equiv(1)_u64 $d3, [$r4+32];
set_align(16)_equiv(1)_nt_u64 $d3, [$r4+32];
set_v2_equiv(1)_u64 ($d1,$d2), [$r0+32];
set_equiv(1)_u64 $d6, [128];
set_arg_roimg $d2, [@roimage2];
set_arg_rwimg $d1, [@rwimage2];
set_arg_woimg $d2, [@woimage2];
set_arg_samp $d3, [@sampler2];
set_global_sig32 $d3, [signal32];
set_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 atomicnoret instructions can specify a memory order and memory scope.

For more information, see:

- 6.2 Memory model (on page 183)
- 6.6 Atomic (atomic) instructions (below)
- 6.7 Atomic no return (atomicnoret) instructions (on page 200)

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 (on page 183)).
6.6.1 Syntax

Table 6–3 Syntax for atomic instructions

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic ld segment order scope equiv(n) _TypeLength</td>
<td>dest, address</td>
</tr>
<tr>
<td>atomic and segment order scope equiv(n) _TypeLength</td>
<td>dest, address, src0</td>
</tr>
<tr>
<td>atomic or segment order scope equiv(n) _TypeLength</td>
<td>dest, address, src0</td>
</tr>
<tr>
<td>atomic xor segment order scope equiv(n) _TypeLength</td>
<td>dest, address, src0</td>
</tr>
<tr>
<td>atomic exch segment order scope equiv(n) _TypeLength</td>
<td>dest, address, src0</td>
</tr>
<tr>
<td>atomic add segment order scope equiv(n) _TypeLength</td>
<td>dest, address, src0</td>
</tr>
<tr>
<td>atomic sub segment order scope equiv(n) _TypeLength</td>
<td>dest, address, src0</td>
</tr>
<tr>
<td>atomic wrapinc segment order scope equiv(n) _TypeLength</td>
<td>dest, address, src0</td>
</tr>
<tr>
<td>atomic wrapdec segment order scope equiv(n) _TypeLength</td>
<td>dest, address, src0</td>
</tr>
<tr>
<td>atomic max segment order scope equiv(n) _TypeLength</td>
<td>dest, address, src0</td>
</tr>
<tr>
<td>atomic min segment order scope equiv(n) _TypeLength</td>
<td>dest, address, src0</td>
</tr>
<tr>
<td>atomic cas segment order scope equiv(n) _TypeLength</td>
<td>dest, address, src0, src1</td>
</tr>
</tbody>
</table>

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 (on page 32).

order: Memory order used to specify synchronization. Can be rlx (relaxed) and scaq (sequentially consistent acquire) for all instructions, and for all instructions except ld can also be screl (sequentially consistent release) or scar (sequentially consistent acquire and release). See 6.2.1 Memory order (on page 183).

scope: Memory scope used to specify synchronization. Can be wave (wavefront) and wg (work-group) for global or group segments, and for global segment can also be agent (kernel agent) or system (system). For a flat address, can be wave, wg, agent, and system, but if the address references the group segment, agent and system behave as if wg was specified. See 6.2.2 Memory scope (on page 184).

equiv(n): Optional: n is an equivalence class. Used to specify the equivalence class of the memory locations being accessed. If omitted, class 0 is used, which indicates that any memory location may be aliased. See 6.1.4 Equivalence classes (on page 182).

Type: a for ld, and, or, xor, exch, cas; u and s for add, sub, max, min; u for wrapinc, wrapdec. See Table 4–2 (on page 108).

Length: 32, 64. See Table 4–2 (on page 108). 64 is not allowed for small machine model. See 2.9 Small and large machine models (on page 40).

Explanation of Operands (see 4.16 Operands (on page 113))

def: Destination register.

address: Source location in the specified segment. Must be an address expression for an address in segment (see 4.18 Address expressions (on page 115)).

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

Exceptions (see Chapter 12 Exceptions (on page 288))

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 (on page 399).

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) $s2 $s1, [&x];
atomic ld_global_scacq_agent_b32 $s1, [&x];
atomic ld_global_group_scacq_wg_b32 $s1, [&x];
atomic ld_scacq_system_b64 $d1, [$d0];

atomic_and_global_scacq_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_scacq_system_b64 $d1, [&x], 23;
atomic_or_global_scrl_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_scacq_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_scrl_agent_b64 $d1, [$d0], 23;

atomic_add_global_scacq_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_scrl_system_s64 $d1, [$d0], 23;

atomic_sub_global_scacq_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_scacq_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_scacq_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_scacq_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_s64 $d1, [$d0], 23;

atomic_min_global_scacq_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_s64 $d1, [$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 (on page 183)).

6.7.1 Syntax

Table 6–4 Syntax for atomic no return instructions

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomicnoret_st_segment_order_scopeequiv(n)_TypeLength</td>
<td>address, src0</td>
</tr>
<tr>
<td>atomicnoret_and_segment_order_scopeequiv(n)_TypeLength</td>
<td>address, src0</td>
</tr>
<tr>
<td>atomicnoret_or_segment_order_scopeequiv(n)_TypeLength</td>
<td>address, src0</td>
</tr>
<tr>
<td>atomicnoret_xor_segment_order_scopeequiv(n)_TypeLength</td>
<td>address, src0</td>
</tr>
<tr>
<td>atomicnoret_add_segment_order_scopeequiv(n)_TypeLength</td>
<td>address, src0</td>
</tr>
<tr>
<td>atomicnoret_sub_segment_order_scopeequiv(n)_TypeLength</td>
<td>address, src0</td>
</tr>
<tr>
<td>atomicnoret_wrapinc_segment_order_scopeequiv(n)_TypeLength</td>
<td>address, src0</td>
</tr>
<tr>
<td>atomicnoret_wrapdec_segment_order_scopeequiv(n)_TypeLength</td>
<td>address, src0</td>
</tr>
<tr>
<td>atomicnoret_max_segment_order_scopeequiv(n)_TypeLength</td>
<td>address, src0</td>
</tr>
<tr>
<td>atomicnoret_min_segment_order_scopeequiv(n)_TypeLength</td>
<td>address, src0</td>
</tr>
</tbody>
</table>

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 (on page 32).

**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 scas (sequentially consistent acquire and release). See 6.2.1 Memory order (on page 183).

**scope**: Memory scope used to specify synchronization. Can be wave (wavefront) and wg (work-group) for global or group segments, and for global segment can also be agent (kernel agent) or system (system). For a flat address, any value can be used, but if the address references the group segment, agent and system behave as if wg was specified. See 6.2.2 Memory scope (on page 184).

**equiv(n)**: Optional: n is an equivalence class. Used to specify the equivalence class of the memory locations being accessed. If omitted, class 0 is used, which indicates that any memory location may be aliased. See 6.1.4 Equivalence classes (on page 182).

**Type**: b for st, and or, xor; u and a for add, sub, max, min; u for wrapinc, wrapdec. See Table 4–2 (on page 108).

**Length**: 32, 64. See Table 4–2 (on page 108). 64 is not allowed for small machine model. See 2.9 Small and large machine models (on page 40).

Explanation of Operands (see 4.16 Operands (on page 113))

**address**: Source location in the specified segment. Must be an address expression for an address in segment (see 4.18 Address expressions (on page 115)).

**src0**: Source. Can be a register or immediate value.

Exceptions (see Chapter 12 Exceptions (on page 288))

Invalid address exceptions are allowed. May generate a memory exception if address is unaligned.
6.7.2 Description

For BRIG syntax, see 18.7.2 BRIG syntax for memory instructions (on page 399).

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

**Examples**

```bash
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_SCREL_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_Wave b64 [x], 23;
atOMICNORET_XOR_Rlx_System b64 [d0], 23;
atOMICNORET_ADD_GLOBAL_Scar_System_u64 [x], 23;
atOMICNORET_ADD_GLOBAL_Rlx_System_s64 [x], 23;
atOMICNORET_ADD_GROUP_Rlx_Wave 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_Wave 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_Wave_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_Wave_u64 [x], 23;
atOMICNORET_WRAPDEC_Rlx_System_u64 [d0], 23;
atOMICNORET_MAX_GLOBAL_Scar_System_u64 [x], 23;
atOMICNORET_MAX_GROUP_Rlx_Wave_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_Wave_u64 [x], 23;
atOMICNORET_MIN_Rlx_System_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 1.2, 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 (on page 40)). 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 instructions 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 (on page 183)) as if the signal value resides in global segment memory, is naturally aligned (see 6.1.3 Alignment (on page 182)) 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 lds 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 typed constant uses the typed constant notation (see 4.8.3 Typed constants (on page 94)): 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 (on page 101)):

- 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:

```c
global_s32 &name0 = sig32(0);
global_s32 &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

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>signal ld order TypeLength signalType</code></td>
<td>dest, signalHandle</td>
</tr>
<tr>
<td><code>signal and order TypeLength signalType</code></td>
<td>dest, signalHandle, src0</td>
</tr>
<tr>
<td><code>signal or order TypeLength signalType</code></td>
<td>dest, signalHandle, src0</td>
</tr>
<tr>
<td><code>signal xor order TypeLength signalType</code></td>
<td>dest, signalHandle, src0</td>
</tr>
<tr>
<td><code>signal exch order TypeLength signalType</code></td>
<td>dest, signalHandle, src0</td>
</tr>
<tr>
<td><code>signal add order TypeLength signalType</code></td>
<td>dest, signalHandle, src0</td>
</tr>
<tr>
<td><code>signal sub order TypeLength signalType</code></td>
<td>dest, signalHandle, src0</td>
</tr>
<tr>
<td><code>signal cas order TypeLength signalType</code></td>
<td>dest, signalHandle, src0, src1</td>
</tr>
<tr>
<td><code>signal wait waitOp order TypeLength signalType</code></td>
<td>dest, signalHandle, src0</td>
</tr>
<tr>
<td><code>signal waittimeout waitOp order TypeLength signalType</code></td>
<td>dest, signalHandle, src0, timeout</td>
</tr>
<tr>
<td><code>signalnoret st order TypeLength signalType</code></td>
<td>signalHandle, src0</td>
</tr>
<tr>
<td><code>signalnoret and order TypeLength signalType</code></td>
<td>signalHandle, src0</td>
</tr>
<tr>
<td><code>signalnoret or order TypeLength signalType</code></td>
<td>signalHandle, src0</td>
</tr>
<tr>
<td><code>signalnoret xor order TypeLength signalType</code></td>
<td>signalHandle, src0</td>
</tr>
<tr>
<td><code>signalnoret add order TypeLength signalType</code></td>
<td>signalHandle, src0</td>
</tr>
<tr>
<td><code>signalnoret sub order TypeLength signalType</code></td>
<td>signalHandle, src0</td>
</tr>
</tbody>
</table>
### 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 (on page 183).

**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 a for add, sub; a for wait, waittimeout. See Table 4-2 (on page 108).

**Length**: 32, 64. See Table 4-2 (on page 108). Must match the signal value size of signalType. See 2.9 Small and large machine models (on page 40).

**signalType**: sig32, sig64. See Table 4-4 (on page 110). Must be sig32 for small machine model and sig64 for large machine model. See 2.9 Small and large machine models (on page 40).

### Explanation of Operands (see 4.16 Operands (on page 113))

**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 (on page 288))

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 (on page 399).

### 6.8.2 Description

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 (on page 195).

The signalnoret instructions have the same definition as the corresponding atomicnoret instructions in a similar manner. See 6.7 Atomic no return (atomicnoret) instructions (on page 200)

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:

```c
timer.init(hsa_signal_get_timeout());
if {  
    original = [signal_value_address(signalHandle)];
} while (!original || waitOp || !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:

```c
// Wait for signal $d1 to be equal to 10
do {  
    signal_wait_eq_scacq_s64 $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 immediately report 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 Version 1.2, section 2.7 Requirement: HSA system timestamp). 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.wait0psrc0) && !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_screl_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 (on page 183).
6.9.1 Syntax

Table 6–6 Syntax for memfence instruction

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Explanation of Modifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>memfence</td>
<td></td>
</tr>
<tr>
<td><em>order</em> <em>scope</em></td>
<td></td>
</tr>
<tr>
<td><strong>order</strong>: Memory order used to specify synchronization. Can be <strong>scacq</strong> (sequentially consistent acquire), <strong>screl</strong> (sequentially consistent release) or <strong>scar</strong> (sequentially consistent acquire and release). See 6.2.1 Memory order (on page 183).</td>
<td></td>
</tr>
<tr>
<td><strong>scope</strong>: Memory scope used to specify synchronization. Can be <strong>wave</strong> (wavefront), <strong>wg</strong> (work-group), <strong>agent</strong> (kernel agent) or <strong>system</strong> (system). See 6.2.2 Memory scope (on page 184).</td>
<td></td>
</tr>
</tbody>
</table>

Exceptions (see Chapter 12 Exceptions (on page 288))

No exceptions are allowed.

For BRIG syntax, see 18.7.2 BRIG syntax for memory instructions (on page 399).

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 (on page 183).

For example:

```c
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

```c
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 (on page 294).

The minimum limits with respect to images are specified in Appendix A Limits (on page 420).

NOTE: For background information, see:

- *The OpenCL Specification Version 2.0*:
  - 5.3 Image Objects
- *The OpenCL C Specification Version 2.0*:
  - 6.13.14 Image Read and Write Functions
  - 5. Image Addressing and Filtering
- *The OpenCL Extension Specification Version 2.0*:

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,
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 size
- Image format
- Image access permission
- Reference to the actual image data
- Optionally, the image data layout

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 (on the facing 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 (on page 213).

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 (on page 220).

Images are accessed using image coordinates. See 7.1.6 Image coordinate (on page 222).

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, image data, and optionally the image data layout. Images are referenced by image instructions using an opaque image handle. See 7.1.7 Image creation and image handles (on page 227).

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 (on page 232).
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 (on page 234).

The image memory model defines the interaction of image operations between different work-items and other agents. See 7.1.10 Image memory model (on page 236).

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

<table>
<thead>
<tr>
<th>Image Geometry</th>
<th>Coordinates</th>
<th>Channel Orders</th>
<th>Image Operations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1d</td>
<td>width</td>
<td>unused</td>
<td>rdimage_1d, ldimage_1d, stimage_1d</td>
<td>one-dimensional image</td>
</tr>
<tr>
<td>2d</td>
<td>width</td>
<td>height</td>
<td>rdimage_2d, ldimage_2d, stimage_2d</td>
<td>two-dimensional image</td>
</tr>
<tr>
<td>3d</td>
<td>width</td>
<td>height</td>
<td>rdimage_3d, ldimage_3d, stimage_3d</td>
<td>three-dimensional image</td>
</tr>
<tr>
<td>1da</td>
<td>width</td>
<td>array index</td>
<td>rdimage_1da, ldimage_1da, stimage_1da</td>
<td>one-dimensional image array</td>
</tr>
<tr>
<td>2da</td>
<td>width</td>
<td>height</td>
<td>rdimage_2da, ldimage_2da, stimage_2da</td>
<td>two-dimensional image array</td>
</tr>
<tr>
<td>1db</td>
<td>width</td>
<td>unused</td>
<td>ldimage_1db, stimage_1db</td>
<td>one-dimensional image buffer</td>
</tr>
<tr>
<td>2ddepth</td>
<td>width</td>
<td>height</td>
<td>rdimage_2ddepth, ldimage_2ddepth, stimage_2ddepth</td>
<td>two-dimensional depth image</td>
</tr>
<tr>
<td>2ddepth</td>
<td>width</td>
<td>height</td>
<td>rdimage_2ddepth, ldimage_2ddepth, stimage_2ddepth</td>
<td>two-dimensional depth image array</td>
</tr>
</tbody>
</table>

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 regardless of the address mode specified by the sampler (even if undefined).

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 regardless of the address mode specified by the sampler (even if undefined).

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 \texttt{rdimage} image instruction does not support 1DB images.

An important difference between 1DB and 1D images is that 1DB images can have larger limits on the maximum image size supported, and always use the linear image data layout (see 7.1.7 Image creation and image handles (on page 227)). On some implementations this may result in a 1DB image having lower performance than an equivalent 1D image.

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 \texttt{depth} or \texttt{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 \texttt{depth} or \texttt{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 18.3.35 `hsa_ext_brig_image_geometry_t` (on page 358).

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 exceptions are 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. (See 7.1.6.2 Addressing mode (on page 223)).

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 (on the next page) for a list of the channel orders supported and their associated border colors.
Table 7–2 Channel order properties

<table>
<thead>
<tr>
<th>Channel Order</th>
<th>Memory Components</th>
<th>Access Components</th>
<th>Border Color</th>
<th>Channel Types</th>
<th>Image Geometries</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>(a)</td>
<td>(0,0,0,a)</td>
<td>(0,0,0,0)</td>
<td>snorm_int8, snorm_int16, unorm_int8, unorm_int16, signed_int8, signed_int16, signed_int32, unsigned_int8, unsigned_int16, unsigned_int32, half_float, float</td>
<td>1D, 2D, 3D, 1DA, 2DA, 1DB</td>
</tr>
<tr>
<td>r</td>
<td>(r)</td>
<td>(r,0,0,1)</td>
<td>(0,0,0,1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rx</td>
<td>(r)</td>
<td>(r,0,0,1)</td>
<td>(0,0,0,0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rg</td>
<td>(r,g)</td>
<td>(r,g,0,1)</td>
<td>(0,0,0,1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rgx</td>
<td>(r,g)</td>
<td>(r,g,0,1)</td>
<td>(0,0,0,0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ra</td>
<td>(r,a)</td>
<td>(r,0,0,a)</td>
<td>(0,0,0,0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rgba</td>
<td>(r,g,b,a)</td>
<td>(r,g,b,a)</td>
<td>(0,0,0,0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rgb</td>
<td>(r,g,b)</td>
<td>(r,g,b,1)</td>
<td>(0,0,0,1)</td>
<td>unorm_short_565, unorm_short_555, unorm_int_101010</td>
<td></td>
</tr>
<tr>
<td>rgbx</td>
<td>(r,g,b)</td>
<td>(r,g,b,1)</td>
<td>(0,0,0,0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bgra</td>
<td>(b,g,r,a)</td>
<td>(r,g,b,a)</td>
<td>(0,0,0,0)</td>
<td>unorm_int8, snorm_int8, signed_int8, unsigned_int8</td>
<td></td>
</tr>
<tr>
<td>argb</td>
<td>(a,r,g,b)</td>
<td>(r,g,b,a)</td>
<td>(0,0,0,0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>abgr</td>
<td>(a,b,g,r)</td>
<td>(r,g,b,a)</td>
<td>(0,0,0,0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>srgb</td>
<td>(r,g,b)</td>
<td>(r,g,b,1)</td>
<td>(0,0,0,1)</td>
<td>unorm_int8 (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.)</td>
<td></td>
</tr>
<tr>
<td>srgbx</td>
<td>(r,g,b)</td>
<td>(r,g,b,1)</td>
<td>(0,0,0,0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>srgba</td>
<td>(r,g,b,a)</td>
<td>(r,g,b,a)</td>
<td>(0,0,0,0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sbgra</td>
<td>(b,g,r,a)</td>
<td>(r,g,b,a)</td>
<td>(0,0,0,0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>intensity</td>
<td>(i)</td>
<td>(i,i,i)</td>
<td>(0,0,0,0)</td>
<td>unorm_int8, unorm_int16, snorm_int8, snorm_int16, half_float, float</td>
<td></td>
</tr>
<tr>
<td>luminance</td>
<td>(l)</td>
<td>(L,L,L,1)</td>
<td>(0,0,0,1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>depth</td>
<td>(d)</td>
<td>(d)</td>
<td>(0)</td>
<td>unorm_int16, unorm_int24, float</td>
<td>2DDEPTH, 2DADEPTH</td>
</tr>
<tr>
<td>depth_stencil</td>
<td>(d,s)</td>
<td>(d)</td>
<td>(0)</td>
<td>unorm_int24, float (The stencil value s is not available in HSAIL.)</td>
<td></td>
</tr>
</tbody>
</table>

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 rgb 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 61966-2-1, at IEC (International Electrotechnical Commission).
The `srgb`, `srgbx`, `srgba`, `sbgra` channel orders differ from the corresponding non-`s`-forms in that they convert the `r`, `g`, and `b` components from linear RGB to sRGB values when storing to memory, and from sRGB to linear RGB on read. When a sampler is used with linear filtering, the conversion is done before the weighted average is computed.

The `s`-form channel orders always use the `unorm_int8` channel type (see 7.1.4.2 Channel type (on the next page)).

The `a` channel of an `s`-form channel order, if present, is always treated as linear, and the UnsignedNormalize conversion method is used (see 7.1.4.2 Channel type (on the next page)).

On a read image instruction, the access component for an `r`, `g`, and `b` channel of an `s`-form channel order, if present, is set to:

```c
srgb_to_linear_rgb(srgb) {
    return (srgb ≤ 0.04045) ? (srgb / 12.92) :
        (((srgb + 0.055) / 1.055) ** 2.4);
}
```

```c
access_component =
    min(max(srgb_to_linear_rgb(float(memory_component) / 255.0), 0.0), 1.0);
```

The result must be in the closed interval of the infinitely accurate result produced by `srgb_to_linear_rgb(float(memory_component) ± 0.5) / 255.0`, with the additional requirements:

- If memory component is 0 must return 0.0.
- If memory component is 255 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 for an `r`, `g`, and `b` channel of an `s`-form channel order, if present, is set to:

```c
linear_rgb_to_srgb(linear_rgb) {
    return (linear_rgb > 1.0) || linear_rgb > plus_infinity) ? 1.0
        : (linear_rgb < 0.0) || linear_rgb < minus_infinity) ? 0.0
        :
            (linear_rgb < 0.0031308) ? (linear_rgb * 12.92)
        : ((1.055 * linear_rgb**0.4) - 0.055);
}
```

```c
memory_component =
    min(max(int_to_lin((linear_rgb_to_srgb(access_component) * 255.0) ± 0.5), 0), 255);
```

The conversion to integer uses zero rounding mode (see 5.19.4 Description of integer rounding modes (on page 175)). The result must be in the closed interval of the infinitely accurate result produced by `int_to_lin((linear_rgb_to_srgb(access_component) * 255.0) ± 0.6 + 0.5), with the additional requirements:

- If access component is 0.0 must return 0.
- If access component is 1.0 then must return 255.
- Must return a value in the closed interval [0, 255].
- No invalid operation exception is generated if the access component 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>
<thead>
<tr>
<th>Channel Type</th>
<th>Memory Type</th>
<th>Access Type</th>
<th>Conversion Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>snorm_int8</td>
<td>8</td>
<td>signed integer</td>
<td>f16 f32</td>
</tr>
<tr>
<td>snorm_int16</td>
<td>16</td>
<td>signed integer</td>
<td></td>
</tr>
<tr>
<td>unorm_int8</td>
<td>8</td>
<td>unsigned integer</td>
<td></td>
</tr>
<tr>
<td>unorm_int16</td>
<td>16</td>
<td>unsigned integer</td>
<td></td>
</tr>
<tr>
<td>unorm_int24</td>
<td>24</td>
<td>unsigned integer</td>
<td></td>
</tr>
<tr>
<td>unorm_short_565</td>
<td>r=5 bits[15:11]</td>
<td>unsigned integer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>g=6 bits[10:05]</td>
<td>unsigned integer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b=5 bits[04:00]</td>
<td>unsigned integer</td>
<td></td>
</tr>
<tr>
<td>unorm_short_555</td>
<td>r=5 bits[14:10]</td>
<td>unsigned integer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>g=5 bits[09:05]</td>
<td>unsigned integer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b=5 bits[04:00]</td>
<td>unsigned integer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ignored bit[15]</td>
<td>unsigned integer</td>
<td></td>
</tr>
<tr>
<td>Channel Type</td>
<td>Memory Type</td>
<td>Bit Size</td>
<td>Encoding</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>unorm int _ 101010</td>
<td>r=10 bits [29:20]</td>
<td>unsigned integer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>g=10 bits [19:10]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>b=10 bits [09:00]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ignored bits [31:30]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>signed int8</td>
<td>8</td>
<td>signed integer</td>
<td></td>
</tr>
<tr>
<td>signed int16</td>
<td>16</td>
<td>signed integer</td>
<td></td>
</tr>
<tr>
<td>signed int32</td>
<td>32</td>
<td>signed integer</td>
<td></td>
</tr>
<tr>
<td>unsigned int8</td>
<td>8</td>
<td>unsigned integer</td>
<td></td>
</tr>
<tr>
<td>unsigned _ int16</td>
<td>16</td>
<td>unsigned integer</td>
<td></td>
</tr>
<tr>
<td>unsigned _ int32</td>
<td>32</td>
<td>unsigned integer</td>
<td></td>
</tr>
<tr>
<td>half_float</td>
<td>16</td>
<td>float</td>
<td>f16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>f32</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>float</td>
<td>f32</td>
</tr>
</tbody>
</table>

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 (on page 118)). 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 (on page 108)).

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 sign or payload are preserved (regardless of the profile specified), or if signaling NaNs are converted to quiet NaNs (see 4.19.4 Not a Number (NaN) (on page 120)). All other values are preserved. Exceptions must not be generated, including invalid operation exception if the value is a signaling NaN, or inexact exception.
HalfFloat()

On a read image instruction, the memory component value is converted from \( f^{16} \) to \( f^{32} \). The conversion must be exact for both normal and subnormal values. An infinity value must be converted to the corresponding infinity value. A NaN value must be converted to a NaN value, however it is implementation defined if the NaN sign is preserved or payload is propagated (regardless of the profile specified) or if a signaling NaN is converted to a quiet NaN (see 4.19.4 Not a Number (NaN) (on page 120)). Exceptions must not be generated, including invalid operation exception if the value is a signaling NaN.

On write image instructions, the access component value is converted from \( f^{32} \) to \( f^{16} \). It is implementation defined whether near or zero rounding mode is used (see 4.19.2 Floating-point rounding (on page 118)). It is implementation defined if subnormal values resulting from the conversion are flushed to zero. The conversion is computed as described in 5.19.5 Description of Floating-Point Rounding Modes (on page 186); except that exceptions must not be generated (including invalid operation exception if the value is a signaling NaN or inexact exception); and it is implementation defined if NaN payloads are propagated (regardless of the profile specified), or if signaling NaNs are converted to quiet NaNs (see 4.19.4 Not a Number (NaN) (on page 120)).

UnsignedClamp(size)

The unsigned integer access component value is clamped to be in the unsigned integer memory component value closed interval \([0, 2^{\text{size}} - 1]\).

On a read image instruction, the access component is set to the memory component value zero extended to \text{u32}.

On write image instructions, the memory component value is set to:

\[
\text{memory\_component} = \min(\text{access\_component}, 2^{\text{size}} - 1);
\]

SignedClamp(size)

The signed integer access component value is clamped to be in the signed integer memory component value closed interval \([-2^{\text{size} - 1}, +2^{\text{size} - 1} - 1]\).

On a read image instruction, the access component is set to the memory component value sign extended to \text{s32}.

On write image instructions, the memory component value is set to:

\[
\text{memory\_component} = \min(\max(\text{access\_component}, -2^{\text{size} - 1}), 2^{\text{size} - 1} - 1);
\]

UnsignedNormalize(size)

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 \([0, 2^{\text{size}} - 1]\), with values outside that range (including infinity) being clamped and NaN values treated as 0.

On a read image instruction, the access component is set to:

\[
\text{access\_component} = \min(\max(\text{float}(\text{memory\_component}) / \text{float}(2^{\text{size}} - 1), 0.0), 1.0);
\]

This must be done with less than or equal to 1.5 ULP of the access type (see 4.19.6 Unit of least precision (ULP) (on page 121)), with the additional requirements:

- If memory component is 0 must return 0.0.
- If memory component is \(2^{\text{size}} - 1\) 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:

\[
\text{memory\_component} = \text{int\_neari\_sat} (\text{access\_component} \times \text{float}(2^{\text{size}} - 1));
\]

The conversion to integer uses neari_sat rounding mode using an unsigned integer type with destLength of size (see 5.19.4 Description of integer rounding modes (on page 175)). The result must be in the closed interval of the infinitely accurate result produced for the access component value \(\pm (0.6 / \text{float}(2^{\text{size}} - 1))\). Exceptions must not be generated, including invalid operation exception if the value is a signaling NaN, or those that may be generated if clamping is performed.

No inexact exception is generated for either conversion.

SignedNormalize(size)

A floating-point access component value in the closed interval \([-1.0 \text{ to } +1.0]\) is scaled to the signed integer memory component value closed interval \([-2^{\text{size}}^{-1}, +2^{\text{size}}^{-1} - 1]\), with values outside that range (including infinity) being clamped and NaN values treated as 0.

On a read image instruction, the access component is set to:

\[
\text{access\_component} = \min (\max (\text{float}(\text{memory\_component}) / \text{float}(2^{\text{size}} - 1), -1.0), 1.0);
\]

This must be done with less than or equal to 1.5 ULP of the access type, with the additional requirements:

- If memory component is \(-2^{\text{size}}^{-1}\) or \(-2^{\text{size}}^{-1} + 1\) then must return \(-1.0\).
- If memory component is 0 must return 0.0.
- If memory component is \(2^{\text{size}}^{-1} - 1\) 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:

\[
\text{memory\_component} = \text{int\_neari\_sat} (\text{access\_component} \times \text{float}(2^{\text{size}} - 1));
\]

The conversion to integer uses neari_sat rounding mode using an unsigned integer type with destLength of size (see 5.19.4 Description of integer rounding modes (on page 175)). The result must be in the closed interval of the infinitely accurate result produced for the access component value \(\pm (0.6 / \text{float}(2^{\text{size}} - 1))\). Exceptions must not be generated, including invalid operation exception if the value is a signaling NaN, or those that may be generated if clamping is performed.

No inexact exception is generated for either conversion.

UnsignedSrgb()

See 7.1.4.1.2 Standard RGB (s-form) channel orders (on page 214).

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 (on page 221).
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 (on the 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

<table>
<thead>
<tr>
<th>Channel Order</th>
<th>Channel Type</th>
<th>Image Geometry</th>
<th>bpp</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>Bits</td>
<td>unorm</td>
<td>snorm</td>
</tr>
<tr>
<td>8</td>
<td>Y (ro,wo, rw)</td>
<td>Y (ro, wo)</td>
<td>Y (ro,wo, rw)</td>
</tr>
<tr>
<td>16</td>
<td>Y (ro, wo)</td>
<td>Y (ro, wo)</td>
<td>Y (ro, wo, rw)</td>
</tr>
<tr>
<td>32</td>
<td>Y (ro, wo, rw)</td>
<td>Y (ro, wo, rw)</td>
<td>Y (ro, wo, rw)</td>
</tr>
<tr>
<td>rx, a</td>
<td>8</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>16</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>32</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>rg</td>
<td>8, 8</td>
<td>Y (ro, wo)</td>
<td>Y (ro, wo)</td>
</tr>
<tr>
<td>16, 16</td>
<td>Y (ro, wo)</td>
<td>Y (ro, wo)</td>
<td>Y (ro, wo, rw)</td>
</tr>
<tr>
<td>32, 32</td>
<td>Y (ro, wo)</td>
<td>Y (ro, wo)</td>
<td>Y (ro, wo)</td>
</tr>
<tr>
<td>rgx, ra</td>
<td>8, 8</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>16, 16</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>32, 32</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>rgb, rgbx</td>
<td>5, 6, 5</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>5, 5, 5, 1</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10, 10, 10, 2</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rgba</td>
<td>8, 8, 8, 8</td>
<td>Y (ro, wo, rw)</td>
<td>Y (ro, wo)</td>
</tr>
<tr>
<td>16, 16, 16</td>
<td>Y (ro, wo)</td>
<td>Y (ro, wo)</td>
<td>Y (ro, wo, rw)</td>
</tr>
<tr>
<td>32, 32, 32</td>
<td>Y (ro, wo, rw)</td>
<td>Y (ro, wo, rw)</td>
<td>Y (ro, wo, rw)</td>
</tr>
<tr>
<td>bgra</td>
<td>8, 8, 8, 8</td>
<td>Y (ro, wo)</td>
<td>Y</td>
</tr>
<tr>
<td>argb, abgr</td>
<td>8, 8, 8, 8</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>argb, argbx</td>
<td>8, 8, 8</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>argba</td>
<td>8, 8, 8, 8</td>
<td>Y (ro)</td>
<td></td>
</tr>
<tr>
<td>sbgra</td>
<td>8, 8, 8, 8</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>intensity, luminance</td>
<td>8</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 7.1 Images in HSAIL

<table>
<thead>
<tr>
<th>Channel Order</th>
<th>Bits</th>
<th>Channel Type</th>
<th>Image Geometry</th>
<th>bpp</th>
</tr>
</thead>
<tbody>
<tr>
<td>depth</td>
<td>16</td>
<td>Y (ro,wo)</td>
<td>2DDEPTH, 2DADEPTH</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>Y</td>
<td></td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>Y (ro,wo)</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>depth_stencil</td>
<td>24, 8</td>
<td>Y</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>Y</td>
<td></td>
<td>64</td>
</tr>
</tbody>
</table>

#### 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 (on page 211).

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 (on page 232)). For the ldimage and stimage image instructions, fixed modes are used (see 7.1.3 Filter mode (on page 225)). 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 7.1.3 Filter mode (on page 225)). 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 (on the 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.

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, coord_dim) where coord_dim is the size of the corresponding dimension. It is specified as an f32 coordinate data type value. The value is multiplied by coord_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 (coord_is_array_index ? unnormalized : normalization_mode) { 
    case unnormalized: 
        switch (coord_data_type) { 
        case u32: 
            switch (filter_mode) { 
            case nearest: return float(coord); 
            } 
        case s32: 
            switch (coord_is_array_index ? nearest : filter_mode) { 
            case nearest: return float(coord); 
            case linear: return float(coord) - 0.5; 
            } 
        case f32: 
            if (coord_is_nan or coord_is_infinity) return is_undefined; 
            switch (coord_is_array_index ? nearest : filter_mode) { 
            case nearest: return coord; 
            case linear: return coord - 0.5; 
            } 
        } 
        case normalized: 
            switch (coord_data_type) { 
            case f32: 
                if (coord_is_nan or coord_is_infinity) return is_undefined; 
                switch (filter_mode) { 
                case nearest: return coord * coord_dim; 
                case linear: return (coord * coord_dim) - 0.5; 
                } 
            } 
        } 
    } 
}
```

7.1.6.2 Addressing mode

The coordinate addressing mode controls how out of range coordinates are processed:

undefined
The image 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 (on page 214)).

**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 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 (even if undefined).

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 (on page 175)).

The addressing mode is computed as follows:

```c
addressing(coord) {
  if (coord_is_undefined) return is_undefined;
  out_of_range = (int_down1(coord) < 0) or (int_down1(coord) > coord_dim - 1);
  if (coord_is_array_index) {
    if (out_of_range and
      ((operation == stimage) or (operation == ldimage))
      return is_undefined;
    return max(0, min(int_neari(coord), coord_dim - 1));
  }
  if ((normalization_mode == unnormalized) and
      ((addressing_mode == repeat) or (addressing_mode == mirrored_repeat)))
    return is_undefined;
  if (not_out_of_range) return int_down1(coord);
  switch(addressing_mode) {
    case undefined:  return is_undefined;
    case clamp_to_edge:  return int_down1(max(0, min(coord, coord_dim - 1)));
    case clamp_to_border:  return is_border;
    case repeat:
      tile = int_down1(coord / coord_dim);
      return int_down1(coord - (tile * coord_dim));
    case mirrored_repeat:
      mirrored_coord = (coord < 0) ? (-coord - 1) : coord;
      tile = int_down1(mirrored_coord / coord_dim);
      mirrored_coord = int_down1(mirrored_coord - (tile * coord_dim));
      if (tile & 1) {
```
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 1D 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 only supported for images with a floating-point access type, and 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 (on page 211)). Of the used coordinate indices, if any are is_undefined, then the behavior of 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 (on page 214)). 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 (on page 216) and 7.1.4.1.2 Standard RGB (s-form) channel orders (on page 214).

load_texel and store_texel map between memory components and access components as shown in Table 7-2 (on page 214). If the image channel order has fewer than four memory components:

- load_texel returns the fixed value from Table 7-2 (on page 214) 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 nearest filter mode.
- ldimage always uses unnormalized normalization mode, undefined addressing mode, and
nearest 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 (on page 232)).

The filter mode is computed as follows:

```hsail
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 2ddepth:
              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 `load_texel` returns a NaN or Infinity value for any texel then the result is implementation defined.
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 computed must match the infinitely accurate result. For all other combinations, the precision and accuracy of the filter mode computations, including associated image element index 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 (width): in elements for one, two and three dimensional image data geometries.
- Height size (height): in elements for two and three dimensional image data geometries.
- Depth size (depth): in elements for three dimensional image data geometries.
- Array size (array_size): in number of image layers for image array geometries.

The image data row pitch (row_pitch) is the size in bytes for a single row. It must be greater than or equal to the width * bpp/8, and be a multiple of bpp/8.

The image data slice pitch (slice_pitch) is the size in bytes of a single 2D slice of a 3D image, or the size in bytes of each image in a 1DA, 2DA, or 2DADEPTH image array. It must be greater than or equal to row_pitch * height for a 3D image, 2DA image array, or 2DADEPTH image array, and greater than or equal to row_pitch for a 1DA image array. It must be a multiple of row_pitch.

The row and slice pitch may include additional padding between the image rows and slices to ensure alignment which can improve performance.

An image layout identifier denotes such aspects of image data layout as tiling and organization of channels in memory. Some image layout identifiers may only apply to specific image geometries, formats, and access permissions. Different agents may support different image layout identifiers, including vendor specific layouts. Note that an agent may not support the same image layout identifier for different access permissions to images with the same image geometry, size, and format. The HSA runtime defines the available image layout identifiers and provides queries to determine which image layout identifiers are supported by an agent and for which image geometries, formats, and access permissions. If multiple agents support the same image layout identifier then it is possible to use separate image handles for each agent that references the same image data.
The HSA runtime defines the linear image layout identifier which has the following image data layout specified in ascending byte address order. For a 3D image, 2DA or 2DDEPTH image array, or 1DA image array the image data is stored as a linear sequence of adjacent 2D image slices, 2D images, or 1D images respectively, spaced according to slice_pitch. Each 2D or 2DDEPTH image is stored as a linear sequence of adjacent image rows, spaced according to row_pitch. Each 1D or 1DB image is stored as a single image row. Each image row is stored as a linear sequence of image elements. Each image element is stored as a linear sequence of image memory components specified by the left to right channel order definition. Each image memory component is stored using the memory type specified by the channel type. The row_pitch of a linear image must be a multiple of the linear image data row pitch alignment for the agents that will access the image data using image instructions. An HSA runtime query can be used to return the linear image data row pitch alignment for a specific agent and must be a power of two. The image data size of a linear image is: slice_pitch * depth for a 3D image; row_pitch * height for a 2D or 2DDEPTH image; slice_pitch * array_size for a 2DA, 2DDEPTH, and 1DA image array; and row_pitch for a 1D and 1DB image.

The 1DB image geometry always uses the linear image data layout.

An image layout consists of an image layout identifier, image data row pitch, and image data slice pitch according to the image geometry.

The HSA runtime can be used in two ways to determine the image data size and alignment required for an image of a specific configuration:

- Opaque image layout (this corresponds to the opaque image data layout defined in the HSA Platform System Architecture Specification Version 1.2, section 2.16 Requirement: Images). An HSA runtime query can be used to return the image data size and image data alignment required for an image of a specific size, geometry, format, and access permission for a specific agent. The HSA runtime will return an error if the specified agent does not support the requested image configuration. The size and alignment is implementation dependent for each agent as they may use different image data layouts and different row and slice padding.

- Explicit image layout. An HSA runtime query can be used to return the image data size and image data alignment required for an image of a specific size, geometry, format, and image layout identifier for a specific agent. The image data row and slice pitch can optionally be specified: if specified they must conform with the rules above, and be supported by the agent; if not specified for the linear image layout they default to the smallest values satisfying the rules above; otherwise they default to implementation dependent values based on the agent and image layout identifier. The HSA runtime will return an error if the agent does not support the requested image configuration.

The image data can be allocated using the determined image data size and image data alignment, that is accessible to image operations executed on the agent(s) using a specific access permission.

The HSA runtime can be used to create an opaque image handle by specifying:

- Agent
- Image geometry
- Image width, height, depth, and array size according to the image geometry
- Image format
- Image access permission
- Address of image data
- Optional image layout

If the image layout is not specified, then an opaque image layout is being used. The other arguments must match those specified when the image data size and alignment was determined for the opaque image layout. The image data layout used is implementation defined, except for 1DB images which always use the linear image data layout.

If the image layout is specified, then an explicit image layout is being used. The other arguments must match those specified when the image data size and alignment was determined for the explicit image layout.

The HSA runtime will return an error if the agent does not support the requested image configuration.

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 (on page 221)). 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 HSA Platform System Architecture Specification Version 1.2, Appendix A Limits). 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 program execution is undefined if an image handle is used after it has been destroyed.

The program execution is undefined if multiple image handles are created to the same image data unless:

- The image data was allocated such that it is accessible to the agent and accesses to the image data conform to the image memory model (see the HSA Platform System Architecture Specification Version 1.2, section 2.16 Requirement: Images).
- The image data must satisfy the size and alignment of the agent.
- The image geometry, size, and format are the same. The exceptions are that:
  - If the image format channel order 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 (on page 214)).
  - If the image format channel order is z then it can be depth and vice versa.
- If an opaque image layout is used, an implementation defined image layout is used and the agent must be the same for all the image handles. In addition, the image access permission must also match unless the agent uses the same image layout for all image access permissions with the specified image geometry, size, and format. There is an HSA runtime query to determine if the same image layout is used.
- If an explicit image layout is used for one of the image handles, the same explicit image layout must be used for all the image handles.

The HSA runtime provides operations to convert between a linear image layout and the possibly implementation defined image 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 (on page 110)). These correspond to the three image access permissions (see 7.1.5 Image access permission (on page 220)). 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>
<thead>
<tr>
<th>Image Handle Type</th>
<th>Image Access Permission</th>
<th>Image Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>roimg</td>
<td>ro</td>
<td>rdimage, ldimage</td>
</tr>
<tr>
<td>woimg</td>
<td>wo</td>
<td>stimage</td>
</tr>
<tr>
<td>rwimg</td>
<td>rw</td>
<td>ldimage, stimage</td>
</tr>
</tbody>
</table>

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 program execution is 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. The program execution is undefined if an HSAIL image instruction is used 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 layouts. Also see 7.1.10 Image memory model (on page 236).

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 (on page 73).
An image handle typed constant uses the typed constant notation (see 4.8.3 Typed constants (on page 94)): 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 (on page 101)):

- 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:

```c
 alloc(agent) global_roimg &name0 = roimg(geometry = 3d,
                                          width = 5, height = 4, depth = 6,
                                          channel_type = unorm_int_101010,
                                          channel_order = rgb);

 decl prog global_roimg &name1;
 decl prog global_roimg &ArrayOfroimg[10];
 alloc(agent) global_woimg &name3 = woimg(geometry = 3d,
                                          width = 5, height = 4, depth = 6,
                                          channel_type = unorm_int_101010,
                                          channel_order = rgb);

 decl prog global_rwimg &namedrwimg12;
 decl prog global_rwimg &namedrwimg2;
 decl prog global_rwimg &namedrwimg3;
 decl prog global_rwimg &ArrayOfroimgs[10];
 alloc(agent) global_rwimg &namedrwimgWithInit[2] =
              rwimg[](rwimg(geometry = 3d,
                            width = 5, height = 4, depth = 6,
                            channel_type = unorm_int_101010,
                            channel_order = rgb),
                             rwimg(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 an agent 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 (on page 50).

The queryimage instruction can be used to query the properties of an image. See 7.5 Query image and query sampler instructions (on page 242).

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 (on page 222)).

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 (on page 420)). 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 (on page 110)). 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 (on page 73).

A sampler handle typed constant uses the typed constant notation (see 4.8.3 Typed constants (on page 94)): 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 (on page 101)):

- 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:

```c
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_edge),
   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_edge)
   };
```

When an agent 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 (on page 50).
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 (on page 225)).

Samplers cannot be used with 1DB images which are not supported by the rdimage image instruction. The querysampler instruction can be used to query the properties of a sampler. See 7.5 Query image and query sampler instructions (on page 242).

### 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 (on page 211).
- 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 (on page 221).
- 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 (on page 221).

#### Table 7–6 Image instruction combinations

<table>
<thead>
<tr>
<th>Image Instruction</th>
<th>Image Handle Type</th>
<th>Coordinate Type</th>
<th>Access Type</th>
<th>Sampler</th>
<th>Image Geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>coord</td>
<td>filter</td>
<td>addressing</td>
<td></td>
</tr>
<tr>
<td>rdimage u32, s32, f16, f32</td>
<td>roimg, wimg</td>
<td>s32</td>
<td>unnormalized</td>
<td>nearest</td>
<td>1D, 2D, 2DA, 2DADEPTH, 2DADEPTH(1D, 2D, 2DA, 2DADEPTH array index coordinate always treated as unnormalized, clamp_to_edge)</td>
</tr>
<tr>
<td>f32</td>
<td>u32, s32</td>
<td>nearest, linear</td>
<td>nearest_to_edge, clamp_to_edge, clamp_to_border</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f16, f32</td>
<td>u32, s32</td>
<td>nearest, linear</td>
<td>nearest_to_edge, clamp_to_edge, clamp_to_border, repeat, mirrored_repeat</td>
<td></td>
<td></td>
</tr>
<tr>
<td>u32, s32</td>
<td>f16, f32</td>
<td>normalized</td>
<td>nearest_to_edge, clamp_to_edge, clamp_to_border, repeat, mirrored_repeat</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldimage u32, s32, f16, f32</td>
<td>roimg, wimg</td>
<td>s32</td>
<td>Sampler not allowed (undefined if coordinate not in range 0 to dimension size – 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stimage wimg, wimg</td>
<td>u32, s32, f16, f32</td>
<td>Sampler not allowed (undefined if coordinate not in range 0 to dimension size – 1)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To access the data in an image, an image handle is loaded into a register using a load (ld) instruction with a source type of roimg, wimg or rimg. 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 handle 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 program execution is 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 variable's value type (roimg, woimg, or rwimg) must match the type of all instructions that use the value. For sampler handles, the original variable's value type 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 queriesampler). 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 program execution is undefined if an image instruction (rdimage, ldimage, and stimage) or queryimage instruction with an image handle value that is not compatible. The image handle value 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 (on page 230).

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 (on page 232).
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 program execution is 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 1.2, section 2.16 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 (on page 230).

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 (on page 225)).

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 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 7.6 Image fence (imagefence) instruction (on page 244)).

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 (on page 244)).
   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 1.2, 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 accesses to image data by image instructions and memory instructions using the global segment are only made coherent at kernel dispatch granularity using the user mode queue packet fences.

8. Any access to image data using global segment memory instructions must use acquire and release memory ordering at an appropriate memory scope in order to allow sharing. See 6.2 Memory model (on page 183).

9. Access to the image data using both global segment memory instructions and image instructions is undefined unless the image data is made coherent between the global segment memory instruction accesses and image instruction accesses by appropriate user mode queue packet memory fences (see HSA Platform System Architecture Specification Version 1.2, section 2.9.1 Packet header).

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

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rdimage_v4_1d_equiv(n)_destType_imageType_coordType</code></td>
<td><code>(destR, destG, destB, destA), image, sampler, coordWidth</code></td>
</tr>
<tr>
<td><code>rdimage_v4_2d_equiv(n)_destType_imageType_coordType</code></td>
<td><code>(destR, destG, destB, destA), image, sampler, (coordWidth, coordHeight)</code></td>
</tr>
<tr>
<td><code>rdimage_v4_3d_equiv(n)_destType_imageType_coordType</code></td>
<td><code>(destR, destG, destB, destA), image, sampler, (coordWidth, coordHeight, coordDepth)</code></td>
</tr>
<tr>
<td><code>rdimage_v4_1da_equiv(n)_destType_imageType_coordType</code></td>
<td><code>(destR, destG, destB, destA), image, sampler, (coordWidth, coordHeight, coordArrayIndex)</code></td>
</tr>
<tr>
<td><code>rdimage_v4_2da_equiv(n)_destType_imageType_coordType</code></td>
<td><code>(destR, destG, destB, destA), image, sampler, (coordWidth, coordHeight, coordArrayIndex)</code></td>
</tr>
<tr>
<td><code>rdimage_2ddepth_equiv(n)_destType_imageType_coordType</code></td>
<td><code>destR, image, sampler, (coordWidth, coordHeight)</code></td>
</tr>
<tr>
<td><code>rdimage_2ddepth_equiv(n)_destType_imageType_coordType</code></td>
<td><code>destR, image, sampler, (coordWidth, coordHeight, coordArrayIndex)</code></td>
</tr>
</tbody>
</table>

#### Explanation of Modifiers

- **v4:** If present, specifies the instruction returns 4 components, otherwise only 1 component is returned.
- **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 (on page 182).
- **destType:** Destination type: u32, s32, f16, or f32. See Table 4–2 (on page 108).
- **imageType:** Image object type: roimg. See Table 4–4 (on page 110).
- **coordType:** Source coordinate element type: s32 or f32. See Table 4–2 (on page 108).

#### Explanation of Operands (see 4.16 Operands (on page 113))

- **destR, destG, destB, destA:** Destination. Must be an s register.
- **image:** A source operand a register that contains a value of an image object of type `imageType`. See 7.1.7 Image creation and image handles (on page 227) and 7.1.9 Using image instructions (on page 234).
- **sampler:** A source operand a register that contains a value of a sampler object. It is always of type aamp. See 7.1.8 Sampler creation and sampler handles (on page 232) and 7.1.9 Using image instructions (on page 234).
- **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 (on page 288))

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 (on page 400).
7.2.2 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 \textit{image} at coordinates given by source operands \textit{coordWidth}, \textit{coordHeight}, \textit{coordDepth}, and \textit{coordArrayIndex}, into destination operands \textit{destR}, \textit{destG}, \textit{destB}, and \textit{destA}. A sampler specified by source operand \textit{sampler} defines how to process the read.

\textit{rdimage} used with integer coordinates has restrictions on the sampler:

- \textit{coord} must be unnormalized.
- \textit{filter} must be nearest.
- The boundary mode must be \texttt{undefined}, \texttt{clamp_to_edge} or \texttt{clamp_to_border}.

1DB images are not supported.

<table>
<thead>
<tr>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{ld} \textit{_global_roimg} $d1, {&amp;roimg1};</td>
</tr>
<tr>
<td>\textit{ld} \textit{kernarg} \textit{roimg} $d2, {&amp;roimg2};</td>
</tr>
<tr>
<td>\textit{ld} \textit{readonly_samp} $d3, {&amp;samp};</td>
</tr>
<tr>
<td>\textit{rdimage} \textit{v4} \textit{ld} \textit{equiv} (12) \textit{s32} \textit{roimg} \textit{f32} ($s0, $s1, $s3, $s4), $d1, $d3, $s6;</td>
</tr>
<tr>
<td>\textit{rdimage} \textit{v4} \textit{ld} \textit{equiv} (12) \textit{s32} \textit{roimg} \textit{f32} ($s0, $s1, $s3, $s4), $d0, $d3, $s6, $s9;</td>
</tr>
<tr>
<td>\textit{rdimage} \textit{v4} \textit{ld} \textit{equiv} (12) \textit{s32} \textit{roimg} \textit{f32} ($s0, $s1, $s3, $s4), $d2, $d3, $s6, $s9, $s2;</td>
</tr>
<tr>
<td>\textit{rdimage} \textit{v4} \textit{ld} \textit{equiv} (12) \textit{s32} \textit{roimg} \textit{f32} ($s0, $s1, $s2, $s3), $d1, $d3, $s6, $s7;</td>
</tr>
<tr>
<td>\textit{rdimage} \textit{v4} \textit{ld} \textit{equiv} (12) \textit{s32} \textit{roimg} \textit{f32} ($s0, $s1, $s3, $s4), $d1, $d3, ($s6, $s9, $s12);</td>
</tr>
<tr>
<td>\textit{rdimage} \textit{v4} \textit{ld} \textit{equiv} (12) \textit{s32} \textit{roimg} \textit{f32} $s0, $d2, $d3, ($s6, $s9);</td>
</tr>
<tr>
<td>\textit{rdimage} \textit{v4} \textit{ld} \textit{equiv} (12) \textit{s32} \textit{roimg} \textit{f32} $s0, $d2, $d3, ($s6, $s9, $s10);</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{ldimage} \textit{v4} \textit{ld} \textit{equiv}(n) \textit{destType}_imageType_coordType</td>
<td>(\textit{destR}, \textit{destG}, \textit{destB}, \textit{destA}), \textit{image}, \textit{coordWidth}</td>
</tr>
<tr>
<td>\textit{ldimage} \textit{v4} \textit{ld} \textit{equiv}(n) \textit{destType}_imageType_coordType</td>
<td>(\textit{destR}, \textit{destG}, \textit{destB}, \textit{destA}), \textit{image}, (\textit{coordWidth}, \textit{coordHeight})</td>
</tr>
<tr>
<td>\textit{ldimage} \textit{v4} \textit{ld} \textit{equiv}(n) \textit{destType}_imageType_coordType</td>
<td>(\textit{destR}, \textit{destG}, \textit{destB}, \textit{destA}), \textit{image}, (\textit{coordWidth}, \textit{coordHeight}, \textit{coordDepth})</td>
</tr>
<tr>
<td>\textit{ldimage} \textit{v4} \textit{ld} \textit{equiv}(n) \textit{destType}_imageType_coordType</td>
<td>(\textit{destR}, \textit{destG}, \textit{destB}, \textit{destA}), \textit{image}, (\textit{coordWidth}, \textit{coordHeight}, \textit{coordArrayIndex})</td>
</tr>
<tr>
<td>\textit{ldimage} \textit{v4} \textit{ld} \textit{equiv}(n) \textit{destType}_imageType_coordType</td>
<td>(\textit{destR}, \textit{destG}, \textit{destB}, \textit{destA}), \textit{image}, (\textit{coordWidth}, \textit{coordHeight}, \textit{coordArrayIndex})</td>
</tr>
<tr>
<td>\textit{ldimage} \textit{v4} \textit{ld} \textit{equiv}(n) \textit{destType}_imageType_coordType</td>
<td>(\textit{destR}, \textit{destG}, \textit{destB}, \textit{destA}), \textit{image}, \textit{coordByteIndex}</td>
</tr>
<tr>
<td>\textit{ldimage} \textit{v4} \textit{ld} \textit{equiv}(n) \textit{destType}_imageType_coordType</td>
<td>\textit{destR}, \textit{image}, (\textit{coordWidth}, \textit{coordHeight})</td>
</tr>
<tr>
<td>\textit{ldimage} \textit{v4} \textit{ld} \textit{equiv}(n) \textit{destType}_imageType_coordType</td>
<td>\textit{destR}, \textit{image}, (\textit{coordWidth}, \textit{coordHeight}, \textit{coordArrayIndex})</td>
</tr>
</tbody>
</table>
### 7.3 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 program execution is undefined if a coordinate is out of bounds (that is, greater than the dimension of the image or less than 0).

The differences between the ldimage 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, [4rwimg1];
ld_kernarg_roimg $d2, [3roimg2];
ldimage_v4 ld equiv(12) f32_rwimg u32 ($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, $s7);
ldimage_v4 1da f32_rwimg u32 ($s1, $s2, $s3, $s4), $d1, ($s5, $s6); 
ldimage_v4 2da f32_roimg u32 ($s1, $s2, $s3, $s4), $d2, ($s5, $s6, $s7);
ldimage_v4 2dimage_f32_roimg u32 $s1, $d1, ($s5, $s6);
ldimage_2ddepth_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

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>stimage_v4 1d equiv(n) <em>srcType</em> _imageType_coordType</td>
<td>(srcR, srcG, srcB, srcA), image, coordWidth</td>
</tr>
<tr>
<td>stimage_v4 2d equiv(n) <em>srcType</em> _imageType_coordType</td>
<td>(srcR, srcG, srcB, srcA), image, (coordWidth, coordHeight)</td>
</tr>
<tr>
<td>stimage_v4 3d equiv(n) <em>srcType</em> _imageType_coordType</td>
<td>(srcR, srcG, srcB, srcA), image, (coordWidth, coordHeight, coordDepth)</td>
</tr>
<tr>
<td>stimage_v4 1da equiv(n) <em>srcType</em> _imageType_coordType</td>
<td>(srcR, srcG, srcB, srcA), image, (coordWidth, coordArrayIndex)</td>
</tr>
<tr>
<td>stimage_v4 2da equiv(n) <em>srcType</em> _imageType_coordType</td>
<td>(srcR, srcG, srcB, srcA), image, (coordWidth, coordArrayIndex)</td>
</tr>
<tr>
<td>stimage_v4 1db equiv(n) <em>srcType</em> _imageType_coordType</td>
<td>(srcR, srcG, srcB, srcA), image, coordArrayIndex</td>
</tr>
<tr>
<td>stimage_2ddepth equiv(n) <em>srcType</em> _imageType_coordType</td>
<td>srcR, image, (coordWidth, coordHeight)</td>
</tr>
<tr>
<td>stimage_2ddepth equiv(n) <em>srcType</em> _imageType_coordType</td>
<td>srcR, image, (coordWidth, coordHeight, coordArrayIndex)</td>
</tr>
</tbody>
</table>

Explanation of Modifiers

v4: If present, specifies the instruction takes 4 components, otherwise only 1 component is taken.

1d, 2d, 3d, 1da, 2da, 1db, 2ddepth, 2ddepth: Image geometry. Specifies the number and meaning of coordinates required to access an image element. Can be 1d or 1db (width); 2d or 2ddepth (width and height); 3d (width, height, and depth); 1da (width and array index); or 2da or 2ddepth (width, height and array index). See 7.1.3 Image geometry (on page 211).

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 (on page 182).

srcType: Source type: u32, s32, f16, or f32. See Table 4-2 (on page 108).

imageType: Image object type: woimg, rwimg. See Table 4-4 (on page 110).

coordType: Source coordinate element type: u32. See Table 4-2 (on page 108).
Chapter 7. Image Instructions  
7.5 Query image and query sampler instructions

### Explanation of Operands (see 4.16 Operands (on page 113))

- **srcR, srcG, srcB, srcA**: Source. Can be a register or immediate value.
- **image**: A source operand is a register that contains a value of an image object of type `imageType`. See 7.1.7 Image creation and image handles (on page 227) and 7.1.9 Using image instructions (on page 234).
- **coordWidth, coordHeight, coordDepth, coordArrayIndex**: A source register or immediate value of type `coordType` that specifies the coordinates being read.

### Exceptions (see Chapter 12 Exceptions (on page 288))

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 (on page 400).

#### 7.4.2 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`.

The program execution is undefined if a coordinate is used 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

```plaintext
ld_global_woiimg $d1, [%roimg1];
ld_global_rwimg $d2, [%rwimg1];
stimage_v4_ld_equiv(12)_f32_woiimg_u32 ($s1, $s2, $s3, $s4), $d1, $s5;
stimage_v4_2d_f32_woiimg_u32 ($s1, $s2, $s3, $s4), $d1, ($s5, $s6);
stimage_v4_3d_f32_woiimg_u32 ($s1, $s2, $s3, $s4), $d1, ($s5, $s6, $s7);
stimage_v4_1d_f32_rwimg_u32 ($s1, $s2, $s3, $s4), $d2, ($s5, $s6);
stimage_v4_2d_rwimg_u32 ($s1, $s2, $s3, $s4), $d2, ($s5, $s6, $s7);
stimage_v4_3d_rwimg_u32 ($s1, $s2, $s3, $s4), $d2, $s5;
stimage_2d_depth_f32_rwimg_u32 $s1, $d2, ($s5, $s6);
stimage_2d_2d_depth_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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>queryimage_geometry_imageProperty_destType_imageType</td>
<td>dest, image</td>
</tr>
<tr>
<td>queriesampler_samplerProperty_destType</td>
<td>dest, sampler</td>
</tr>
</tbody>
</table>

Explanation of Modifiers

- **geometry**: Image geometry: 1d, 2d, 3d, 1da, 2da, 1db, 2ddepth, 2ddepth. See 7.1.3 Image geometry (on page 211).
- **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 (on the next page).
- **destType**: Destination type: u32. See Table 4–2 (on page 108).
- **imageType**: Image object type: roimg, wimg, rwimg. See Table 4–4 (on page 110).

Explanation of Operands (see 4.16 Operands (on page 113))

- **dest**: Destination register of type u32.
- **image**: A source operand and register that contains a value of an image object of type imageType. See 7.1.7 Image creation and image handles (on page 227) and 7.1.9 Using image instructions (on page 234).
- **sampler**: A source operand and register that contains a value of a sampler object. It is always of type samp. See 7.1.8 Sampler creation and sampler handles (on page 232) and 7.1.9 Using image instructions (on page 234).

Exceptions (see Chapter 12 Exceptions (on page 288))

No exceptions are allowed.

For BRIG syntax, see 18.7.3 BRIG syntax for image instructions (on page 400).

7.5.2 Description

Each query returns a 32-bit value giving a property of the source:

Table 7–11 Explanation of imageProperty modifier

<table>
<thead>
<tr>
<th>imageProperty</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>image width in elements. Allowed for all image geometries.</td>
</tr>
<tr>
<td>height</td>
<td>image height in elements. Only allowed for 2D, 3D, 2DA, 2DDEPTH or 2DADEPTH image geometries.</td>
</tr>
<tr>
<td>depth</td>
<td>image depth in elements. Only allowed for 3D image geometry.</td>
</tr>
<tr>
<td>array</td>
<td>The number of image layers. Only allowed for 1DA, 2DA or 2DADEPTH image geometries.</td>
</tr>
<tr>
<td>channelorder</td>
<td>An image channel order property encoded as an integer according to 18.3.33 hsa_ext_brig_image_channel_order_t (on page 357).</td>
</tr>
<tr>
<td>channeltype</td>
<td>An image channel type property encoded as an integer according to 18.3.34 hsa_ext_brig_image_channel_type_t (on page 357).</td>
</tr>
</tbody>
</table>
7.6 Image fence (imagefence) instruction

The image fence (imagefence) instruction synchronizes image operations. See 6.2 Memory model (on page 183) and 7.1.10 Image memory model (on page 236).

7.6.1 Syntax

Table 7–13 Syntax for imagefence instruction

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Explanation of Modifiers</th>
<th>Exceptions (see Chapter 12 Exceptions (on page 288))</th>
</tr>
</thead>
<tbody>
<tr>
<td>imagefence</td>
<td>No modifiers are allowed.</td>
<td>No exceptions are allowed.</td>
</tr>
</tbody>
</table>

For BRIG syntax, see 18.7.3 BRIG syntax for image instructions (on page 400).

7.6.2 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.

Table 7–12 Explanation of samplerProperty modifier

<table>
<thead>
<tr>
<th>samplerProperty</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>addressing</td>
<td>A sampler addressing mode property encoded as an integer according to 18.3.37 hsa_ext Brig sampler addressing t (on page 359). If undefined was specified when the sampler was initialized, it is implementation defined what addressing mode is returned. It may be any of the addressing modes, including undefined.</td>
</tr>
<tr>
<td>coord</td>
<td>A sampler coordinate property encoded as an integer according to 18.3.38 hsa_ext Brig sampler coord normalization t (on page 359).</td>
</tr>
<tr>
<td>filter</td>
<td>A sampler filter property encoded as an integer according to 18.3.39 hsa_ext Brig sampler filter t (on page 359).</td>
</tr>
</tbody>
</table>

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_1d_array_u32_roimg $s1, $d2;
queryimage_2d_array_u32_regrading $s0, $d2;
queryimage_2d_channeldtype_u32_roimg $s0, $d3;
queryimage_2d_channeldtype_u32_roimg $s0, $d3;
querysampler_addressing_u32 $s0, $d4;
querysampler_coord_u32 $s0, $d4;
querysampler_filter_u32 $s0, $d4;
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:

```hsa
stimage_v4_1d_f32_rwimg_f32 ($s1, $s2, $s3, $s4), $d1, $s4;
imagefence; // Will ensure image data stored by stimage is visible to
// subsequently ldimage in same work-item.
ldimage_v4_1d_f32_rwimg_f32 ($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:

```hsa
stimage_v4_1d_f32_rwimg_f32 ($s1, $s2, $s3, $s4), $d1, $s4;
imagefence;
barrier;
imagefence;
ldimage_v4_1d_f32_rwimg_f32 ($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 to 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:

```hsa
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 (on page 236).
### 8.1 Syntax

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>br</code></td>
<td><code>label</code></td>
</tr>
<tr>
<td><code>cbr_width_b1</code></td>
<td><code>src, label</code></td>
</tr>
<tr>
<td><code>sbr_width_uLength</code></td>
<td><code>src [labellist]</code></td>
</tr>
</tbody>
</table>

**Explanation of Modifiers**

- `width`: Optional: `width(n)`, `width(WAVESIZE)`, or `width(all)`. The width modifier specifies the result uniformity of the target for branches. All active work-items in the same slice are guaranteed to branch to the same target. If the width modifier is omitted, it defaults to `width(1)`, indicating each active work-item can branch independently. See 2.12.2 Using the width modifier with control transfer instructions (on page 45).
- `Length`: 32, 64.

**Explanation of Operands**

- `src`: Source. Can be a register or immediate value.
- `label`: Must be an identifier of a label in the same code block as the branch instruction.
- `labellist`: 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**

No exceptions are allowed.

For BRIG syntax, see 18.7.4 BRIG syntax for branch instructions (on page 401).

### 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 (on page 263).

- `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`. 

Like many programming languages, HSAIL supports branch instructions that can alter the control flow.
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 machine code. See 2.12 Divergent control flow (on page 42).

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 program execution is undefined if the number of labels in labelList is less than or equal to the index value. src can either be of type u32 or u64.

Since a switch branch can potentially transfer to more than one target, it can result in control flow divergence which can introduce a performance issue. The width modifier can be used to specify properties about the control flow divergence that may result in the finalizer producing more efficient machine code. See 2.12 Divergent control flow (on page 42).

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.
See 2.12 Divergent control flow (on page 42) for a full list of communication operations.

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

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>barrier_width</td>
<td></td>
</tr>
<tr>
<td>wavebarrier</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Explanation of Modifiers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>width: Optional: width(n), width(WAVESIZE), or width(all). Used to specify the communication uniformity among the work-items of a work-group. If omitted, defaults to width(all). See the Description below.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exceptions (see Chapter 12 Exceptions (on page 288))</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>No exceptions are allowed.</td>
<td></td>
</tr>
</tbody>
</table>

For BRIG syntax, see 18.7.5 BRIG syntax for parallel synchronization and communication instructions (on page 401).

9.1.2 Description

The barrier and wavebarrier instructions are execution barriers. See 9.3 Execution barrier (on page 257).

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 (on page 42)). 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 machine 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**

```c
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 (on page 72)). For example:  

```c
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 (on page 37)): 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.
Chapter 9. Parallel Synchronization and Communication Instructions  9.2 Fine-grain (fbarrier) instructions

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

<table>
<thead>
<tr>
<th>Opcodes</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>initfbar</td>
<td>src</td>
</tr>
<tr>
<td>jjoinfbar_width</td>
<td>src</td>
</tr>
<tr>
<td>waitfbar_width</td>
<td>src</td>
</tr>
<tr>
<td>arrivefbar_width</td>
<td>src</td>
</tr>
<tr>
<td>leavefbar_width</td>
<td>src</td>
</tr>
<tr>
<td>releasefbar</td>
<td>src</td>
</tr>
<tr>
<td>ldf_u32</td>
<td>dest, fbarrierName</td>
</tr>
</tbody>
</table>

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 (on page 42).

Explanation of Operands (see 4.16 Operands (on page 113))

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 (on page 288))

No exceptions are allowed.

For BRIG syntax, see 18.7.5 BRIG syntax for parallel synchronization and communication instructions (on page 401).

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 segment 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 (on page 42)). 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.

A work-item must not join an fbarrier that has an `arrive_count` that is not 0. This means that other work-items have arrived or are waiting at the same fbarrier that has not been satisfied. This requirement can be satisfied by using the `barrier` instruction, or the `waitfbar` instruction (on another fbarrier) between the `joinfbar` and the fbarrier instructions that use it.

`waitfbar`

Is an execution barrier, see 9.3 Execution barrier (on page 257).

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 (on page 42)). 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 (on page 257).
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 (on page 42)). 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 memfence 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 (on page 42)). 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 (on page 37)) 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 (on page 40)). 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 (on page 257).

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 Pseudo code 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.

```c
01: fbarrier %fbl;
02: if (workitemflatid_u32 == 0) {
03:   initfbar %fbl;
04: }
05: barrier;
06: joinfbar %fbl; // 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 %fbl; // fbl only has work-items for which
14:   memfence_scacq_system;
15: ...
16: leavefbar %fbl;
17: } else {
18:   leavefbar %fbl;
19: }
20: } else {
21:   leavefbar %fbl;
22: }
23: barrier;
24: if (workitemflatid_u32 == 0) {
```

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.

```c
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
16:     // join fb1 on line 14 or arrive at
17:     // line 23 or 26
18:     memfence_screl_system;
19:     waitfbar %fb1; // fb1 only has work-items for which
20:     // cond1 && cond2 is true
21:     memfence_scacq_system;
22:     ...
23:     leavefbar %fb1;
24:     } else {
25:         waitfbar %fb0;
26:     }
27: }
28: leavefbar %fb0;
29: barrier;
30: if (workitemflatid_u32 == 0) {
31:     releasefbar %fb0;
32:     releasefbar %fb1;
33: }
```

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.

```c
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 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 (on page 183)) 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. Initially each barrier_instance has arrived_count set to 0 and participant_count set to the number of members.

while (atomic ld group rlx wg(barrier_instance.arrived_count) < 0) sleep;
if (atomic add group rlx wg(barrier_instance.arrived_count, 1) ==
  (barrier_instance.participant_count - 1)) {
  atomic st group rlx wg(barrier_instance.arrived_count,
  -barrier_instance.participant_count - 1);
} else {
  while (atomic ld group rlx wg(barrier_instance.arrived_count) > 0) sleep;
  atomic add group rlx wg(barrier_instance.arrived_count, 1);
}

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 control flow ensures that there is an interleaving of relaxed atomic instructions to the same location such that every participating work-item performs a store/load before a load/store of every other participating work-item. 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.

Note that the barrier pseudo code is for illustration purposes only as such communication is not guaranteed to make forward progress. See 2.13 Forward progress (on page 46).

The fine-grain barriers require additional complexity to support the `arrivefbar` and `leavefbar` instructions. Since these instructions do not wait for all participating work-items to arrive at the `fbarrier`, acquire memory fences cannot be used after the `arrivefbar` and `leavefbar` instructions to make memory instructions performed by other participating work-items visible. Also note that the `participant_count` is initially 0 and is updated by `joinfbar` and `leavefbar` and so needs to be accessed using atomic instructions.

See 7.1.10 Image memory model (on page 236) for additional rules related to `imagefence`.

The clock (see 11.4 Miscellaneous instructions (on page 283)) and signal (see 6.8 Notification (signal) instructions (on page 202)) 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 (on the facing page)), `cleardetectexcept`, `getdetectexcept`, and `setdetectexcept` (see 11.2 Exception instructions (on page 279)) instructions are required to be execution uniform (see 2.12 Divergent control flow (on page 42)). 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 (on page 42).

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 `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 machine 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 machine code for such barriers. The `requiredworkgroupsize` and `maxflatworkgroupsize` control directives (see 13.5 Control directives for low-level performance tuning (on page 317)) 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 (on page 29)).

### 9.4.1 Syntax

Table 9-3 Syntax for cross-lane instructions

<table>
<thead>
<tr>
<th>Opcodes</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>activelaneCount_width_u32_b1</code></td>
<td><code>dest, src</code></td>
</tr>
<tr>
<td><code>activelaneId_width_u32</code></td>
<td><code>dest</code></td>
</tr>
<tr>
<td><code>activelanemask_v4_width_b64_b1</code></td>
<td><code>dest0, dest1, dest2, dest3, src</code></td>
</tr>
<tr>
<td><code>activelanepermute_width_blength</code></td>
<td><code>dest, src, laneId, identity, useIdentity</code></td>
</tr>
</tbody>
</table>

**Explanation of Modifier**

- **width**: Optional; `width(n), width(WAVESIZE),` or `width(all)`. Used to specify the execution uniformity among the work-items of a work-group. Each active lane in a wavefront can have different values for the source operands, and produce a different value, regardless of the width modifier. If the width modifier is omitted, it defaults to `width(1)`, indicating each lane of the wavefront can be independently active or inactive. See 2.12 Divergent control flow (on page 42).

- **Length**: 1, 32, 64, 128.
### Explanation of Operands (see 4.16 Operands (on page 113))

| dest, dest0, dest1, dest2, dest3: Destination register. |
| src, laneId, identity, useIdentity: Sources. Can be a register or immediate value. |

### Exceptions (see Chapter 12 Exceptions (on page 288))

No exceptions are allowed.

For BRIG syntax, see 18.7.5 BRIG syntax for parallel synchronization and communication instructions (on page 401).

#### 9.4.2 Description

**activelaneid**

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 *bl* 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 *bl*. *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 *bl*.

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 pseudo code:

```c
type result[WAVESIZE];
for(l = 0; l < WAVESIZE; ++l) {
    result[l] = identity;
    if (lane[l].active &&
        !lane[l].useIdentity &&
        lane[l].laneId % WAVESIZE].active) {  
        result[l] = lane[l].laneId % WAVESIZE].src;
    }
} 
for(l = 0; l < WAVESIZE; ++l) {
    if (lane[l].active) lane[l].dest = result[l];
}
```

Examples

activelanecount_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:

```hsail
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:

```hsail
// 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 (on page 66) and 4.3.3 Function (on page 61).

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 (on page 82).

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 (on page 267).
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 (on page 267)).

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 program execution is 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 (on page 46)) by the call instruction.

For the actual argument that corresponds to the output formal argument, the program execution is 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:
  - 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 (on page 269), 10.7 Switch call (scall) instruction (on page 270) and 10.8 Indirect call (icall) instruction (on page 271).

#### 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 (on page 61).

For example:

```c
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 (on page 61).

For example:

```c
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;
};
```

```c
function &caller()
{
  // ...
  {  
    arg_u32 %input1;
    arg_u32 %input2;
    arg_u32 %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 (on page 64).

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 (on page 271).

10.4 Variadic functions

A variadic function is a function that accepts a variable number of arguments.

In HSAIL, variadic functions are declared by specifying the last formal argument as an array with no specified size (for example, `uint32 extra_args[]`). The matching actual argument passed by a call 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];  // %n holds the number to add
  mov_b32 $s1, 0;         // %s1 holds the sum
  mov_b32 $s3, 0;         // %s3 is the offset into last
  @loop:
    cmp_eq_b1_u32 $c1, $s0, 0;  // see if the count is zero
    cbr_b1 $c1, %done;         // if it is, jump to done
    ld_arg_f32 $s4, [%last][%s3];  // load a value
    add_f32 $s3, $s3, 4;      // add the value
    add_f32 $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];
```
Chapter 10. Function Instructions  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 (on page 73).

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:

```c
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:

```c
function &foo(){(align(8) arg_b8 %z[1])
{
 // ...
    ret;
};
function &top(){
{
    // ...
    global_f64 %d;
    global_u8 %c[4];
    ld_global_f64 $d0, [%d];
    ld_global_u8 $s0, [%c];
    {        // ensures that sk starts on an 8-byte boundary so that both 32-bit and 64-bit stores are naturally aligned
        align(8) arg_b8 %sk[12];
        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

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>call</td>
<td>function (outputArgs) (inputArgs)</td>
</tr>
</tbody>
</table>

**Explanation of Operands (see 4.16 Operands (on page 113))**

- **function**: Must be the identifier of a function (either non-indirect or indirect). The function output and input formal arguments must match the `outputArgs` and `inputArgs` specified.
- **outputArgs**: List of zero or one call argument.
- **inputArgs**: List of zero or more comma-separated call arguments.

**Exceptions (see Chapter 12 Exceptions (on page 288))**

No exceptions are allowed.

For BRIG syntax, see 18.7.6 BRIG syntax for function instructions (on page 402).

10.6.2 Description

A direct call instruction transfers control to a specific function specified by the `function` operand. The 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 runtime executable must be used to provide the definition, such as to a host application variable. See 4.2 Program, code object, and executable (on page 50).

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 (on page 263).

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.
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

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scall_width_uLength</code></td>
<td><code>src (outputArgs) (inputArgs) [functionList]</code></td>
</tr>
</tbody>
</table>

**Explanation of Modifier**

- `width`: Optional: `width(n)`, `width(WAVESIZE)`, or `width(all)`. Used to specify the result uniformity of the target for switch calls. All active work-items in the same slice are guaranteed to call the same target. If the width modifier is omitted, it defaults to `width(1)`, indicating each active work-item can call a different target. See 2.12.2 Using the width modifier with control transfer instructions (on page 45).

  - `Length`: 32, 64.

**Explanation of Operands (see 4.16 Operands (on page 113))**

- `src`: Source. Can be a register or immediate value.
- `outputArgs`: List of zero or one call argument.
- `inputArgs`: List of zero or more comma-separated call arguments.
- `functionList`: Comma-separated list of global identifiers of both non-indirect and indirect functions. All functions must have the same input and output formal arguments, but do not have to match whether they are indirect functions or not. The functions' output and input formal arguments must match the `outputArgs` and `inputArgs` specified.

**Exceptions (see Chapter 12 Exceptions (on page 288))**

- No exceptions are allowed.

For BRIG syntax, see 18.7.6 BRIG syntax for function instructions (on page 402).

### 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 program execution is undefined if the number of functions in `functionList` is less than or equal to the index value. `src` can either be of type `u32` or `u64`.

---

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];
The functions in `functionList` can be either a non-indirect or an indirect function. 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 runtime executable must be used to provide the definition, such as to a host application variable. See 4.2 Program, code object, and executable (on page 50).

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 machine code. See 2.12 Divergent control flow (on page 42).

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 (on page 263).

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 by 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

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>icall_width_uLength</td>
<td>src (outputArgs) (inputArgs) signature</td>
</tr>
</tbody>
</table>

#### Explanation of Modifier

**width**: Optional: width(n), width(WAVESIZE), or width(all). 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 width(1), indicating each active work-item can call a different target. See 2.12.2 Using the width modifier with control transfer instructions (on page 45).

**Length**: 32, 64. Must match the size of an indirect function code handle (see Table 2-3 (on page 41)).

#### Explanation of Operands (see 4.16 Operands (on page 113))

- **outputArgs**: List of zero or one call argument.
- **inputArgs**: List of zero or more comma-separated call arguments.
- **src**: A register.
- **signature**: Global identifier of a signature. The signature output and input formal arguments must match the outputArgs and inputArgs specified.

#### Exceptions (see Chapter 12 Exceptions (on page 288))

No exceptions are allowed.

For BRIG syntax, see 18.7.6 BRIG syntax for function instructions (on page 402).

### 10.8.2 Description

The icall instruction is not supported by the base profile. See 16.2.1 Base profile requirements (on page 328).

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 program execution is undefined unless all the following are true:

- \texttt{src} is a valid indirect function code handle obtained from an agent code object that:
  - is currently loaded in the same executable specifying the same kernel agent as the currently executing kernel;
  - was loaded before the currently executing kernel was launched.
- \texttt{src} refers to an indirect function with formal input and output arguments that match \texttt{signature}.

See 4.2 Program, code object, and executable (on page 50).

At the time of finalizing, the actual indirect function that an \texttt{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 machine code. See 2.12 Divergent control flow (on page 42).

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 (on page 263).

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;

ekernel &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 \texttt{(ret)} instruction returns from a function back to the caller’s environment. \texttt{ret} can also be used to exit a kernel.

If there is no \texttt{ret} instruction before the exit of the kernel’s or function’s code block, the finalizer will act as if a \texttt{ret} instruction was present at the end of the code block.
10.9.1 Syntax

Table 10–4 Syntax for ret instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>ret</th>
</tr>
</thead>
</table>

Exceptions (see Chapter 12 Exceptions (on page 288))
No exceptions are allowed.

For BRIG syntax, see 18.7.6 BRIG syntax for function instructions (on page 402).

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>alloca_align(n)_u32</th>
</tr>
</thead>
</table>

| Operands | dest, src |

Explanation of Modifiers

align(n): Optional. Used to specify the byte alignment of the base of the memory being allocated. If omitted, 1 is used indicating no alignment. See the Description below.

Explanation of Operands (see 4.16 Operands (on page 113))

dest: Destination. Must be a 32-bit register.

src: Source. Can be a 32-bit register or immediate value. The value of src is the minimum amount of space (in bytes) requested.

Exceptions (see Chapter 12 Exceptions (on page 288))
No exceptions are allowed.

For BRIG syntax, see 18.7.6 BRIG syntax for function instructions (on page 402).
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 (on page 334).

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 (on page 263).

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

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>currentworkgroupsizex_u32</td>
<td>dest, dimNumber</td>
</tr>
<tr>
<td>currentworkitemflataxid_u32</td>
<td>dest</td>
</tr>
<tr>
<td>dim_u32</td>
<td>dest</td>
</tr>
<tr>
<td>gridgroups_u32</td>
<td>dest, dimNumber</td>
</tr>
<tr>
<td>gridsize_u32</td>
<td>dest, dimNumber</td>
</tr>
<tr>
<td>packetcompletionsig_signalType</td>
<td>dest</td>
</tr>
<tr>
<td>packetid_u64</td>
<td>dest</td>
</tr>
<tr>
<td>workgroupid_u32</td>
<td>dest, dimNumber</td>
</tr>
<tr>
<td>workgroupsize_u32</td>
<td>dest, dimNumber</td>
</tr>
<tr>
<td>workitemabsid_u32</td>
<td>dest, dimNumber</td>
</tr>
<tr>
<td>workitemflatabsid_uLength</td>
<td>dest</td>
</tr>
<tr>
<td>workitemflataxid_u32</td>
<td>dest</td>
</tr>
<tr>
<td>workitemid_u32</td>
<td>dest, dimNumber</td>
</tr>
</tbody>
</table>

Explanation of Modifiers

signalType: Must be sig32 for small machine model and sig64 for large machine model. See Table 4–4 (on page 110) and 2.9 Small and large machine models (on page 40).

Length: 32, 64.

Explanation of Operands (see 4.16 Operands (on page 113))

dest: Destination. For packetcompletionsig and packetid must be a d register; otherwise must be an a register. See Table 2–3 (on page 41).

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 (on page 288))

No exceptions are allowed.

For BRIG syntax, see 18.7.7.1 BRIG syntax for kernel dispatch packet instructions (on page 403).
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 (on page 26).

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 (on page 28).

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 (on page 24).

gridgroups
Returns the upper bound for work-group identifiers (IDs) (that is, the number of work-groups) within the grid for the `dimNumber` dimension and stores the result in the destination `dest`.

If the grid was launched with fewer dimensions than `dimNumber`, then `gridgroups` stores 1 in destination `dest`.

gridgroups is always equal to `gridsize` divided by `workgroupsize` rounded up to the nearest integer.

gridsize
Returns the upper bound for work-item absolute identifiers (IDs) within the grid for the `dimNumber` dimension and stores the result in the destination `dest`.

If the grid was launched with fewer dimensions than `dimNumber`, then `gridsize` stores 1 in destination `dest`.

packetcompletionsig
Returns the signal handle of the completion signal specified for this dispatch in `dest`. The value may be 0 indicating there is no associated completion signal (see 6.8 Notification (signal) instructions (on page 202)). See HSA Platform System Architecture Specification Version 1.2, section 2.8 Requirement: User mode queuing.

packetid
Returns a 64-bit 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 1.2, 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 (on page 26).
This instruction computes the three-dimensional ID of the work-group, selects the \( \text{dimNumber} \) dimension, and stores the result in the destination \( \text{dest} \).
If the grid was launched with fewer than three dimensions, \( \text{workgroupid} \) returns 0 for the unused dimensions.

workgroupsize
Accesses the work-group size specified when the kernel was dispatched for the \( \text{dimNumber} \) dimension and stores the result in the destination \( \text{dest} \). See 2.2 Work-groups (on page 26).
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 \( \text{currentworkgroupsize} \) 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 \( \text{currentworkgroupsize} \) and \( \text{workgroupsize} \) will always be the same, and it might be more efficient to use \( \text{workgroupsize} \).
If the kernel was dispatched with fewer dimensions than \( \text{dimNumber} \), then \( \text{workgroupsize} \) stores 1 in destination \( \text{dest} \).

workitemabsid
Accesses the work-item absolute identifier (ID) within the entire grid and stores the result for the \( \text{dimNumber} \) dimension in the destination \( \text{dest} \). See 2.3.3 Work-item absolute ID (on page 28).
If the work-group was launched with fewer dimensions than \( \text{dimNumber} \), \( \text{workitemabsid} \) stores 0 in destination \( \text{dest} \).

workitemflatabsid
Accesses the flattened form of the work-item absolute identifier (ID) within the entire grid and stores the result in the destination \( \text{dest} \). Can either be returned as a \( \text{u32} \) or \( \text{u64} \). If \( \text{u32} \) then the lower 32 bits of the ID are returned. See 2.3.4 Work-item flattened absolute ID (on page 28).

workitemflatid
Accesses the flattened form of the work-item identifier (ID) within the work-group and stores the result in the destination \( \text{dest} \). See 2.3.2 Work-item flattened ID and current work-item flattened ID (on page 28).

workitemid
Accesses the work-item identifier (ID) within the work-group and stores the result for the \( \text{dimNumber} \) dimension in the destination \( \text{dest} \). See 2.3.1 Work-item ID (on page 27).
If the work-group was launched with fewer dimensions than \( \text{dimNumber} \), \( \text{workitemid} \) stores 0 in the destination \( \text{dest} \).
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

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>cleardetectexcept_u32</td>
<td>exceptionsNumber</td>
</tr>
<tr>
<td>getdetectexcept_u32</td>
<td>dest</td>
</tr>
<tr>
<td>setdetectexcept_u32</td>
<td>exceptionsNumber</td>
</tr>
</tbody>
</table>

Explanation of Operands (see 4.16 Operands (on page 113))

dest: Destination. Must be an $s$ register. See Table 2–3 (on page 41).

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 (on page 288))

No exceptions are allowed.

For BRIG syntax, see 18.7.7.2 BRIG syntax for exception instructions (on page 403).
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 12.2 Divergent control flow (on page 42)), 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 12.2 Divergent control flow (on page 42)), 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 12.2 Divergent control flow (on page 42)), and might lead to deadlock.

11.2.3 Additional information

DETECT exception processing operates on the five exceptions specified in 12.2 Hardware exceptions (on page 288).

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 12.2 Divergent control flow (on page 42).

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 13.5 Control directives for low-level performance tuning (on page 317)). 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 machine code produced by the finalizer 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 (on page 335).

### Examples

```c
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 1.2, 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>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addqueuewriteindex_segment_order_u64</code></td>
<td><code>dest, address, src</code></td>
</tr>
<tr>
<td><code>casqueuewriteindex_segment_order_u64</code></td>
<td><code>dest, address, src0, src1</code></td>
</tr>
<tr>
<td><code>ldqueuereadindex_segment_order_u64</code></td>
<td><code>dest, address</code></td>
</tr>
<tr>
<td><code>ldqueuewriteindex_segment_order_u64</code></td>
<td><code>dest, address</code></td>
</tr>
<tr>
<td><code>stqueuereadindex_segment_order_u64</code></td>
<td><code>address, src</code></td>
</tr>
<tr>
<td><code>stqueuewriteindex_segment_order_u64</code></td>
<td><code>address, src</code></td>
</tr>
</tbody>
</table>

### Explanation of Modifiers

- **segment**: Optional segment. If omitted, flat is used. Only flat and global is allowed. See 2.8 Segments (on page 32).
- **order**: Memory order used to specify synchronization. See 6.2.1 Memory order (on page 183).
- **Length**: 32, 64. Must match the address size for the global segment (see Table 2-3 (on page 41)).
### Explanation of Operands (see 4.16 Operands (on page 113))

| dest: Destination. Must be a d register. |
| src, src0, src1: Sources. Can be a register or immediate value. |
| address: Address expression for an address in the specified segment for a user mode queue created by the HSA runtime (see 4.18 Address expressions (on page 115)). |

### Exceptions (see Chapter 12 Exceptions (on page 288))

No exceptions are allowed.

For BRIG syntax, see 18.7.3 BRIG syntax for user mode queue instructions (on page 404).

#### 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 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 packet IDs. 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 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 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 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 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

<table>
<thead>
<tr>
<th>Opcodes and Modifiers</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock_u64</td>
<td>dest</td>
</tr>
<tr>
<td>cuid_u32</td>
<td>dest</td>
</tr>
<tr>
<td>debugtrap_u32</td>
<td>src</td>
</tr>
<tr>
<td>groupbaseptr_u32</td>
<td>dest</td>
</tr>
<tr>
<td>groupstaticsize_u32</td>
<td>dest</td>
</tr>
<tr>
<td>grouptotalsize_u32</td>
<td>dest</td>
</tr>
<tr>
<td>kernargbaseptr_u32</td>
<td>dest</td>
</tr>
<tr>
<td>laneid_u32</td>
<td>dest</td>
</tr>
<tr>
<td>maxcuid_u32</td>
<td>dest</td>
</tr>
<tr>
<td>maxwaveid_u32</td>
<td>dest</td>
</tr>
<tr>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>nullptr_segment_uLength</td>
<td>dest</td>
</tr>
<tr>
<td>waveid_u32</td>
<td>dest</td>
</tr>
</tbody>
</table>

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 (on page 41)).

*segment*: Optional segment. If omitted, flat is used. Can be flat, group, private, and kernarg. See 2.8 Segments (on page 32).

Explanation of Operands (see 4.16 Operands (on page 113))

*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 (on page 41).

*src*: Source. Can be a register or immediate value.

Exceptions (see Chapter 12 Exceptions (on page 288))

No exceptions are allowed.

For BRIG syntax, see 18.7.7.4 BRIG syntax for miscellaneous instructions (on page 404).

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 Version 1.2*, section 2.7 Requirement: HSA system timestamp.

The *clock* instruction is treated as if it is a read-modify-write relaxed atomic memory instruction (see 6.2 Memory model (on page 183)). 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.

dbgtrap

Halts execution of the wavefront executing the instruction and generates a debug exception. See 12.4 Debug exceptions (on page 291).

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 (on page 291).

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 (on page 291). 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 defined statically and 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 static size determined when the kernel was finalized. The group segment static size is available using HSA runtime queries on the executable for the specific kernel, or by using the `groupstaticsize` instruction.

If the kernel dispatch uses dynamic group segment memory, it is allocated by setting a group segment size in the kernel dispatch packet that is larger than the group segment static size. The base of the dynamically allocated group segment memory for the work-group of a work-item is obtained by adding the group segment static size, to the group segment address returned by the `groupbaseptr` instruction.

The required alignment of the base of the dynamic group segment memory can be achieved using the `requiredgroupbaseptralign` control directive (see 13.5 Control directives for low-level performance tuning (on page 317)), together with rounding up the group segment static size to the same alignment using explicit instructions. Note that the group segment size specified in the kernel dispatch packet must account for any rounding done on the group segment static size.

See 4.20 Dynamic group segment memory allocation (on page 123) and 4.2 Program, code object, and executable (on page 50).

groupstaticsize

Returns the group segment static size determined when the kernel being executed was finalized, and stores the result in the destination `dest`. See the description of the `groupbaseptr` instruction.
grouptotalsize
Returns the total group segment byte size for the work-group of the work-item executing the instruction, and stores the result in the destination dest. Note that this value may be greater than the group segment size value specified in the kernel dispatch packet as an implementation may allocate group segment memory at larger than byte granularity. See 4.20 Dynamic group segment memory allocation (on page 123).

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 (on page 73)) with n larger than 16, then the returned address will have alignment at least the maximum n specified. See 4.21 Kernarg segment (on page 125).
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 machine 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).
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

```assembly
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
groupstaticsize_u32 $s2; // group segment static size
grouptotalsize_u32 $s2; // group segment total size
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 if it is active. See 12.4 Debug exceptions (on page 291).

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 (on page 290)). 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 (on page 328)), 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 (on page 291). 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 (on page 118).
- Underflow
A non-zero tiny floating-point value is computed and either:

- the \texttt{ftz} modifier was specified,
- or the \texttt{ftz} modifier was not specified and the value cannot be represented exactly.

See 4.19.2 Floating-point rounding (on page 118).

- Division by zero

A finite non-zero floating-point value is divided by zero.

It is implementation defined whether integer \texttt{div} or \texttt{rem} instructions with a divisor of zero will generate a divide by zero exception.

See 4.19.2 Floating-point rounding (on page 118).

- 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.
- \texttt{Multiplication: mul(0.0, infinity) or mul(infinity, 0.0)}.
- Fused multiply add: \texttt{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: \texttt{add(positive infinity, negative infinity), sub(positive infinity, positive infinity)}.
- Division: \texttt{div(0.0, 0.0) or div(infinity, infinity)}.
- Square root: \texttt{sqrt(negative)}.
- Conversion: A \texttt{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 (on page 118).
- In addition, it is implementation defined whether instructions with the \texttt{ftz} modifier that cause a value to be flushed to zero generate the inexact exception. See 4.19.3 Flush to zero (ftz) (on page 119).

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 \texttt{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 (on page 288). Whether either exception policy is supported by a kernel agent depends on the profile specified (see 16.2 Profile-specific requirements (on page 328)).

- 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 (on page 279).

- 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 signalled exceptions (on the facing 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 machine 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 (on page 335).

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 enabledetectexceptions 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 (on page 276)), 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 1.2 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 (on page 57)). 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.

![Figure 13-1 extension syntax diagram](image)

The syntax is:

```
extension name { : major : minor } ;
```

**name**

The name of the extension as a string literal. See 4.5 Strings (on page 80).

Vendor specific extension names should have the syntax `vendor:label`, where `vendor` is the (possibly abbreviated) name of the vendor, and `label` is one or more words separated by underscores `_` that provides a short name for the extension. Both the vendor name and the label words must consist of only lowercase letters and digits. The vendor name `hsa` is reserved for use by HSA Foundation specifications.

**major**

An integer literal of type `u64` and must be in the right-open interval `[0, 2^{32})`. WAVESIZE is not allowed. Specifies the major version of the extension. See 4.8.1 Integer constants (on page 86).

**minor**

An integer literal of type `u64` and must be in the right-open interval `[0, 2^{32})`. WAVESIZE is not allowed. Specifies the minor version of the extension. See 4.8.1 Integer constants (on page 86).

Version A of an extension must be backwards compatible with version B of the same extension if their major versions are the same and the minor version of A is greater than B.
Chapter 13. Directives  13.1 extension directive

The version number of an extension is optional, and if omitted defaults to the HSAIL version number specified by the module header (see Chapter 14 module Header (on page 323)).

An extension with an empty name has no effect, and the version is ignored. The finalizer must report no errors for such extensions.

An extension with the same non-empty name can be specified multiple times in a module provided they all specify the same version (either explicitly or by default), otherwise the finalizer must report an error.

For vendor specific extensions, any additional names introduced in HSAIL must be prefixed with the extension's vendor name and a colon (:). For example, any new instructions, instruction modifiers, data types, memory segments, and so forth.

For example, if a finalizer from a vendor named foo was to support an extension named bar for version 0.1, an application could enable it using code like this:

```hsail
extension "foo:bar":0:1;
```

If a finalizer for a specific instruction set architecture does not support an extension that is enabled in a module, or does not support the specified version of the extension, then the finalizer must report an error. If a finalized code object is loaded that uses an extension that the HSA runtime does not support, or does not support the specified version of the extension, then the HSA runtime loader must report an error. In the case of an agent code object, the check is based on the specific kernel agent on which it is being loaded.

It is implementation defined if systems that support an extension with a given major and minor version also support the extension with the same major version and a smaller minor version.

HSA runtime queries can be used to get the list of supported extensions for the finalizer and loader, and to determine if a particular version of an extension is supported.

### 13.1.1 extension CORE

The "CORE" extension specifies that no extensions are allowed in the module in which it appears:

```hsail
extension "CORE";
```

If the "CORE" extension directive is present, the only other extension directives allowed in the same module are other "CORE" directives. The extension version must be omitted. 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 (on page 209) are allowed in the module in which it appears:

```hsail
extension "IMAGE";
```

Generally, the extension version can be omitted to use the image operations corresponding to the same HSAIL version as the module. However, it is also allowed to specify the extension version if a different version of the image operations is being used.

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
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 extensions

Any vendor extensions should add identifiers to the BRIG type definition that are prefixed by `hsa_ven_vendor_brig_` or `HSA_VEN_VENDOR_BRIG_` as appropriate, where `vendor` is the extension’s vendor name and `VENDOR` is the extension’s vendor name converted to uppercase. Any enumeration names added should have values that are greater or equal to the corresponding `HSA_BRING_*_FIRST_USER_DEFINED` value. See 13.1 extension directive (on page 293) and Chapter 18 BRIG: HSAIL Binary Format (on page 337).

For example, 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.

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:

```hsail
module &ext:1:1:$full:$large:$default;
extension "xyz:newext";
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:
    xyz:max3_f32 $s11, $s10, $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.

![Figure 13–2 location syntax diagram](image)

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 pad directive

The pad directive can be used to insert padding into the HSAIL binary representation of BRIG. This can be used by tools that directly edit BRIG to overwrite long directives and instructions with short ones, provided the tool sets the remaining words to be the pad directive.

![Figure 13–3 pad syntax diagram](image)

The pad directive can appear anywhere in the HSAIL code that an annotation can appear (see 4.3.1 Annotations (on page 59)).
The operand is treated as a u64 type, and if omitted the default value is 0. The value must be in the range 0 to 214-1. The value specifies the number of 4 byte words of zeros to leave in the BRIG representation in addition to the 4 bytes for the directive itself. See 18.5.1.12 hsa_brig_directive_none_t (on page 368).

**Example**

```hsa
global_u32 &i;
pad 5; // Leave (5+1)*4=24 bytes of padding in the BRIG

kernel 4pad_example(kernarg_u64 %float_buf)
{
    ld_kernarg_u64 $d1, [%float_buf];
    pad; // Leave (0+1)*4=4 bytes of padding in the BRIG
    // ...
}
```

### 13.4 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 13–4 pragma syntax diagram**

```
pragma

"pragma" -> TOKEN_STRING_LITERAL

pragmaOperand "" -> "",
```

**Figure 13–5 pragmaOperand syntax diagram**

```
pragmaOperand

TOKEN_STRING_LITERAL

immediateOperand

aggregateConstant

identifierOperand

TOKEN_LABEL_IDENTIFIER
```

The `pragma` directive can appear anywhere in the HSAIL code that an annotation can appear (see 4.3.1 Annotations (on page 59)). 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 literal. See 4.5 Strings (on page 80).
- 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 (on page 85).
- An aggregate constant. This is treated as an array of b8 the byte size of the aggregate constant. See 4.8.4 Aggregate constants (on page 99).
- An identifier. This includes the identifier of a variable, fbarrier, kernel, function, signature, module, register, and label. See 4.6 Identifiers (on page 81) and 4.7 Registers (on page 83).

The first operand of a pragma directive is the name of the pragma and must be a string literal. Names starting with hsa. are reserved for use by HSA Foundation specifications. Vendor specific pragma names should start with vendor. where vendor is the (possibly abbreviated) name of the vendor, which must consist of only lowercase letters and digits.

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 (on page 82).

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 "xyz.rti", "init", "symbolic", &i_p, &i, 4;

kernel &pragma_example(kernarg_u64 %float_buf)
{
   pragma "xyz.rti", "kernel", "arg", %float_buf, "%float";
   // ...
}
```

13.4.1 Loop pragmas

Pragmas with a name starting with hsa.loop provide information about loops in kernels and functions. They are experimental features (see 1.3 HSAIL experimental features (on page 23)).

The following definitions are used:
1. A basic block of a kernel or function is defined as the maximal sequence of one or more instructions that have no intervening labels or branch instructions, together with zero or more labels that immediately precede the first instruction of the sequence, and at most one branch or return instruction that immediately follows the last instruction of the sequence.

2. A control flow edge of a kernel or function denotes flow of control from a source basic block to a target basic block as a result of a branch instruction, or the result of a fall through edge where the execution of one basic block can fall through to the lexically flowing basic block.

3. A back edge is defined as a control flow transfer where the target dominates the source. See 2.12.3 (Post-)dominator and immediate (post-)dominator (on page 46).

4. The back edges of a label are defined as the back edges from branch instructions that explicitly specify the label, together with back edges that are fall through edges where the label is the first lexical label of the following basic block.

5. A loop label is a label with at least one back edge.

6. A natural loop formed from a back edge is defined as the basic blocks reachable from the back edge source basic block by only following reverse control flow edges, but not including any reverse control flow edges from the back edge target basic block.

7. The loop formed from a loop label is defined as the union of the basic blocks of the natural loops formed from the back edges of the loop label, together with the basic blocks of the natural loops formed from the back edges of any labels that are lexically between the loop label and the lexically following instruction.

8. A loop execution begins when control flow enters the loop label basic block from a basic block not in the loop, and ends when control flow leaves a basic block in the loop to a basic block not in the loop. The iteration number of a loop execution starts at 0 when the loop is entered, and increases by 1 each time a back edge of the loop is executed. Basic block execution A executes in a different loop iteration to basic block execution B if they are members of the same loop execution and have different loop iteration numbers. Basic block execution A executes in a later (earlier) loop iteration to basic block execution B if they execute in different loop iterations and the loop iteration number of A is greater (less) than the loop iteration number of B.

9. A loop carried dependence exists if a variable is accessed in two basic block executions, such that the basic blocks are executed in different loop iterations, and there is a dependency between the two accesses. The variable can either be a register or a memory location. The dependency can be a read-read dependency in which both accesses read the same value of the variable, a write-read dependency in which the later loop iteration access reads the value of the earlier loop iteration write access, a read-write dependency in which a later loop iteration access writes a value to a variable whose immediately previous value was read by an earlier loop iteration access, and a write-write dependency in which a later loop iteration access writes a value to a variable whose immediately previous value was written by the earlier loop iteration access.

10. An induction variable is a variable that is assigned values in the loop that are only a function of immediate values, values defined outside the loop and the current loop execution iteration number. For example, an induction variable may be assigned 0 outside the loop, and then assigned its current value plus 1 in a basic block of the loop that dominates each basic block that is the source of a back edge of the loop.
13.4.1.1 Parallel loop pragma

The parallel loop pragma is an optional hint that guarantees that the iterations of a loop are independent and that the number of iterations can be determined before any loop iterations are executed. This allows the iterations to be executed in any order, including interleaved or concurrently.

This information may be used by a finalizer to parallelize a loop. A finalizer that does not support the parallel loop pragma, or one which does not support parallelizing a specific loop marked with the parallel loop pragma, can ignore the parallel loop pragma and process the loop as a sequential loop.

For example, an HSAIL producer can use properties of a high level language, or perform memory dependence analysis, to identify parallel loops and encode the information in HSAIL using the parallel loop pragma. Note that although the finalizer is relieved from having to perform potentially costly memory dependence analysis, it must still ensure that it can parallelize a marked loop, including checking it supports the kind of instructions in the loop, the induction variable computations, and any iteration continuations or exits.

The parallel loop pragma syntax is:

```
pragma "hsa.loop.parallel";
```

If a parallel loop pragma is present it must obey all the following rules or the behavior is undefined:

1. The HSAIL lexical element immediately following a parallel loop pragma, after skipping any annotations, must be a label, and is termed the parallel loop label. See 4.9 Labels (on page 103).
2. A parallel loop label must be a loop label.
3. A parallel loop is defined as the loop formed from a parallel loop label. The control flow of a parallel loop must be reducible.
4. The only loop carried dependencies in a parallel loop must either be read-read dependencies on registers or memory variables, read-write or write-write dependencies on register variables, or write-read dependencies on registers that are parallel loop induction variables. There can be no other register or memory loop carried dependencies. As a consequence, parallel loop induction variables must be registers, and induction variable computations must not involve memory instructions except for reads of values defined outside the loop.
5. All parallel loop exit conditions must be computed from parallel loop induction variables, values computed by instructions not in the parallel loop, or immediate values.
6. No register value updated in a parallel loop must be used outside the parallel loop. However, register values updated outside the parallel loop may be used inside the parallel loop.
7. The instructions of a parallel loop must not contain any communication operations. See 2.12 Divergent control flow (on page 42).
The example is a three-dimensional loop with the following properties:

- `@loop_entry_par_outer`, `@loop_entry_seq` and `@loop_entry_par_inner` are loop labels as they are the target of back edges.
- The loops formed from `@loop_entry_par_outer` and `@loop_entry_par_inner` are parallel loops.
- The loop formed from `@loop_entry_seq` is contained in the loop formed from `@loop_entry_par_outer`. The iterations of the `@loop_entry_par_outer` loop can be executed in any order, but the contained `@loop_entry_seq` loop would need to be executed sequentially unless iteration independence can be proven by the finalizer or the kernel agent hardware.
- The loop formed from `@loop_entry_par_inner` is contained in the loop formed from `@loop_entry_seq`. Since it is marked as a parallel loop its iterations can be executed in any order and so may be able to be parallelized.
- The outer loop formed from `@loop_entry_par_outer` has two back edges, one with $c2$ as the predicate, another with $c3$ as the predicate.

### 13.4.2 Tools pragmas

Pragmas with a name starting with `hsa.tools` allow tools related information to be added to HSAIL.

#### 13.4.2.1 Profile event pragmas

Pragmas with a name starting with `hsa.tools.profile.event` allow profile events to be created within HSAIL kernels and functions. See *HSA Runtime Programmer’s Reference Manual Version 1.2*, section 3.5 *Profile events* for more information, including how to register, enable, disable, and access generated profile events. They are experimental features (see 1.3 HSAIL experimental features (on page 23)).

**NOTE:** Pragmas starting with `hsa.tools.profiling` are deprecated and have been replaced by corresponding pragmas starting with `hsa.tools.profile`.

#### 13.4.2.1.1 Profile event trigger pragma

The profile event trigger pragma can be placed in a kernel or function anywhere an instruction is allowed. It specifies the identity of a profile event, together with values for the metadata fields defined by the profile event.
A work-item executes an event trigger pragma by first determining if the event specified by the pragma is enabled. If it is not enabled then no action is taken. If it is enabled, a profile event for the specified event is generated and its metadata fields are set to the corresponding values specified by the pragma.

The syntax is:

```c
pragma "hsa.tools.profile.event.trigger", eventID {, value};
```

**eventID**

The identifier for the profile event to trigger. Must be an integer literal or a `d` register. The value of the identifier must match the value passed to the HSA runtime operation that registers the profile event.

**value**

The position of the value in the list identifies to which metadata field of the event it corresponds. The value can be an integer constant (which is used as a `u64` type), a floating-point constant, an integer or floating-point typed constant, a register, a string, or an HSA metadata placeholder.

If the value is a string, it is undefined behavior if the code object that creates a profile event that specifies the string is destroyed before the profile event is accessed.

HSA metadata placeholders are special global identifiers which will be evaluated to the relevant value when the event is triggered. The supported standard placeholders are:

- `&__ hasa.tools.profile.workitemflatabsid` — The work-item flattened absolute ID. Equivalent to using the return value of the `workitemflatabsid` instruction as a value.
- `&__ hasa.tools.profile.queueid` — The ID of the user mode queue which the packet was enqueued on.
- `&__ hasa.tools.profile.packetid` — User mode queue packet ID, unique to the user mode queue used for the dispatch. Equivalent to using the return value of the `packetid` instruction as a value.

It is undefined behavior if the type of the value does not match the corresponding type of the metadata field specified when the profile event is registered.

### 13.4.2.2 Debug pragmas

Pragmas with a name starting with `hsa.tools.debug` allow high-level language source code debugging information to be added to HSAIL. This allows the information to be transported through HSAIL and used by the finalizer to generate debug information for the generated machine code that correlates to the original high-level language source code. They are experimental features (see 1.3 HSAIL experimental features (on page 23)).

For example, these pragmas are designed such that a finalizer can generate the *DWARF Debugging Information Format Version 4* referencing the original OpenCL source code.

There are four kinds of debug pragmas: `hsa.tools.debug` pragmas are general, such as allowing lists of other entities; `hsa.tools.debug.src` pragmas describe entities in the high-level source code, such as compile units, types, and variables; `hsa.tools.debug.ref` pragmas provide the means to associate these high-level concepts with the HSAIL declarations and instructions; and `hsa.tools.debug.expr` pragmas provide a means to compute values and locations of high-level variables.
13.4.2.2.1 Version

The version of the debug pragmas present in the module. Must precede any other debug pragma. If omitted defaults to the HSAIL version number specified by the module header (see Chapter 14 module Header (on page 323)).

The syntax is:

```
pragma "hsa.tools.debug.version", major, minor;
```

*major*: integerConstant

The major version. Major number changes are incompatible, so debug information with one major number cannot used by a consumer that only supports a different major number. The major version of the debug pragmas defined by this specification is 1.

*minor*: integerConstant

The minor version. Minor number changes correspond to added functionality. Minor changes are compatible, so debug information with one minor level can be used by a consumer that supports a larger minor level. The minor version of the debug pragmas defined by this specification is 2.

13.4.2.2.2 Tool

The tool that generated the HSAIL.

The syntax is:

```
pragma "hsa.tools.debug.tool", tool;
```

*tool*: TOKEN_STRING_LITERAL

Information about the tool used to generate the HSAIL.

13.4.2.2.3 Source description

Source level debug information entity description. Use this pragma category to describe a single source level debug information entity.

The overall syntax for this pragma group is:

```
pragma "hsa.tools.debug.src", entityId, "srcTag" {, srcTagOperand};
```

*entityId*: integerConstant

Entity identification numbers start from one and must be unique amongst all `hsa.tools.debug.src` and `hsa.tools.debug.expr` pragmas within an HSAIL module; 0 is reserved for optional references.

13.4.2.2.3.1 List

List of debug pragma entity identification numbers.

The syntax is:

```
pragma "hsa.tools.debug.src", entityId, "List" {, listOperand};
```

*entityId*: integerConstant

List identification number.

*listOperand*: integerConstant

Entity identification number of a source description pragma.
Chapter 13. Directives  13.4 pragma directive

13.4.2.2.3.2 CompileUnit

Compilation unit description.

The syntax is:

```
pragma "hsa.tools.debug.src", entityId, "CompileUnit", scopeId, langString, producerString,
flagString, enumIdList, typeIdList, subprogramIdList, variableIdList, importIdList;
```

**entityId**: integerConstant  
Compilation unit identification number.

**scopeId**: integerConstant  
Identification number of the scope pragma that encapsulates the compilation unit.

**langString**: TOKEN_STRING_LITERAL  

**producerString**: TOKEN_STRING_LITERAL  
String that holds name of the producer.

**flagString**: TOKEN_STRING_LITERAL  
String that holds compiler flags used for the compilation.

**enumIdList**: integerConstant  
Entity identification number of the list pragma that provides the list of identification numbers of enumerator pragma descriptions that are used in this compilation unit.

**typeIdList**: integerConstant  
Entity identification number of the list pragma that provides the list of identification numbers of type pragma descriptions that are used in this compilation unit.

**subprogramIdList**: integerConstant  
Entity identification number of the list pragma that provides the list of identification numbers of subprograms pragma descriptions (declarations and definitions) that are used in this compilation unit.

**variableIdList**: integerConstant  
Entity identification number of the list pragma that provides the list of identification numbers of global variable pragma descriptions that are used in this compilation unit.

**importIdList**: integerConstant  
Entity identification number of the list pragma that provides the list of identification numbers of imported entity pragma descriptions that are used in this compilation unit.

13.4.2.2.3.3 File

File description.

The syntax is:

```
pragma "hsa.tools.debug.src", entityId, "File", nameString, folderString, text, macroList;
```

**entityId**: integerConstant  
File identification number.

**nameString**: TOKEN_STRING_LITERAL  
File name.
folderString: TOKEN_STRING_LITERAL
Path where the file is located.

text: integerConstant
Entity identification number of the text pragma that provides the textual contents of the file, or 0 if the textual contents must be obtained by another means such as reading an external file using the file and folder names.

NOTE: This can be used for languages that support on line compilation and do not use external files.

macroList: integerConstant
Entity identification number of the list pragma that provides the list of identification numbers of macro pragma definitions that are found in this file.

13.4.2.2.3.4 Text
Source text.
The syntax is:
pragma "hsa.tools.debug.src", entityId, "Text", sourceText;

entityId: integerConstant
Text identification number.

sourceText: TOKEN_STRING_LITERAL
Text string.

13.4.2.2.3.5 Location
Source line location description.
The syntax is:
pragma "hsa.tools.debug.src", entityId, "Location", fileId, lineNum, columnNumber, discriminator, inlinedAtId;

entityId: integerConstant
Location identification number.

fileId: integerConstant
Identification number of the file that encapsulates the source line location description.

lineNum: integerConstant
Line number.

columnNumber: integerConstant
Column number.

discriminator: integerConstant
Discriminator number that identifies which path the instructions belongs to if there are more than one trace path that share the same source line. A 0 value signifies that the source line is not shared.

inlinedAtId: integerConstant
Optional source line location description identification number if the described source line has been inlined by the compiler.

13.4.2.2.3.6 Subprogram
Subprogram description.
The syntax is:

```
pragma "hsa.tools.debug.src", entityId, "Subprogram", scopeId, locationId, typeId, nameString,
flagString, variableIdList;
```

- **entityId**: integerConstant
  - Subprogram identification number.
- **scopeId**: integerConstant
  - Identification number of the scope pragma that encapsulates the subprogram.
- **locationId**: integerConstant
  - Identification number of the source line location description for the beginning of the subprogram.
- **typeId**: integerConstant
  - Identification number of the subprogram prototype description.
- **nameString**: TOKEN_STRING_LITERAL
  - Name string.
- **flagString**: TOKEN_STRING_LITERAL
  - String that holds compiler flags used for the subprogram.
- **variableIdList**: integerConstant
  - Entity identification number of the list pragma that provides the list of identification numbers of argument and local variable pragma descriptions that are defined in this subprogram.

### 13.4.2.2.3.7 LexicalBlock

Lexical block description.

The syntax is:

```
pragma "hsa.tools.debug.src", entityId, "LexicalBlock", locationId;
```

- **entityId**: integerConstant
  - Block identification number.
- **locationId**: integerConstant
  - Identification number of the source line location description for the beginning of the block.

### 13.4.2.2.3.8 Namespace

Namespace description.

The syntax is:

```
pragma "hsa.tools.debug.src", entityId, "Namespace", scopeId, locationId, nameString;
```

- **entityId**: integerConstant
  - Namespace identification number.
- **scopeId**: integerConstant
  - Identification number of the scope pragma that encapsulates the namespace.
- **locationId**: integerConstant
  - Identification number of the source line location description for the beginning of the block.
- **nameString**: TOKEN_STRING_LITERAL
  - Name string.
13.4.2.2.3.9 GlobalVariable

Global variable description.

The syntax is:

```
pragma "hsa.tools.debug.src", entityId, "GlobalVariable", scopeId, locationId, typeId, nameString, linkNameString, flagString;
```

**entityId**: integerConstant
Global variable identification number.

**scopeId**: integerConstant
Identification number of the scope pragma that encapsulates the variable.

**locationId**: integerConstant
Identification number of the source line location description for the beginning of the variable.

**typeId**: integerConstant
Identification number of the variable type description.

**nameString**: TOKEN_STRING_LITERAL
Name string.

**linkNameString**: TOKEN_STRING_LITERAL
Name used for linking.

**flagString**: TOKEN_STRING_LITERAL
String that holds flags that describe a variable in more detail (example: isDefinition, isOptimised, ...).

**isLocalToUnit**: integerConstant
The value is 1 if the global variable is local to its compilation unit and 0 otherwise.

**isDefinition**: integerConstant
The value is 1 if the global variable is local to its compilation unit and 0 otherwise.

13.4.2.2.3.10 LocalVariable

Local variable description.

The syntax is:

```
pragma "hsa.tools.debug.src", entityId, "LocalVariable", scopeId, locationId, typeId, nameString, argNum;
```

**entityId**: integerConstant
Local variable identification number.

**scopeId**: integerConstant
Identification number of the scope pragma that encapsulates the variable.

**locationId**: integerConstant
Identification number of the source line location description for the beginning of the variable.

**typeId**: integerConstant
Identification number of the variable type description.

**nameString**: TOKEN_STRING_LITERAL
Name string.
argNum: integerConstant
    Holds an argument order number if the variable is an argument of a subroutine. A 0 value signifies that a variable is just a local variable.

13.4.2.2.3.11 Subrange

    Subrange description.
The syntax is:
    
    pragma "hsa.tools.debug.src", entityId, "Subrange", scopeId, countNum, lowerBoundNum;

entityId: integerConstant
    Subrange identification number.

scopeId: integerConstant
    Identification number of the scope pragma that encapsulates the subrange.

countNum: integerConstant
    Number of elements.

lowerBoundNum: integerConstant
    Starting element.

13.4.2.2.3.12 Enumerator

    Enumerator description.
The syntax is:
    
    pragma "hsa.tools.debug.src", entityId, "Enumerator", scopeId, nameString, valueString;

entityId: integerConstant
    Enumerator identification number.

scopeId: integerConstant
    Identification number of the scope pragma that encapsulates the enumerator.

nameString: TOKEN_STRING_LITERAL
    Name string.

valueString: TOKEN_STRING_LITERAL
    Value string.

13.4.2.2.3.13 BasicType

    Basic type description.
The syntax is:
    
    pragma "hsa.tools.debug.src", entityId, "BasicType", scopeId, size, alignment, nameString, encodingString;

entityId: integerConstant
    Basic type identification number.

scopeId: integerConstant
    Identification number of the scope pragma that encapsulates the type.

size: integerConstant
    Size.
alignment: integerConstant
Alignment.

nameString: TOKEN_STRING_LITERAL
Name string.

encodingString: TOKEN_STRING_LITERAL

13.4.2.2.3.14 DerivedType

Derived type description.

The syntax is:

```
pragma "hsa.tools.debug.src", entityId, "DerivedType", scopeId, size, alignment, derivedFromId, tagString, segment;
```

entityId: integerConstant
Derived type identification number.

scopeId: integerConstant
Identification number of the scope description that encapsulates the type.

size: integerConstant
Size.

alignment: integerConstant
Alignment.

derivedFromId: integerConstant
Identification number of the type this type derived from.

tagString: TOKEN_STRING_LITERAL
String that describes tag of the type. Expected values are "Volatile", "Typedef", "Restrict", "Pointer", "Reference", "Member", "Interface", "RValue", "Const", "Shared", "Unit", "Thrown", "TemplateType", "TemplateValue", "Set", "Packet", and "PtrToMember".

segment: integerConstant
Segment which the pointer or reference type points to. It is encoded using the values of hsa Brig Segment_t (see 18.3.27 hsa Brig Segment_t (on page 352)). If this is not a pointer or reference type, the value should be 0.

13.4.2.2.3.15 IndexableType

Indexable type description.

The syntax is:

```
pragma "hsa.tools.debug.src", entityId, "IndexableType", scopeId, size, alignment, locationId, baseId, subrangeIdList, tagString;
```

entityId: integerConstant
Indexable type identification number.

scopeId: integerConstant
Identification number of the scope pragma that encapsulates the type.
size: integerConstant
   Size.

alignment: integerConstant
   Alignment.

locationId: integerConstant
   Identification number of the source line location description that holds the type definition.

dependId: integerConstant
   Identification number of the type that is the base element of the type.

subrangeIdList: integerConstant
   Entity identification number of the list pragma that provides the list of identification numbers of subrange
   pragma descriptions, one per every type dimension.

tagString: TOKEN_STRING_LITERAL
   String that describes tag of the type. Expected values are: "Array" and "Vector".

13.4.2.2.3.16 CompositeType

   Composite type description.
   The syntax is:
   pragma "hsa.tools.debug.src", entityId, "CompositeType", scopeId, size, alignment, locationId,
   elementIdList, tagString;

entityId: integerConstant
   Composite type identification number.

scopeId: integerConstant
   Identification number of the scope pragma that encapsulates the type.

size: integerConstant
   Size.

alignment: integerConstant
   Alignment.

locationId: integerConstant
   Identification number of the source line location description that holds the type definition.

elementIdList: integerConstant
   Entity identification number of the list pragma that provides the list of identification numbers of element pragma
descriptions that are encapsulated by the type.

tagString: TOKEN_STRING_LITERAL
   String that describes tag of the type. Expected values are: "Structure", "Union", and "Class".

13.4.2.3.17 EnumerationType

   Enumeration type description.
   The syntax is:
   pragma "hsa.tools.debug.src", entityId, "EnumerationType", scopeId, nameString, locationId,
   enumeratorIdList;

entityId: integerConstant
   Enumeration type identification number.
scopeId: integerConstant
Identification number of the scope pragma that encapsulates the type.

nameString: TOKEN_STRING_LITERAL
Name string.

locationId: integerConstant
Identification number of the source line location description that holds the type definition.

enumeratorIdList: integerConstant
Entity identification number of the list pragma that provides the list of identification numbers of enumerator pragma descriptions that are encapsulated by the type.

13.4.2.2.3.18 SubprogramType

Subprogram type description.
The syntax is:
pragma "hsa.tools.debug.src", entityId, "SubprogramType", scopeId, returnTypeId, argTypeIdList;

digitId: integerConstant
Subprogram type identification number.

scopeId: integerConstant
Identification number of the scope pragma that encapsulates the type.

returnTypeId: integerConstant
Identification number of the type returned by the subprogram.

argTypeIdList: integerConstant
Entity identification number of the list pragma that provides the list of identification numbers for the subprogram argument type pragma descriptions.

13.4.2.2.3.19 MacroDefine

Macro define.
The syntax is:
pragma "hsa.tools.debug.src", entityId, "MacroDefine", lineNum, nameString, valueString;

digitId: integerConstant
Macro identification number.

lineNum: integerConstant
Line number where macro is defined.

nameString: TOKEN_STRING_LITERAL
Macro name string. The value of this string will be the name of the macro symbol that was defined at the indicated line number. In the case of a function-like macro it is followed immediately by the macro formal parameter list including the surrounding parentheses.

valueString: TOKEN_STRING_LITERAL
Macro definition string.

13.4.2.2.3.20 MacroUndefine

Macro undefined.
Chapter 13. Directives  13.4 pragma directive

The syntax is:

```
pragma "hsa.tools.debug.src", entityId, "MacroUndefine", lineNum, nameString;
```

**entityId**: integerConstant
Macro identification number.

**lineNum**: integerConstant
Line number where macro is undefined.

**nameString**: TOKEN_STRING_LITERAL
Macro name string.

13.4.2.2.3.21 **MacroInclude**

Macro include file.

The syntax is:

```
pragma "hsa.tools.debug.src", entityId, "MacroInclude", lineNum, fileId;
```

**entityId**: integerConstant
Macro identification number.

**lineNum**: integerConstant
Line number where file was included.

**fileId**: integerConstant
Identification number of the file that was included.

13.4.2.2.4 **Source reference**

HSAIL to source level reference description.

Use this pragma category to describe a reference point between HSAIL and the source code debug information description.

The overall syntax for this pragma group is:

```
pragma "hsa.tools.debug.ref", "refTag" {, refTagOperand};
```

13.4.2.2.4.1 **SourceLine**

Source line debug information reference.

Describes the HSAIL instructions to source level line mapping where all HSAIL instructions following the pragma are considered to represent that source line up to the next source line pragma.

The syntax is:

```
pragma "hsa.tools.debug.ref", "SourceLine", locationId;
```

**locationId**: integerConstant
Entity identification number of an hsa.tools.debug.src source line location pragma. If the value is 0 then this pragma is a null source line pragma and is used only to terminate the previous source line pragma.
13.4.2.2.4.2 GlobalAddr

Address for, possibly a piece of, a source level global variable. It applies to the whole HSAIL module, except for the parts for which a local source reference pragma applies. Each piece of a source level variable must be associated with at most one global source reference pragma.

The pragma must be at module scope.

The syntax is:

```
pragma "hsa.tools.debug.ref", "GlobalAddr", variableId, hsailIdentifier, expressionId;
```

variableId: integerConstant
Entity identification number of an hsa.tools.debug.src global variable pragma.

hsailIdentifier: identifierOperand
HSAIL identifier of a visible module variable.

expressionId: integerConstant
Entity identification number of an hsa.tools.debug.expr expression pragma. The expression stack is initialized with the segment address of the HSAIL module variable. The evaluation of the expression must leave the expression stack with a segment address that can be used to access the source level variable. The expression may end with a fragment operation to indicate which piece of the source level variable it holds. If 0, the HSAIL module variable holds the complete source level variable.

13.4.2.2.4.3 GlobalImplicit

Implicit value for, possibly a piece of, a source level global variable. It applies to the whole HSAIL module, except for the parts for which a local source reference pragma applies. Each piece of a source level variable must be associated with at most one global source reference pragma.

The pragma must be at module scope.

Used to express that there is no HSAIL entity associated with the source level global variable, but the value of the source level variable is known.

The syntax is:

```
pragma "hsa.tools.debug.ref", "GlobalImplicit", variableId, expressionId;
```

variableId: integerConstant
Entity identification number of an hsa.tools.debug.src global variable pragma.

expressionId: integerConstant
Entity identification number of an hsa.tools.debug.expr expression pragma. The expression stack is initialized empty. The evaluation of the expression must leave the expression stack with the value of the source level variable. The expression may end with a fragment operation to indicate which piece of the source level variable it holds. Must not be 0.

13.4.2.2.4.4 LocalAddr

Address for, possibly a piece of, a source level variable at a particular point of execution of a kernel or function. It applies from the point of execution at which it is placed, up to the point of execution of the next source reference pragma that specifies the same piece of the source level variable.

The pragma must be at function scope.
It can be placed after an HSAIL definition or statement that starts the lifetime of an HSAIL variable or formal argument associated with a source level variable.

Allows the HSAIL entity that is associated with a source level variable to change during the execution of a kernel or function.

The syntax is:

```
pragma "hsa.tools.debug.ref", "LocalAddr", variableId, hsaIdentifier, expressionId;
```

**variableId**: integerConstant
Entity identification number of an `hsa.tools.debug.src` variable pragma.

**hsaIdentifier**: identifierOperand
HSAIL identifier of a visible variable or formal argument.

**expressionId**: integerConstant
Entity identification number of an `hsa.tools.debug.expr` expression pragma. The expression stack is initialized with the segment address of the HSAIL variable or formal argument. The evaluation of the expression must leave the expression stack with a segment address that can be used to access the source level variable. The expression may end with a fragment operation to indicate which piece of the source level variable it holds. If 0, the HSAIL variable or formal argument holds the complete source level variable.

### 13.4.2.2.4.5 LocalImplicit

Implicit value for, possibly a piece of, a source level variable at a particular point of execution of a kernel or function. It applies from the point of execution at which it is placed, up to the point of execution of the next local source reference pragma that specifies the same piece of the source level variable.

The pragma must be at function scope.

It can be placed where there is no HSAIL entity associated with the source level variable, and where the value of the source level variable is either known or not available. For example, it can be placed before the HSAIL statement that ends the lifetime of an HSAIL entity associated with a source level variable, which then becomes unavailable.

Allows the HSAIL entity that is associated with a source level variable to change during the execution of a kernel or function.

The syntax is:

```
pragma "hsa.tools.debug.ref", "LocalImplicit", variableId, expressionId;
```

**variableId**: integerConstant
Entity identification number of an `hsa.tools.debug.src` variable pragma.

**expressionId**: integerConstant
Entity identification number of an `hsa.tools.debug.expr` expression pragma. The expression stack is initialized empty. The evaluation of the expression must leave the expression stack either with the value of the source level variable, or empty if the source level variable is unavailable. The expression may end with a fragment operation to indicate which piece of the source level variable it holds. If 0, then the complete source level variable is unavailable.
13.4.2.2.4.6 LocalRegister

HSAIL register that holds, possibly a piece of, a source level variable at a particular point of execution of a kernel or function. It applies from the point of execution at which it is placed, up to the point of execution of the next local source reference pragma that specifies the same piece of the source level variable.

The pragma must be at function scope.

It can be placed after an HSAIL definition or statement that starts the lifetime of an HSAIL register holding a source level variable.

Allows the HSAIL entity that is associated with a source level variable to change during the execution of a kernel or function.

The syntax is:

```
pragma "hsa.tools.debug.ref", "LocalRegister", variableId, hsailIdentifier, expressionId;
```

**variableId**: integerConstant

Entity identification number of an hsa.tools.debug.src variable pragma.

**hsailIdentifier**: identifierOperand

HSAIL identifier of a register.

**expressionId**: integerConstant

Entity identification number of an hsa.tools.debug.expr expression pragma. The expression stack is initialized with the HSAIL register. The expression must contain no calculation operations. The expression may end with a fragment operation to indicate which piece of the source level variable it holds. If 0, then the HSAIL register holds the complete source level variable.

13.4.2.2.4.7 Subprogram

Source level subprogram definition reference.

The pragma must be at module scope. Each HSAIL kernel or function must be specified by at most one debug pragma.

The syntax is:

```
pragma "hsa.tools.debug.ref", "Subprogram", subprogramId, hsailIdentifier;
```

**subprogramId**: integerConstant

Entity identification number of an hsa.tools.debug.src subprogram pragma.

**hsailIdentifier**: identifierOperand

HSAIL kernel or function identifier that implements a source level subprogram definition.

13.4.2.2.5 Expression

An expression is used for computing a value or naming the location of a source level variable.

This pragma category uses a stack machine model where the stack is initialized as defined by the associated source reference pragma, which also defines the required final state of the stack.

If chained together, each expression pragma will find the stack in a state left by the previous expression pragma.

The overall syntax for this pragma group is:

```
pragma "hsa.tools.debug.expr", entityId, nextExprId, "exprTag" {, exprTagOperand};
```
**entityId**: integerConstant
Entity identification numbers start from one and must be unique amongst all `hsa.tools.debug.src` and `hsa.tools.debug.expr` pragmas within an HSAIL module; 0 is reserved for optional references.

**nextExprId**: integerConstant
Entity identification number of the next expression pragma in the chain and is used for modeling complex expressions, or 0 if there are no further expressions in the chain.

### 13.4.2.2.5.1 Dereference

Pops the top element of the stack which must be a segment address, loads a segment address from the popped address, and finally pushes the loaded segment address on top of the stack.

The syntax is:

```hsa.tools.debug.expr", entityId, nextExprId, "Dereference", segment;```

**entityId**: integerConstant
Expression identification number.

**nextExprId**: integerConstant
Identification number of the next expression pragma in the chain.

**segment**: TOKEN_STRING_LITERAL
Segment to use for the segment address that is loaded. It defines the number of bytes that will be loaded, and the segment of the address that will be pushed on the stack. It does not specify the segment that the load will access, as that is defined by the popped segment address itself. It can have the value "flat", "global", "group", "private", "readonly", and "kernarg".

### 13.4.2.2.5.2 Constu

Pushes an unsigned 64-bit value on top of the stack.

The syntax is:

```hsa.tools.debug.expr", entityId, nextExprId, "Constu", value;```

**entityId**: integerConstant
Expression identification number.

**nextExprId**: integerConstant
Identification number of the next expression pragma in the chain.

**value**: integerConstant
Value, treated as a u64, to push on the stack.

### 13.4.2.2.5.3 PlusUconst

Pops the top element of the stack which must be a segment address, adds the value to it, and pushes the result of the operation on top of the stack as a segment address.

The syntax is:

```hsa.tools.debug.expr", entityId, nextExprId, "PlusUconst", value;```

**entityId**: integerConstant
Expression identification number.

**nextExprId**: integerConstant
Identification number of the next expression pragma in the chain.
value: integerConstant
Value, treated as a u64, that needs to be added to the top element of the stack.

13.4.2.2.5.4 Fragment

Specifies the portion of the source level variable with which the expression is associated.
Must be the last element of an expression chain.

The syntax is:
pragma "hsa.tools.debug.expr", entityId, nextExprId, "Fragment", bitOffset, bitSize;

entityId: integerConstant
Expression identification number.

nextExprId: integerConstant
Identification number of the next expression pragma in the chain.

bitOffset: integerConstant
Offset in bits from the beginning of the source level variable.

bitSize: integerConstant
Size in bits of the portion of the source level variable. Also specifies the number of least significant bits of the entity being associated with the source level variable.

13.5 Control directives for low-level performance tuning

HSAIL provides control directives to allow implementations to pass information to the finalizer. These directives are used for low-level performance tuning. See Table 13–1 (below).

Table 13–1 Control directives for low-level performance tuning

<table>
<thead>
<tr>
<th>Directive</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>enablebreakexceptions</td>
<td>exceptionsNumber</td>
</tr>
<tr>
<td>enabledetectexceptions</td>
<td>exceptionsNumber</td>
</tr>
<tr>
<td>maxdynamicgroupsize</td>
<td>size</td>
</tr>
<tr>
<td>maxflatgridsize</td>
<td>count</td>
</tr>
<tr>
<td>maxflatworkgroupsize</td>
<td>count</td>
</tr>
<tr>
<td>requireddim</td>
<td>nd</td>
</tr>
<tr>
<td>requiredgridsize</td>
<td>nx, ny, nz</td>
</tr>
<tr>
<td>requiredgroupbaseptralign</td>
<td>align</td>
</tr>
<tr>
<td>requiredworkgroupsize</td>
<td>nx, ny, nz</td>
</tr>
<tr>
<td>requirenopartialwavefronts</td>
<td></td>
</tr>
<tr>
<td>requirenopartialworkgroups</td>
<td></td>
</tr>
</tbody>
</table>

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 (on page 328)).

size: The number of bytes. Must be an integer literal constant value of data type u32. WAVESIZE is not allowed.

count: The number of work-items. Must be an integer literal constant value, greater than 0, of data type u32 for maxflatworkgroupsize and u64 for maxflatgridsize. WAVESIZE is allowed.
### Explanation of Arguments

- **align**: The byte alignment of the base of the memory. Must be an integer literal constant with the value 1, 2, 4, 8, 16, 32, 64, 128, and 256 of data type u32. WAVESIZE is not allowed.

- **nd**: The number of dimensions. Must be an integer literal 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 integer literal constant value, greater than 0, of data type u32 for `requiredworkgroupsize` and u64 for `requiredgridsize`. WAVESIZE is allowed.

See also 18.3.8 hsa Brig Control Directive (on page 343).

The control directives must appear in the code block of a kernel or function and only apply to that kernel or function. This allows an HAIL finalizer and linker to process kernels and functions independently, since control directives in one kernel or function can not alter another.

Control directives must appear before the first HSAIL code block definition or statement (see 4.3.5 Code block (on page 65)). 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 (on page 290). `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 it is undefined whether 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 (on page 328)). 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 (on page 121).
enabledetectexceptions

Specifies the set of exceptions that must be enabled for the DETECT policy. See 12.3 Hardware exception policies (on page 290). 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 it is undefined whether 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 (on page 280)), 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 (on page 328)). 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 (on page 121).

maxdynamicgroupsize

Specifies the maximum number of bytes of dynamic group memory (see 4.20 Dynamic group segment memory allocation (on page 123)) 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 machine 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 machine 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 machine 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 maxflatworkgroupsize 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 program execution is 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 machine 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 \texttt{requireddim} is specified (either by a control directive or when the finalizer was invoked), it must be consistent with \texttt{requiredgridsize} and \texttt{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 \texttt{requireddim} control directive.

\texttt{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 \texttt{u64}, with a value greater than 0 (\texttt{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 program execution is undefined if the kernel is dispatched with a grid size that does not match these values. A finalizer might be able to generate better machine code for the \texttt{gridsize} instruction. Also, if the total grid size is less than \( 2^{24} \cdot 1 \), then faster \texttt{mul24} instructions might be able to be used for the \texttt{workitemid, workitemabsid, workitemflatid, and workitemflatabsid} instructions, because the terms for dimensions above the value specified can be treated as 1. In conjunction with \texttt{requiredworkgroupsize}, a finalizer might also be able to generate better machine code for \texttt{gridgroups} and \texttt{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 \texttt{requiredgridsize} is specified (either by a control directive or when the finalizer was invoked), it must be consistent with \texttt{requiredworkgroupsize} and \texttt{requireddim} if specified: invalid dimensions must be 1, and valid dimension must not be 0.

If the values for required grid size are specified when the finalizer is invoked, they must match the corresponding values given in any \texttt{requiredgridsize} control directive. The product of the values must also be less than or equal to the value specified by \texttt{maxflatgridsize}.

\texttt{requiredgroupbaseptralign}

Specifies the byte alignment of the group segment address returned by the \texttt{groupbaseptr} instruction.

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 required alignment is specified when the finalizer is invoked, the value must match the value in any \texttt{requiredgroupbaseptralign} control directive.

See \texttt{4.20 Dynamic group segment memory allocation (on page 123)}.

\texttt{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 \texttt{u32}, with a value greater than 0 (\texttt{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 program execution is undefined if the kernel is dispatched with a work-group size that does not match these values.
A finalizer might be able to generate better machine 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 machine 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 requiredworkgroups is specified (either by a control directive or when the finalizer was invoked), it must be consistent with requiredgridsize and requireddim if specified: invalid dimensions must be 1, and valid dimension must not be 0.

If the values for required work-group size are specified when the finalizer is invoked, they must match the corresponding values given in any requiredworkgroups control directive. The product of the values must also be less than or equal to the value specified by maxflatworkgroupsizesize.

requirenopartialwavefronts

Specifies that the kernel must be dispatched with no partial wavefronts. 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 program execution is undefined if the kernel is dispatched with a work-group and grid size that results in any work-group not being an exact multiple of the wavefront size. The dispatch may have partial workgroups, but only if they are all an exact multiple of the wavefront size.

A finalizer might be able to generate better machine code for cross lane instructions that specify width (all) if it knows there are no partial wavefronts, because no test is required to determine if there are inactive lanes. A kernel agent might be able to dispatch a kernel more efficiently if it knows there are no partial wavefronts.

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 wavefronts is specified when the finalizer is invoked, the kernel behaves as if the requirenopartialwavefronts control directive has been specified.

requirenopartialworkgroups

Specifies that the kernel must be dispatched with no partial work-groups. It can be placed in either a kernel or a function code block. This is only a hint and can be ignored by the finalizer.

The program execution is undefined if the kernel is dispatched with any dimension of the grid size not being an exact multiple of the corresponding dimension of the work-group size.

A finalizer might be able to generate better machine code for currentworkgroups if it knows there are no partial work-groups, because the result becomes the same as the workgroupsizesize 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 requirenopartialworkgroupscontrol 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 (on page 59)).

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 (on page 81).

**major**

An integer literal of type u64 and must be in the right-open interval [0, 2^{32}).

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 literal of type u64 and must be in the right-open interval [0, 2^{32}).

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:

- $\text{base}$ — The base profile should be used. Inclusion of this option indicates that the associated HSAIL uses or requires features of the base profile.
- $\text{full}$ — 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 (on page 327).

machine_model

Specifies which machine model is used during finalization. Possibilities are:

- $\text{large}$ — Specifies large model, in which all flat and global addresses are 64 bits.
- $\text{small}$ — 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 (on page 40).

default_float_rounding

Specifies which default floating-point rounding mode is used during finalization. Possibilities are:

- $\text{default}$ — 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 machine code, which can be either $\text{zero}$ or $\text{near}$. 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.
- $\text{zero}$ — Specifies that $\text{zero}$ floating-point rounding must be used. An error must be reported by the finalizer if the module header specifies the $\text{base}$ profile and the kernel agent does not support $\text{zero}$ floating-point rounding mode for the base profile.
- $\text{near}$ — Specifies that $\text{near}$ floating-point rounding must be used. An error must be reported by the finalizer if the module header specifies the $\text{base}$ profile and the kernel agent does not support $\text{near}$ floating-point rounding mode for the base profile.

For more information, see 4.19.2 Floating-point rounding (on page 118) and 16.2.1 Base profile requirements (on page 328).

It is an error to add an HSAIL module to an HAIL program unless the rules defined in 4.2.1 Finalization (on page 51) are met.

See 4.2 Program, code object, and executable (on page 50).

Examples

```
module &m1:1:full:$small:$default;
module &m2:1:full:$large:$zero;
module &m3:1:base:$small:$near;
module &m4:1: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 (on page 50) and 4.12 Linkage (on page 106).

15.2 Library Example

An example of library code is shown below:

```hsail
module &lib:1:1:$full:$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

dcl 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 (on page 50).
- The required profile must be selected by a modifier on the module header. See 14.1 Syntax of the module header (on page 323).
- 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 (on page 118).
- For all floating-point arithmetic instructions (see 5.11 Floating-point arithmetic instructions (on page 151)); cmp with floating-point sources (see 5.18 Compare (cmp) instruction (on page 166)); and cvt with a floating-point source type (see 5.19 Conversion (cvt) instruction (on page 171)):
  - 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 (on page 157)) or native floating-point instructions (see 5.14 Native floating-point instructions (on page 159)).

- 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 (on page 420) 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 `fract` instructions.
  - Does not support the 64-bit floating-point type (`f64`), 64-bit packed floating-point type (`2x64`), double-precision floating point constants, nor any instructions on the types.
  - Must provide `div` instructions less than or equal to 2.5 ULP (see 4.19.6 Unit of least precision (ULP) (on page 121)) 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 (on page 118)).
  - 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 (on page 175)) 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 \texttt{ftz} modifier (when \texttt{ftz} is valid).

• For all floating-point arithmetic instructions (see 5.11 Floating-point arithmetic instructions (on page 151)) and \texttt{cvt} with a floating-point source and destination type (see 5.19 Conversion (\texttt{cvt}) instruction (on page 171)), 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) (on page 120)).
  
  The exception to this rule is \texttt{min} and \texttt{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 (on page 323)). The default floating-point rounding modes supported must be either \texttt{zero}, \texttt{near}, or both \texttt{zero} and \texttt{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 \texttt{icall} instruction is not supported. See 10.8 Indirect call (\texttt{icall}) instruction (on page 271).

• It is optional if the DETECT or BREAK exception policies (see 12.3 Hardware exception policies (on page 290)) for the five exceptions specified in 12.2 Hardware exceptions (on page 288) 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 (on page 121).

• An implementation is only required to support \texttt{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 (on page 184)). In particular, it is not required that memory allocated by a system memory allocator support \texttt{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 \texttt{add}, \texttt{sub}, \texttt{mul}, \texttt{fract}, \texttt{div}, \texttt{fma}, and \texttt{sqrt} instructions.

  • Must support the 64-bit floating-point type (\texttt{f64}), 64-bit packed floating-point type (\texttt{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 (on page 118)) and all integer rounding modes (see 5.19.4 Description of integer rounding modes (on page 175)).

  • Must support floating-point subnormal values.

  • Must support the \texttt{ftz} modifier and IEEE/ANSI Standard 754-2008 gradual underflow.

  • For all floating-point arithmetic instructions (see 5.11 Floating-point arithmetic instructions
(on page 151)) and cvt with a floating-point source and destination type (see 5.19 Conversion (cvt) instruction (on page 171)), 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) (on page 120).

- 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 (on page 323)) 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 (on page 290)) for the five exceptions specified in 12.2 Hardware exceptions (on page 288). 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 (on page 121).
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.

Machine 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, \( \text{div}(src0, src1) \) could be replaced by \( src0 \times \text{nrcp}(src1) \) 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 machine 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 (on page 42)) 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 (on page 248)), to behave differently.

Consider the following pseudo HSAIL example:

```hsail
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:

```hsail
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) { } if (x) { }
A; cross-lane-operation; A;
cross-lane-operation; } else { } else {
B; } else {
B; cross-lane-operation; B;
cross-lane-operation; }

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.

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:
\[
\begin{align*}
  & y = 1; \\
  & \text{if} \ (x != 1) \ { } \ \{ \\
  & \quad y = 0; \\
  & \} \\
\end{align*}
\]

Thread 2:
\[
\begin{align*}
  & x = 1; \\
  & \text{if} \ (y != 1) \ { } \ \{ \\
  & \quad x = 0; \\
  & \} \\
\end{align*}
\]

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 \texttt{align} modifier. This might allow the finalizer to generate more efficient machine 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 since the start of the associated kernel dispatch, it should be marked with the \texttt{const} modifier. On some implementations, knowing a load is accessing memory that has not changed since the start of the associated kernel dispatch might be more efficient. The program execution is undefined if a memory location accessed by a load marked \texttt{const} is changed since the start of the execution of the associated kernel dispatch; on some implementations this might result in incorrect values being loaded. See \texttt{6.3 Load (ld) instruction (on page 187)}.

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 \texttt{const} qualifier. See \texttt{4.3.10 Declaration and definition qualifiers (on page 73)}.

An HSAIL global or readonly segment variable definition marked with the \texttt{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 nonull 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 nonull 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 (on page 36)). If a high-level compiler can determine that an address is either the null pointer value or an address in the global segment, if 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 (on page 290)), 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 (on page 79).
- 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 floating-point constant; adjacent aggregate constant elements of the same type or array element type may be collapsed to a single array type element; an integer symbolic expression constant may be canonicalized such as removing a 0 offset argument to addr; or consecutive aggregate constant zero bytes (specified as literals or the aggregate constant zero element) may be collapsed to use a single aggregate constant zero element. See 18.6.1 Constant operands (on page 382).
- 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 hsa Brig Directive None t 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.
For information on how to support vendor specific extensions in BRIG, see 13.1.3 How to set up extensions (on page 295).

18.2 BRIG module

An HSAIL module (see 4.3 Module (on page 57)) is represented in BRIG as a single contiguous block of memory that contains the following elements:

- a hsa Brig_module_header_t
- a BRIG section index
- three or more BRIG sections

These elements can be positioned within the BRIG module in any order, except that the hsa Brig_module_header_t (see 18.3.18 hsa Brig_module_header_t (on page 346)) must start at offset 0 from the start of the BRIG module. Elements must not overlap.

The base of the hsa Brig_module_header_t 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.

hsa Brig_module_t is a pointer to the contiguous memory for a single BRIG module:

typedef hsa Brig_module_header_t* hsa Brig_module_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 (on page 360).
- 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 machine code. See 18.5 hsa_code section (on page 360).
- 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 (on page 380).

The BRIG section index is indexed by hsa Brig_section_index_t (see 18.3.24 hsa Brig_section_index_t (on page 351)). 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 hsa Brig_section_header_t, is not specified by HSAIL (see 18.3.25 hsa Brig_section_header_t (on page 351)). 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 `hsa_brig_section_header_t` 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 18.3.25 `hsa_brig_section_header_t` (on page 351).

For the standard BRIG sections, `hsa_data`, `hsa_code`, and `hsa_operand`, the `hsa_brig_section_header_t` 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 `hsa_brig_base_t` structure (see 18.3.6 `hsa_brig_base_t` (on page 342)) 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 `hsa_brig_section_header_t` uses `uint64_t` for the section size.) See 18.3.1 Section offsets (on the next page).

A number of entries in the `hsa_code` and `hsa_operand` sections (for example, `hsa_brig_directive_control_t` and `hsa_brig_operand_code_list_t`) 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 `hsa_brig_module_header_t`, the fields in `hsa_brig_section_header_t`, 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 `hsa_brig_section_header_t` 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.

```c
typedef uint32_t hsa Brig_data_offset32_t;
typedef uint32_t hsa Brig_code_offset32_t;
typedef uint32_t hsa Brig_operand_offset32_t;
```

Note that an offset into the hsa_data section references the start of the hsa Brig_data_t entry, not the data, which starts at the bytes field within hsa Brig_data_t (see 18.4 hsa.data section (on page 360)). For hsa_data section offsets, the following types are used to indicate the contents of the hsa_data section entry referenced:

```c
typedef hsa Brig_data_offset32_t hsa Brig_data_offset_string32_t;
typedef hsa Brig_data_offset32_t hsa Brig_data_offset_code_list32_t;
typedef hsa Brig_data_offset32_t hsa Brig_data_offset_operand_list32_t;
```

- `hsa Brig_data_offset_string32_t` — The entry contains a textual string or byte data.
- `hsa Brig_data_offset_code_list32_t` — The entry contains an array of hsa Brig_code_offset32_t values. The byte_count of the entry must be exactly \(4 \times\) number of array elements.
- `hsa Brig_data_offset_operand_list32_t` — The entry contains an array of hsa Brig_operand_offset32_t values. The byte_count of the hsa_data section entry must be exactly \(4 \times\) number of array elements.

18.3.2 hsa Brig_alignment_t

hsa Brig_alignment_t 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 HSA_BRIG_ALIGNMENT_1 means any byte boundary, HSA_BRIG_ALIGNMENT_2 is any even byte boundary, HSA_BRIG_ALIGNMENT_4 is any multiple of four, and so forth. For more information, see 4.3.10 Declaration and definition qualifiers (on page 73).

```c
typedef uint8_t hsa Brig_alignment8_t;
typedef enum {
    HSA_BRIG_ALIGNMENT_NONE = 0,
    HSA_BRIG_ALIGNMENT_1 = 1,
    HSA_BRIG_ALIGNMENT_2 = 2,
    HSA_BRIG_ALIGNMENT_4 = 3,
    HSA_BRIG_ALIGNMENT_8 = 4,
    HSA_BRIG_ALIGNMENT_16 = 5,
    HSA_BRIG_ALIGNMENT_32 = 6,
    HSA_BRIG_ALIGNMENT_64 = 7,
    HSA_BRIG_ALIGNMENT_128 = 8,
    HSA_BRIG_ALIGNMENT_256 = 9,
    HSA_BRIG_ALIGNMENT_MAX = HSA_BRIG_ALIGNMENT_256
} hsa Brig_alignment_t;
```

18.3.3 hsa Brig_allocation_t

hsa Brig_allocation_t is used to specify the memory allocation for variables. For more information, see 4.3.10 Declaration and definition qualifiers (on page 73).

```c
typedef uint8_t hsa Brig_allocation8_t;
typedef enum {
```
18.3.4 hsa Brig Alu Modifier

defines bit masks that can be used to access the modifiers for arithmetic logic unit instructions.

typedef uint8_t hsa Brig Alu Modifier8_t;

typedef enum {
  HSA_Brig_ALU_MODIFIER_FTZ = 1 << 0,
  HSA_Brig_ALU_MODIFIER_INTEGER_SAT = 1 << 1
} hsa Brig Alu Modifier_t;

- **HSA Brig ALU Modifier 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.

- **HSA Brig ALU Modifier Integer Sat** — A bit mask that can be used to select the setting for the sat (integer saturate) modifier. If the instruction destination and source types do not allow integer saturation or the instruction does not support the sat 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 hsa Brig Atomic Operation

t is used to specify the type of atomic memory and signal instructions.

For more information, see 6.5 Atomic memory instructions (on page 194) and 6.8 Notification (signal) instructions (on page 202).

typedef uint8_t hsa Brig Atomic Operation8_t;

typedef enum {
  HSA_Brig_ATOMIC_OPERATION_ADD = 0,
  HSA_Brig_ATOMIC_OPERATION_AND = 1,
  HSA_Brig_ATOMIC_OPERATION_CAS = 2,
  HSA_Brig_ATOMIC_OPERATION_EXCH = 3,
  HSA_Brig_ATOMIC_OPERATION_LD = 4,
  HSA_Brig_ATOMIC_OPERATION_MAX = 5,
  HSA_Brig_ATOMIC_OPERATION_MIN = 6,
  HSA_Brig_ATOMIC_OPERATION_OR = 7,
  HSA_Brig_ATOMIC_OPERATION_ST = 8,
  HSA_Brig_ATOMIC_OPERATION_SUB = 9,
  HSA_Brig_ATOMIC_OPERATION_WRAPEC = 10,
  HSA_Brig_ATOMIC_OPERATION_WRAPINC = 11,
  HSA_Brig_ATOMIC_OPERATION_XOR = 12,
  HSA_Brig_ATOMIC_OPERATION_WAIT_EQ = 13,
  HSA_Brig_ATOMIC_OPERATION_WAIT_NE = 14,
  HSA_Brig_ATOMIC_OPERATION_WAIT_LT = 15,
  HSA_Brig_ATOMIC_OPERATION_WAIT_GTE = 16,
  HSA_Brig_ATOMIC_OPERATION_WAIT_TIMEOUT_EQ = 17,
  HSA_Brig_ATOMIC_OPERATION_WAIT_TIMEOUT_NE = 18,
  HSA_Brig_ATOMIC_OPERATION_WAIT_TIMEOUT_LT = 19,
  HSA_Brig_ATOMIC_OPERATION_WAIT_TIMEOUT_GTE = 20,
  HSA_Brig_ATOMIC_OPERATION_FIRST_USER_DEFINED = 128
} hsa Brig Atomic Operation_t;

Values 21 through 127 are reserved, but values 128 to 255 are available for implementation defined extensions.
18.3.6 hsa Brig base_t

All entries in the hsa code and hsa operand sections start with the hsa Brig base_t structure.

Syntax is:

```c
typedef struct hsa Brig base_s {
    uint16_t byte_count;
    hsa Brig kind16_t kind;
} hsa Brig base_t;
```

Fields are:

- `uint16_t byte_size` — Size of the entry in bytes, including the hsa Brig base_t structure. Must be a multiple of 4.
- `hsa Brig kind16_t kind` — Can be any member of the hsa Brig kind_t enumeration indicating the kind of this entry. Must only be HSA Brig KIND DIRECTIVE_* or HSA Brig KIND INST_* for entries in the hsa code section, and HSA Brig KIND OPERAND_* for entries in the hsa operand section. See 18.3.12 hsa Brig kind_t (on page 344).

18.3.7 hsa Brig compare operation_t

hsa Brig compare operation_t is used to specify the type of compare operation. For more information, see 5.18 Compare (cmp) instruction (on page 166).

```c
typedef uint8_t hsa Brig compare operation8_t;
typedef enum {
    HSA Brig compare operation_EQ = 0,
    HSA Brig compare operation_NE = 1,
    HSA Brig compare operation_LT = 2,
    HSA Brig compare operation_LE = 3,
    HSA Brig compare operation_GT = 4,
    HSA Brig compare operation_GE = 5,
    HSA Brig compare operation_EQU = 6,
    HSA Brig compare operation_NEU = 7,
    HSA Brig compare operation_LTU = 8,
    HSA Brig compare operation_LEU = 9,
    HSA Brig compare operation_GTU = 10,
    HSA Brig compare operation_GEU = 11,
    HSA Brig compare operation_NUM = 12,
    HSA Brig compare operation.NaN = 13,
    HSA Brig compare operation_SEQ = 14,
    HSA Brig compare operation_SNE = 15,
    HSA Brig compare operation_SLT = 16,
    HSA Brig compare operation_SLE = 17,
    HSA Brig compare operation_GTG = 18,
    HSA Brig compare operation_GEG = 19,
    HSA Brig compare operation_SGEU = 20,
    HSA Brig compare operation_SEQU = 21,
    HSA Brig compare operation_SNUE = 22,
    HSA Brig compare operation_SLTU = 23,
    HSA Brig compare operation_SLEU = 24,
    HSA Brig compare operation_SNUN = 25,
    HSA Brig compare operation_SNNAN = 26,
    HSA Brig compare operation_SGTU = 27,
    HSA Brig compare operation_FIRST_USER_DEFINED = 128
} hsa Brig compare operation_t;
```

Values 28 through 127 are reserved, but values 128 to 255 are available for implementation defined extensions.
18.3.8 hsa_brig_controlDirective_t

hsa_brig_controlDirective_t is used to specify the type of control directive. For more information, see 13.5 Control directives for low-level performance tuning (on page 317).

typedef uint16_t hsa_brig_controlDirective16_t;
typedef enum {
    HSA_BRIG_CONTROL_DIRECTIVE_NONE = 0,
    HSA_BRIG_CONTROL_DIRECTIVE_ENABLEBREAKEXCEPTIONS = 1,
    HSA_BRIG_CONTROL_DIRECTIVE_ENABLEDTEECTEXCEPTIONS = 2,
    HSA_BRIG_CONTROL_DIRECTIVE_MAXDYNAMICGROUPSIZE = 3,
    HSA_BRIG_CONTROL_DIRECTIVE_MAXFLATGRIDSIZE = 4,
    HSA_BRIG_CONTROL_DIRECTIVE_MAXFLATWORKGROUPSIZE = 5,
    HSA_BRIG_CONTROL_DIRECTIVE_REQUIREDDIM = 6,
    HSA_BRIG_CONTROL_DIRECTIVE_REQUIREDGRIDSIZE = 7,
    HSA_BRIG_CONTROL_DIRECTIVE_REQUIREDWORKGROUPSIZE = 8,
    HSA_BRIG_CONTROL_DIRECTIVE_REQUIRENOPARTIALWORKGROUPS = 9,
    HSA_BRIG_CONTROL_DIRECTIVE_REQUIRENOPARTIALWAVEFRONTS = 10,
    HSA_BRIG_CONTROL_DIRECTIVE_REQUIREGROUPBASEALIGN = 11,
    HSA_BRIG_CONTROL_DIRECTIVE_FIRST_USER_DEFINED = 32768
} hsa_brig_controlDirective_t;

Values 11 through 32767 are reserved, but values 32768 to 65535 are available for implementation defined extensions.

18.3.9 hsa_brig_exceptions_t

hsa_brig_exceptions_t defines the bit mask used to specify a set of exceptions for each of the five exceptions specified in 12.2 Hardware exceptions (on page 288). For more information, see 11.2 Exception instructions (on page 279).

typedef uint32_t hsa_brig_exceptions32_t;
typedef enum {
    HSA_BRIG_EXCEPTIONS_INVALID_OPERATION = 1 << 0,
    HSA_BRIG_EXCEPTIONS_DIVIDE_BY_ZERO = 1 << 1,
    HSA_BRIG_EXCEPTIONS_OVERFLOW = 1 << 2,
    HSA_BRIG_EXCEPTIONS_UNDERFLOW = 1 << 3,
    HSA_BRIG_EXCEPTIONS_INEXACT = 1 << 4,
    HSA_BRIG_EXCEPTIONS_FIRST_USER_DEFINED = 1 << 16
} hsa_brig_exceptions_t;

Bits 5 through 15 are reserved, but bits 16 to 32 are available for implementation defined extensions.

18.3.10 hsa_brig_executableModifier_t

hsa_brig_executableModifier_t defines bit masks that can be used to access properties about an executable kernel or function.

typedef uint8_t hsa_brig_executableModifier8_t;
typedef enum {
    HSA_BRIG_EXECUTABLE_MODIFIER_DEFINITION = 1
} hsa_brig_executableModifier_t;

- HSA_BRIG_EXECUTABLE_MODIFIER_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 hsa_brigDirective_executable_t (on page 363).

18.3.11 hsa_brig_expression_operation_t

hsa_brig_expression_operation_t is used to specify the operation of a constant expression.
typedef uint16_t hsa_brig_expression_operation16_t;
typedef enum {
    HSA_BRIG_EXPRESSION_OPERATION_NULLPTR_FLAT = 0,
    HSA_BRIG_EXPRESSION_OPERATION_NULLPTR_GROUP = 1,
    HSA_BRIG_EXPRESSION_OPERATION_NULLPTR_PRIVATE = 2,
    HSA_BRIG_EXPRESSION_OPERATION_NULLPTR_KERNARG = 3,
    HSA_BRIG_EXPRESSION_OPERATION_ADDR = 4,
    HSA_BRIG_EXPRESSION_OPERATION_FIRST_USER_DEFINED = 32768
} hsa_brig_expression_operation_t;

18.3.12 hsa_brig_kind_t

hsa_brig_kind_t 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 hsa_brig_kind16_t;
typedef enum {
    HSA_BRIG_KIND_NONE = 0x0000,
    HSA_BRIG_KINDDirective_BEGIN = 0x1000,
    HSA_BRIG_KINDDirective_ARG_BLOCK_END = 0x1001,
    HSA_BRIG_KINDDirective_ARG_BLOCK_START = 0x1002,
    HSA_BRIG_KINDDirective_COMMENT = 0x1003,
    HSA_BRIG_KINDDirective_EXTENSION = 0x1004,
    HSA_BRIG_KINDDirective_FUNCTION = 0x1005,
    HSA_BRIG_KINDDirective INDIRECT_FUNCTION = 0x1006,
    HSA_BRIG_KINDDirective INDIRECT_KERNEL = 0x1007,
    HSA_BRIG_KINDDirective LABEL = 0x1008,
    HSA_BRIG_KINDDirective LOC = 0x1009,
    HSA_BRIG_KINDDirective_MODULE = 0x100a,
    HSA_BRIG_KINDDirectivePragma = 0x100b,
    HSA_BRIG_KINDDirective_SIGMA = 0x100c,
    HSA_BRIG_KINDDirective_VARIABLE = 0x100d,
    HSA_BRIG_KINDDirective_EXTENSION_VERSION = 0x100e,
    HSA_BRIG_KINDDirective_EXTENSION_VERSION = 0x100f,
    HSA_BRIG_KINDDirective_END = 0x1010,
    HSA_BRIG_KIND_INST_BEGIN = 0x2000,
    HSA_BRIG_KIND_INST_ADDR = 0x2001,
    HSA_BRIG_KIND_INST_ATOMIC = 0x2002,
    HSA_BRIG_KIND_INST_BASIC = 0x2003,
    HSA_BRIG_KIND_INST_BR = 0x2004,
    HSA_BRIG_KIND_INST_CMP = 0x2005,
    HSA_BRIG_KIND_INST.CV = 0x2006,
    HSA_BRIG_KIND_INST_LANE = 0x2007,
    HSA_BRIG_KIND_INST_MEM = 0x2008,
    HSA_BRIG_KIND_INST_MEM_FENCE = 0x2009,
    HSA_BRIG_KIND_INST_MOD = 0x200a,
    HSA_BRIG_KIND_INST_QUERY_IMAGE = 0x200b,
    HSA_BRIG_KIND_INST_QUERY_SAMPLER = 0x200c,
    HSA_BRIG_KIND_INST_QUEUE = 0x200d,
    HSA_BRIG_KIND_INST_SEG = 0x200e,
    HSA_BRIG_KIND_INST_SIGMA = 0x200f,
    HSA_BRIG_KIND_INST_SIGNAL = 0x2010,
    HSA_BRIG_KIND_INST_SOURCE_TYPE = 0x2011,
    HSA_BRIG_KIND_INST_END = 0x2012,
    HSA_BRIG_KIND_OPERAND_BEGIN = 0x3000,
    HSA_BRIG_KIND_OPERAND_ADDRESS = 0x3001,
    HSA_BRIG_KIND_OPERAND_ALIGN = 0x3002,
    HSA_BRIG_KIND_OPERAND_CODE_LIST = 0x3003,
18.3.13 hsa Brig linkage_t

hsa Brig linkage_t is used to specify linkage. For more information, see 4.12 Linkage (on page 106).

typedef uint8_t hsa Brig linkage8_t;
typedef enum {
    HSA_BRIG_LINKAGE_NONE = 0,
    HSA_BRIG_LINKAGE_PROGRAM = 1,
    HSA_BRIG_LINKAGE_MODULE = 2,
    HSA_BRIG_LINKAGE_FUNCTION = 3,
    HSA_BRIG_LINKAGE_ARG = 4
} hsa Brig linkage_t;

18.3.14 hsa Brig machine model_t

hsa Brig machine model_t is used to specify the kind of machine model. For more information, see 2.9 Small and large machine models (on page 40).

typedef uint8_t hsa Brig machine model8_t;
typedef enum {
    HSA_BRIG_MACHINE_MODEL_SMALL = 0,
    HSA_BRIG_MACHINE_MODEL_LARGE = 1
} hsa Brig machine model_t;

18.3.15 hsa Brig memory modifier_t

hsa Brig memory modifier_t defines bit masks that can be used to access the modifiers for memory instructions.

typedef uint8_t hsa Brig memory modifier8_t;
typedef enum {
    HSA_BRIG_MEMORY_MODIFIER_CONST = 1,
    HSA_BRIG_MEMORY_MODIFIER_NONTEMPORAL = 2
} hsa Brig memory modifier_t;

- HSA_BRIG_MEMORY_MODIFIER_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.

- HSA_BRIG_MEMORY_MODIFIER_NONTEMPORAL — A bit mask that can be used to select the setting for the nt modifier. A 0 value means it is absent and a 1 value means it is present. If the instruction does not support the nt modifier, then the value must be 0.

18.3.16 hsa Brig memory order_t

hsa Brig memory order_t is used to specify the memory order of an atomic memory instruction. For more information, see 6.2.1 Memory order (on page 183).
typedef uint8_t has brig_memory_order8_t;
typedef enum {
    HSA_BRIG_MEMORY_ORDER_NONE = 0,
    HSA_BRIG_MEMORY_ORDER_RELAXED = 1,
    HSA_BRIG_MEMORY_ORDER_SC_ACQUIRE = 2,
    HSA_BRIG_MEMORY_ORDER_SC_RELEASE = 3,
    HSA_BRIG_MEMORY_ORDER_SC_ACQUIRE_RELEASE = 4
} has brig_memory_order_t;

18.3.17 has brig_memory_scope_t

has brig_memory_scope_t 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 (on page 184).
typedef uint8_t has brig_memory_scope8_t;
typedef enum {
    HSA_BRIG_MEMORY_SCOPE_NONE = 0,
    HSA_BRIG_MEMORY_SCOPE_WORKITEM = 1,
    HSA_BRIG_MEMORY_SCOPE_WAVEFRONT = 2,
    HSA_BRIG_MEMORY_SCOPE_WORKGROUP = 3,
    HSA_BRIG_MEMORY_SCOPE_AGENT = 4,
    HSA_BRIG_MEMORY_SCOPE_SYSTEM = 5
} has brig_memory_scope_t;

18.3.18 has brig_module_header_t

The first entry in a BRIG module must be has brig_module_header_t. It must be 16-byte aligned. See
18.2 BRIG module (on page 338).

Syntax is:
typedef struct has brig_module_header_s {
    char identification[8];
    has brig_version32_t brig_major;
    has brig_version32_t brig_minor;
    uint64_t byte_count;
    uint8_t hash[64];
    uint32_t reserved;
    uint32_t section_count;
    uint64_t section_index;
} has brig_module_header_t;

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”.

- has brig_version32_t brig_major — The BRIG object format major version. When
  generating BRIG, must be HSA_BRIG_VERSION_BRIG_MAJOR. When consuming BRIG, must be
  HSA_BRIG_VERSION_BRIG_MAJOR to be compatible with this revision of the BRIG object format
  specification. See 18.3.31 has brig_version_t (on page 355).

- has brig_version32_t brig_minor — The BRIG object format minor version. When
  generating BRIG, must be HSA_BRIG_VERSION_BRIG_MINOR. When consuming BRIG, brig_
  major must be HSA_BRIG_VERSION_BRIG_MAJOR and brig_minor must be less than or
  equal to HSA_BRIG_VERSION_BRIG_MINOR to be compatible with this revision of the BRIG object
  format specification. See 18.3.31 has brig_version_t (on page 355).

- uint64_t byte_count — 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 section_count** — Number of sections in the module. Must be at least 3 for the standard sections. See 18.2 BRIG module (on page 338).

- **uint64_t section_index** — 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 section_count entries of type uint64_t in the the array. See 18.2 BRIG module (on page 338).

### 18.3.19 hsa brig opcode_t

**hsa brig opcode_t** is used to specify the opcode for the HSAIL instruction.

```c
typedef uint16_t hsa_brig_opcode16_t;
typedef enum {
    HSA_BRIG_OPCODE_NOP = 0,
    HSA_BRIG_OPCODE_ABS = 1,
    HSA_BRIG_OPCODE_ADD = 2,
    HSA_BRIG_OPCODE_BORROW = 3,
    HSA_BRIG_OPCODE_CARRY = 4,
    HSA_BRIG_OPCODE_CEIL = 5,
    HSA_BRIG_OPCODE_COPYSIGN = 6,
    HSA_BRIG_OPCODE_DIV = 7,
    HSA_BRIG_OPCODE_FLOOR = 8,
    HSA_BRIG_OPCODE_FMA = 9,
    HSA_BRIG_OPCODE_FRACT = 10,
    HSA_BRIG_OPCODE_MAX = 11,
    HSA_BRIG_OPCODE_MIN = 12,
    HSA_BRIG_OPCODE_MUL = 14,
    HSA_BRIG_OPCODE_MUL_HI = 15,
    HSA_BRIG_OPCODE_NEG = 16,
    HSA_BRIG_OPCODE_REM = 17,
    HSA_BRIG_OPCODE_RINT = 18,
    HSA_BRIG_OPCODE_SQRT = 19,
    HSA_BRIG_OPCODE_SUB = 20,
    HSA_BRIG_OPCODE_TRUNC = 21,
    HSA_BRIG_OPCODE_MAD24 = 22,
    HSA_BRIG_OPCODE_MAD24_HI = 23,
    HSA_BRIG_OPCODE_MUL24 = 24,
    HSA_BRIG_OPCODE_MUL24_HI = 25,
    HSA_BRIG_OPCODE_SHL = 26,
    HSA_BRIG_OPCODE_SHR = 27,
    HSA_BRIG_OPCODE_AND = 28,
    HSA_BRIG_OPCODE_NOT = 29,
    HSA_BRIG_OPCODE_OR = 30,
    HSA_BRIG_OPCODE_POPCOUNT = 31,
    HSA_BRIG_OPCODE_XOR = 32,
    HSA_BRIG_OPCODE_BITEXTRACT = 33,
    HSA_BRIG_OPCODE_BITINSERT = 34,
    HSA_BRIG_OPCODE_BITMASK = 35,
    HSA_BRIG_OPCODE_BITREV = 36,
    HSA_BRIG_OPCODE_BITSELECT = 37,
    HSA_BRIG_OPCODE_FIRSTBIT = 38,
    HSA_BRIG_OPCODE_LASTBIT = 39,
    HSA_BRIG_OPCODE_COMBINE = 40,
    HSA_BRIG_OPCODE_EXPAND = 41,
}```
Chapter 18. BRIG: HSAIL Binary Format

18.3 Support types

HSA_BRIG_OPCODE_LDA = 42,
HSA_BRIG_OPCODE_MOV = 43,
HSA_BRIG_OPCODE_SHUFFLE = 44,
HSA_BRIG_OPCODE_UNPACKHI = 45,
HSA_BRIG_OPCODE_UNPACKLO = 46,
HSA_BRIG_OPCODE_PACK = 47,
HSA_BRIG_OPCODE_UNPACK = 48,
HSA_BRIG_OPCODE_SAD = 64,
HSA_BRIG_OPCODE_SADHI = 65,
HSA_BRIG_OPCODE_MUTEX = 66,
HSA_BRIG_OPCODE_FTOS = 67,
HSA_BRIG_OPCODE_STOF = 68,
HSA_BRIG_OPCODE_CMP = 69,
HSA_BRIG_OPCODE_CVT = 70,
HSA_BRIG_OPCODE_LD = 71,
HSA_BRIG_OPCODE_ST = 72,
HSA_BRIG_OPCODE_ATOMIC = 73,
HSA_BRIG_OPCODE_ATOMICNORET = 74,
HSA_BRIG_OPCODE_SIGNAL = 75,
HSA_BRIG_OPCODE_SIGNALNORET = 76,
HSA_BRIG_OPCODE_MEMFENCE = 77,
HSA_BRIG_OPCODE_RDIMAGE = 78,
HSA_BRIG_OPCODE_RDWRIMAGE = 79,
HSA_BRIG_OPCODE_STIMAGE = 80,
HSA_BRIG_OPCODE_IMAGEFENCE = 81,
HSA_BRIG_OPCODE_QUERYIMAGE = 82,
HSA_BRIG_OPCODE_QUERYSAMPLE = 83,
HSA_BRIG_OPCODE_CBR = 84,
HSA_BRIG_OPCODE_BR = 85,
HSA_BRIG_OPCODE_SBR = 86,
HSA_BRIG_OPCODE_BARRIER = 87,
HSA_BRIG_OPCODE_WAVEBARRIER = 88,
HSA_BRIG_OPCODE_INITFBARRIER = 89,
HSA_BRIG_OPCODE_JOINFBARRIER = 90,
HSA_BRIG_OPCODE_LEAVEFBARRIER = 91,
HSA_BRIG_OPCODE_RELEASEFBARRIER = 92,
HSA_BRIG_OPCODE_WAITFBARRIER = 93,
HSA_BRIG_OPCODE_LDF = 94,
HSA_BRIG_OPCODE_CURRENTWORKGROUPSIZE = 105,
HSA_BRIGR_OPCODE_CURRENTWORKITEMFLATID = 106,
HSA_BRIGR_OPCODE_DIM = 107,
HSA_BRIGR_OPCODE_GRIDGROUPS = 108,
HSA_BRIGR_OPCODE_GIDSIZE = 109,
HSA_BRIGR_OPCODE_PACKETCOMPLETIONSIGN = 110,
HSA_BRIGR_OPCODE_PACKETID = 111,
HSA_BRIGR_OPCODE_WORKGROUPID = 112,
HSA_BRIGR_OPCODE_WORKITEMABSID = 114,
HSA_BRIGR_OPCODE_WORKITEMFLATABSID = 115,
HSA_BRIGR_OPCODE_WORKITEMFLATID = 116,
HSA_BRIGR_OPCODE_WORKITEMID = 117,
HSA_BRIGR_OPCODE_CLEARDETECTEXCEPT = 118,
HSA_BRIGR_OPCODE_GETDETECTEXCEPT = 119,
HSA_BRIGR_OPCODE_SETDETECTEXCEPT = 120,
HSA_BRIGR_OPCODE_ADDQUEUEWRITEINDEX = 121,
HSA_BRIGR_OPCODE_CASQUEUEWRITEINDEX = 122,
HSA_BRIGR_OPCODE_LDQUEUEREADINDEX = 123,
HSA_BRIGR_OPCODE_LDQUEUEWRITEINDEX = 124,
HSA_BRIGR_OPCODE_STQUEUEREADINDEX = 125,
HSA_BRIGR_OPCODE_STQUEUEWRITEINDEX = 126,
HSA_BRIGR_OPCODE_CLOCK = 127,
HSA_BRIGR_OPCODE_CUID = 128,
HSA_BRIGR_OPCODE_DEBUGTRAP = 129,
HSA_BRIGR_OPCODE_GROUPBASEPTR = 130,
HSA_BRIGR_OPCODE_KERNARGBASEPTR = 131,
HSA_BRIGR_OPCODE_LANEID = 132,
HSA_BRIGR_OPCODE_MAXCUID = 133,
HSA_BRIGR_OPCODE_MAXWAVEID = 134,
HSA_BRIGR_OPCODE_NULLPTR = 135,
HSA_BRIGR_OPCODE_WAVEID = 136,
HSA_BRIGR_OPCODE_GROUPSTATICSIZE = 137,
HSA_BRIGR_OPCODE_GROUPTOTALSIZE = 138,
HSA_BRIGR_OPCODE_FIRST_USER_DEFINED = 32768
} hsa_brig_opcode_t;

Values 139 through 32767 are reserved, but values 32768 to 65535 are available for implementation defined extensions.

### 18.3.20 hsa_brig_pack_t

hsa_brig_pack_t is used to specify the kind of packing control for packed data. For more information, see 4.14 Packing controls for packed data (on page 110).

typedef uint8_t hsa_brig_pack8_t;
typedef enum {
    HSA_BRIGR_PACK_NONE = 0,
    HSA_BRIGR_PACK_PP = 1,
    HSA_BRIGR_PACK_PS = 2,
    HSA_BRIGR_PACK_SP = 3,
    HSA_BRIGR_PACK_SS = 4,
    HSA_BRIGR_PACK_S = 5,
    HSA_BRIGR_PACK_P = 6,
    HSA_BRIGR_PACK_PPSAT = 7,
    HSA_BRIGR_PACK_PSSAT = 8,
    HSA_BRIGR_PACK_SPSAT = 9,
    HSA_BRIGR_PACK_SSSAT = 10,
    HSA_BRIGR_PACK_SSAT = 11,
    HSA_BRIGR_PACK_PSAT = 12
} hsa_brig_pack_t;
18.3.21 hsa_brig_profile_t

hsa_brig_profile_t is used to specify the kind of profile. For more information, see 16.1 What are profiles? (on page 327).

typedef uint8_t hsa_brig_profile8_t;
typedef enum {
    HSA_BRIG_PROFILE_BASE = 0,
    HSA_BRIG_PROFILE_FULL = 1
} hsa_brig_profile_t;

18.3.22 hsa_brig_register_kind_t

hsa_brig_register_kind_t is used to specify the kind of HSAI register. For more information, see 4.7 Registers (on page 83).

typedef uint16_t hsa_brig_register_kind16_t;
typedef enum {
    HSA_BRIG_REGISTER_KIND_CONTROL = 0,
    HSA_BRIG_REGISTER_KIND_SINGLE = 1,
    HSA_BRIG_REGISTER_KIND_DOUBLE = 2,
    HSA_BRIG_REGISTER_KIND_QUAD = 3
} hsa_brig_register_kind_t;

18.3.23 hsa_brig_round_t

hsa_brig_round_t is used to specify rounding. For more information, see 4.19.2 Floating-point rounding (on page 118) and 5.19.3 Rules for rounding for conversions (on page 175).

If the instruction does not support a rounding mode, then HSA_BRIG_ROUND_NONE must be used.

If the instruction supports a floating-point rounding mode but does not explicitly specify one, then HSA_BRIG_ROUND_FLOAT_DEFAULT must be specified. If the instruction supports an integer rounding mode but does not explicitly specify one, then HSA_BRIG_ROUND_INTEGER_ZERO must be specified. Otherwise, the appropriate rounding mode must be used.

typedef uint8_t hsa_brig_round8_t;
typedef enum {
    HSA_BRIG_ROUND_NONE = 0,
    HSA_BRIG_ROUND_FLOAT_DEFAULT = 1,
    HSA_BRIG_ROUND_FLOAT_NEAR_EVEN = 2,
    HSA_BRIG_ROUND_FLOAT_ZERO = 3,
    HSA_BRIG_ROUND_FLOAT_PLUS_INFINITY = 4,
    HSA_BRIG_ROUND_FLOAT_MINUS_INFINITY = 5,
    HSA_BRIG_ROUND_INTEGER_NEAR_EVEN = 6,
    HSA_BRIG_ROUND_INTEGER_ZERO = 7,
    HSA_BRIG_ROUND_INTEGER_PLUS_INFINITY = 8,
    HSA_BRIG_ROUND_INTEGER_MINUS_INFINITY = 9,
    HSA_BRIG_ROUND_INTEGER_NEAR_EVEN_SAT = 10,
    HSA_BRIG_ROUND_INTEGER_MINUS_INFINITY_SAT = 11,
    HSA_BRIG_ROUND_INTEGER_PLUS_INFINITY_SAT = 12,
    HSA_BRIG_ROUND_INTEGER_MINUS_INFINITY_SAT = 13,
    HSA_BRIG_ROUND_INTEGER_SIGNALING_NEAR_EVEN = 14,
    HSA_BRIG_ROUND_INTEGER_SIGNALING_ZERO = 15,
    HSA_BRIG_ROUND_INTEGER_SIGNALING_PLUS_INFINITY = 16,
    HSA_BRIG_ROUND_INTEGER_SIGNALING_MINUS_INFINITY = 17,
    HSA_BRIG_ROUND_INTEGER_SIGNALING_NEAR_EVEN_SAT = 18,
    HSA_BRIG_ROUND_INTEGER_SIGNALING_MINUS_INFINITY_SAT = 19,
    HSA_BRIG_ROUND_INTEGER_SIGNALING_PLUS_INFINITY_SAT = 20,
    HSA_BRIG_ROUND_INTEGER_SIGNALING_MINUS_INFINITY_SAT = 21
} hsa_brig_round_t;
18.3.24 hsa brig section index_t

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 hsa brig section index_t. Any additional sections have an index starting after these. See 18.2 BRIG module (on page 338).

typedef uint32_t hsa brig section index32_t;
typedef enum {
  HSA_BRIG_SECTION_INDEX_DATA = 0,
  HSA_BRIG_SECTION_INDEX_CODE = 1,
  HSA_BRIG_SECTION_INDEX_OPERAND = 2,
  HSA_BRIG_SECTION_INDEX_FIRST_USER_DEFINED = 3
} hsa brig section index_t;

18.3.25 hsa brig section header_t

The first entry in every BRIG section must be hsa brig section header_t. It must be 16-byte aligned. See 18.2 BRIG module (on page 338).

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:

typedef struct hsa brig section header_s {
  uint64_t byte_count;
  uint32_t header_byte_count;
  uint32_t name length;
  uint8_t name[1];
} hsa brig section header_t;

Field is:

- uint64_t byte_count — Size in bytes of the section, including the size of the hsa brig section header_t. Must be a multiple of 4.
- uint32_t header_byte_count — 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 name length — Length of the section name in bytes.
- uint8_t name[1] — Section name, name length bytes long.

The section name may be followed by any implementation specific data. This must be followed by sufficient zero padding bytes to make header byte count a multiple of 4.

18.3.26 hsa brig seg cvt modifier_t

hsa brig seg cvt modifier_t defines bit masks that can be used to access the modifiers for instructions which convert between segment and flat addresses.

typedef uint8_t hsa brig seg cvt modifier8_t;
typedef enum {
  HSA_BRIG_SEG_CVT_MODIFIER_NONNULL = 1
} hsa brig seg cvt modifier_t;

- HSA BRIG SEG CVT_MODIFIER_NONNULL — A bit mask that can be used to select the setting for the nonull modifier. A 0 value means it is absent and a 1 value means it is present. If the instruction does not support the nonull modifier, then the value must be 0.
18.3.27 hsa Brig segment_t

hsa Brig segment_t 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 (on page 32).

typedef uint8_t hsa Brig segment8_t;
typedef enum {
    HSA Brig SEGMENT_NONE = 0,
    HSA Brig SEGMENT_FLAT = 1,
    HSA Brig SEGMENT_GLOBAL = 2,
    HSA Brig SEGMENT_READONLY = 3,
    HSA Brig SEGMENT_KERNARG = 4,
    HSA Brig SEGMENT_GROUP = 5,
    HSA Brig SEGMENT_PRIVATE = 6,
    HSA Brig SEGMENT_SPILL = 7,
    HSA Brig SEGMENT_ARG = 8,
    HSA Brig SEGMENT_FIRST_USER_DEFINED = 128
} hsa Brig segment_t;

Values 9 through 127 are reserved, but values 128 to 255 are available for implementation defined extensions.

18.3.28 hsa Brig type_t

hsa Brig type_t is used to specify the data compound type of instructions, operands, and variables.

The hsa Brig type_t 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 (on page 108).

typedef enum {
    HSA Brig TYPE_CLASS_BASE_SIZE = 5,
    HSA Brig TYPE_CLASS_PACK_SIZE = 2,
    HSA Brig TYPE_CLASS_ARRAY_SIZE = 1,

    HSA Brig TYPE_CLASS_BASE_SHIFT = 0,
    HSA Brig TYPE_CLASS_PACK_SHIFT = HSA Brig TYPE_CLASS_BASE_SHIFT + HSA Brig TYPE_CLASS_BASE_SIZE,
    HSA Brig TYPE_CLASS_ARRAY_SHIFT = HSA Brig TYPE_CLASS_PACK_SHIFT + HSA Brig TYPE_CLASS_PACK_SIZE,

    HSA Brig TYPE_CLASS_BASE_MASK = ((1 << HSA Brig TYPE_CLASS_BASE_SIZE) - 1)
        << HSA Brig TYPE_CLASS_BASE_SHIFT,
    HSA Brig TYPE_CLASS_PACK_MASK = ((1 << HSA Brig TYPE_CLASS_BASE_SIZE) - 1)
        << HSA Brig TYPE_CLASS_PACK_SHIFT,
    HSA Brig TYPE_CLASS_ARRAY_MASK = ((1 << HSA Brig TYPE_CLASS_ARRAY_SIZE) - 1)
        << HSA Brig TYPE_CLASS_ARRAY_SHIFT,

    HSA Brig TYPE_CLASS_PACK_NONE = 0 << HSA Brig TYPE_CLASS_PACK_SHIFT,
    HSA Brig TYPE_CLASS_PACK_32 = 1 << HSA Brig TYPE_CLASS_PACK_SHIFT,

} hsa Brig type_t;
typedef uint16_t hasa_type_class_t;
typedef enum {
    hasa_type_class_U8 = 1,
    hasa_type_class_U16 = 2,
    hasa_type_class_U32 = 3,
    hasa_type_class_U64 = 4,

    hasa_type_class_S8 = 5,
    hasa_type_class_S16 = 6,
    hasa_type_class_S32 = 7,
    hasa_type_class_S64 = 8,

    hasa_type_class_F16 = 9,
    hasa_type_class_F32 = 10,
    hasa_type_class_F64 = 11,

    hasa_type_class_B1 = 12,
    hasa_type_class_B8 = 13,
    hasa_type_class_B16 = 14,
    hasa_type_class_B32 = 15,
    hasa_type_class_B64 = 16,
    hasa_type_class_B128 = 17,

    hasa_type_class_SAMP = 18,
    hasa_type_class_ROIMG = 19,
    hasa_type_class_WOIMG = 20,
    hasa_type_class_RWIMG = 21,

    hasa_type_class_S1G32 = 22,
    hasa_type_class_S1G64 = 23,
} hasa_type_class_array_t;

HSA_BRIG_TYPE_CLASS_PACK_64 = 2 << HSA_BRIG_TYPE_CLASS_PACK_SHIFT,
HSA_BRIG_TYPE_CLASS_PACK_128 = 3 << HSA_BRIG_TYPE_CLASS_PACK_SHIFT,
HSA_BRIG_TYPE_CLASS_ARRAY = 1 << HSA_BRIG_TYPE_CLASS_ARRAY_SHIFT
Chapter 18. BRIG: HSAIL Binary Format  18.3 Support types

HSA_BRIG_TYPE_F16X2  =  HSA_BRIG_TYPE_F16 | HSA_BRIG_TYPE_CLASS_PACK_32,
HSA_BRIG_TYPE_F16X4  =  HSA_BRIG_TYPE_F16 | HSA_BRIG_TYPE_CLASS_PACK_64,
HSA_BRIG_TYPE_F16X8  =  HSA_BRIG_TYPE_F16 | HSA_BRIG_TYPE_CLASS_PACK_128,
HSA_BRIG_TYPE_F32X2  =  HSA_BRIG_TYPE_F32 | HSA_BRIG_TYPE_CLASS_PACK_64,
HSA_BRIG_TYPE_F32X4  =  HSA_BRIG_TYPE_F32 | HSA_BRIG_TYPE_CLASS_PACK_128,
HSA_BRIG_TYPE_F32X8  =  HSA_BRIG_TYPE_F32 | HSA_BRIG_TYPE_CLASS_PACK_256,
HSA_BRIG_TYPE_F64X2  =  HSA_BRIG_TYPE_F64 | HSA_BRIG_TYPE_CLASS_PACK_64,
HSA_BRIG_TYPE_F64X4  =  HSA_BRIG_TYPE_F64 | HSA_BRIG_TYPE_CLASS_PACK_128,
HSA_BRIG_TYPE_U8X16  =  HSA_BRIG_TYPE_U8 | HSA_BRIG_TYPE_CLASS_ARRAY,
HSA_BRIG_TYPE_U16X16 =  HSA_BRIG_TYPE_U16 | HSA_BRIG_TYPE_CLASS_ARRAY,
HSA_BRIG_TYPE_U32X4  =  HSA_BRIG_TYPE_U32 | HSA_BRIG_TYPE_CLASS_ARRAY,
HSA_BRIG_TYPE_U64X4  =  HSA_BRIG_TYPE_U64 | HSA_BRIG_TYPE_CLASS_ARRAY,
HSA_BRIG_TYPE_S8X8   =  HSA_BRIG_TYPE_S8 | HSA_BRIG_TYPE_CLASS_ARRAY,
HSA_BRIG_TYPE_S16X4  =  HSA_BRIG_TYPE_S16 | HSA_BRIG_TYPE_CLASS_ARRAY,
HSA_BRIG_TYPE_S32X4  =  HSA_BRIG_TYPE_S32 | HSA_BRIG_TYPE_CLASS_ARRAY,
HSA_BRIG_TYPE_S64X4  =  HSA_BRIG_TYPE_S64 | HSA_BRIG_TYPE_CLASS_ARRAY,
HSA_BRIG_TYPE_B128   =  HSA_BRIG_TYPE_B128 | HSA_BRIG_TYPE_CLASS_ARRAY,
HSA_BRIG_TYPE_B256   =  HSA_BRIG_TYPE_B256 | HSA_BRIG_TYPE_CLASS_ARRAY,
18.3.29 hsa brig uint64_t

hsa brig uint64_t 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:

typedef struct hsa Brig_uint64_s {
  uint32_t lo;
  uint32_t hi;
} hsa Brig_uint64_t;

Fields are:

- uint32_t lo — The low 32 bits of the 64-bit integer. lo is combined with hi to form a 64-bit value:
  
  value = (uint64_t(hi) |< 32) << uint64_t(lo)

- uint32_t hi — The high 32 bits of the 64-bit integer.

18.3.30 hsa Brig variable modifier_t

hsa brig variable modifier_t defines bit masks that can be used to access properties about a variable.

typedef uint8_t hsa Brig variable modifier8_t;
typedef enum {
  HSA Brig VARIABLE MODIFIER DEFINITION = 1,
  HSA Brig VARIABLE MODIFIER CONST = 2
} hsa Brig variable modifier t;

- HSA Brig VARIABLE MODIFIER 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.

- HSA Brig VARIABLE MODIFIER 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 an initialized. Only global or readonly segment variables can be constant.

See 18.5.1.14 hsa Brig directive variable t (on page 369).

18.3.31 hsa Brig version_t

The literal values of hsa Brig version_t 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). To be conformant with this revision of the HSAIL virtual ISA specification, a producer must generate BRIG with these values and a consumer must accept BRIG with compatible values.
typedef uint32_t hsa brig version 32_t;
typedef enum {
    HSA BRIG VERSION_HSAIL_MAJOR = 1,
    HSA BRIG VERSION_HSAIL_MINOR = 2,
    HSA BRIG VERSION_BRIG_MAJOR = HSA BRIG VERSION_HSAIL_MAJOR,
    HSA BRIG VERSION_BRIG_MINOR = HSA BRIG VERSION_HSAIL_MINOR
} hsa brig version t;

- HSA 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 (on page 323). BRIG with an HSAIL major version different from this value is not compatible with this revision of the HSAIL virtual ISA specification.

- HSA 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 (on page 323). 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.

- HSA BRIG VERSION_BRIG_MAJOR — The major version of this revision of the BRIG object format specification. Must equal the value of HSA BRIG VERSION_HSAIL_MAJOR. This is the value used in the brig major field of the BRIG module header (see 18.3.18 hsa brig module header t (on page 346)). BRIG with a BRIG major version different from this value is not compatible with this revision of the BRIG object format specification.

- HSA BRIG VERSION_BRIG_MINOR — The minor version of this revision of the BRIG object format specification. Must equal the value of HSA BRIG VERSION_HSAIL_MINOR. This is the value used in the brig minor field of the BRIG module header (see 18.3.18 hsa brig module header t (on page 346)). 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.

18.3.32 hsa brig width t

hsa brig width t 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 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 (on page 42).

typedef uint8_t hsa brig width 8 t;
typedef enum {
    HSA BRIG WIDTH_NONE = 0,
    HSA BRIG WIDTH_1 = 1,
    HSA BRIG WIDTH_2 = 2,
    HSA BRIG WIDTH_4 = 3,
    HSA BRIG WIDTH_8 = 4,
    HSA BRIG WIDTH_16 = 5,
    HSA BRIG WIDTH_32 = 6,
    HSA BRIG WIDTH_64 = 7,
    HSA BRIG WIDTH_128 = 8,
    HSA BRIG WIDTH_256 = 9,
    HSA BRIG WIDTH_512 = 10,
    HSA BRIG WIDTH_1024 = 11,
    HSA BRIG WIDTH_2048 = 12,
    HSA BRIG WIDTH_4096 = 13,
    HSA BRIG WIDTH_8192 = 14,
18.3.33 hsa_ext_brig_image_channel_order_t

hsa_ext_brig_image_channel_order_t is used to specify the order of image components. For more information, see 7.1.4.1 Channel order (on page 213).

typedef uint8_t hsa_ext_brig_image_channel_order8_t;
typedef enum {
    HSA_EXT_BRIG_IMAGE_CHANNEL_ORDER_A = 0,
    HSA_EXT_BRIG_IMAGE_CHANNEL_ORDER_R = 1,
    HSA_EXT_BRIG_IMAGE_CHANNEL_ORDER_RX = 2,
    HSA_EXT_BRIG_IMAGE_CHANNEL_ORDER_RG = 3,
    HSA_EXT_BRIG_IMAGE_CHANNEL_ORDER_RGX = 4,
    HSA_EXT_BRIG_IMAGE_CHANNEL_ORDER_RA = 5,
    HSA_EXT_BRIG_IMAGE_CHANNEL_ORDER_RGB = 6,
    HSA_EXT_BRIG_IMAGE_CHANNEL_ORDER_RGBA = 7,
    HSA_EXT_BRIG_IMAGE_CHANNEL_ORDER_BGRA = 8,
    HSA_EXT_BRIG_IMAGE_CHANNEL_ORDER_BGRA = 9,
    HSA_EXT_BRIG_IMAGE_CHANNEL_ORDER_ARGB = 10,
    HSA_EXT_BRIG_IMAGE_CHANNEL_ORDER_ABGR = 11,
    HSA_EXT_BRIG_IMAGE_CHANNEL_ORDER_SRGB = 12,
    HSA_EXT_BRIG_IMAGE_CHANNEL_ORDER_SRGBX = 13,
    HSA_EXT_BRIG_IMAGE_CHANNEL_ORDER_SRGBA = 14,
    HSA_EXT_BRIG_IMAGE_CHANNEL_ORDER_SBGRA = 15,
    HSA_EXT_BRIG_IMAGE_CHANNEL_ORDER_SBGRA = 16,
    HSA_EXT_BRIG_IMAGE_CHANNEL_ORDER_DEPTH = 17,
    HSA_EXT_BRIG_IMAGE_CHANNEL_ORDER_DEPTH_STENCIL = 18,
    HSA_EXT_BRIG_IMAGE_CHANNEL_ORDER_FIRST_USER_DEFINED = 128
} hsa_ext_brig_image_channel_order_t;

Values 20 through 127 are reserved, but values 128 to 255 are available for implementation defined extensions.

18.3.34 hsa_ext_brig_image_channel_type_t

hsa_ext_brig_image_channel_type_t is used to specify the image channel type. For more information, see 7.1.4.2 Channel type (on page 216).

typedef uint8_t hsa_ext_brig_image_channel_type8_t;
typedef enum {
    HSA_EXT_BRIG_IMAGE_CHANNEL_TYPE_SNORM_INT8 = 0,
    HSA_EXT_BRIG_IMAGE_CHANNEL_TYPE_SNORM_INT8 = 1,
HSA_EXT_BRIG_IMAGE_CHANNEL_TYPE_UNORM_INT8 = 2,
HSA_EXT_BRIG_IMAGE_CHANNEL_TYPE_UNORM_INT16 = 3,
HSA_EXT_BRIG_IMAGE_CHANNEL_TYPE_UNORM_INT24 = 4,
HSA_EXT_BRIG_IMAGE_CHANNEL_TYPE_UNORM_SHORT_555 = 5,
HSA_EXT_BRIG_IMAGE_CHANNEL_TYPE_UNORM_SHORT_565 = 6,
HSA_EXT_BRIG_IMAGE_CHANNEL_TYPE_UNORM_INT_101010 = 7,
HSA_EXT_BRIG_IMAGE_CHANNEL_TYPE_SIGNED_INT8 = 8,
HSA_EXT_BRIG_IMAGE_CHANNEL_TYPE_SIGNED_INT16 = 9,
HSA_EXT_BRIG_IMAGE_CHANNEL_TYPE_SIGNED_INT32 = 10,
HSA_EXT_BRIG_IMAGE_CHANNEL_TYPE_UNSIGNED_INT8 = 11,
HSA_EXT_BRIG_IMAGE_CHANNEL_TYPE_UNSIGNED_INT16 = 12,
HSA_EXT_BRIG_IMAGE_CHANNEL_TYPE_UNSIGNED_INT32 = 13,
HSA_EXT_BRIG_IMAGE_CHANNEL_TYPE_HALF_FLOAT = 14,
HSA_EXT_BRIG_IMAGE_CHANNEL_TYPE_FLOAT = 15,
HSA_EXT_BRIG_IMAGE_CHANNEL_TYPE_FIRST_USER_DEFINED = 128
} hsa_ext brig_image_channel_type_t;

Values 16 through 127 are reserved, but values 128 to 255 are available for implementation defined extensions.

18.3.35 hsa_ext brig_image_geometry_t

hsa_ext brig_image_geometry_t is used to specify the number of coordinates needed to access an image. For more information, see 7.1.3 Image geometry (on page 211).

typedef uint8_t hsa_ext brig_image_geometry8_t;
typedef enum {
    HSA_EXT_BRIG_IMAGE_GEOMETRY_1D = 0,
    HSA_EXT_BRIG_IMAGE_GEOMETRY_2D = 1,
    HSA_EXT_BRIG_IMAGE_GEOMETRY_3D = 2,
    HSA_EXT_BRIG_IMAGE_GEOMETRY_1DA = 3,
    HSA_EXT_BRIG_IMAGE_GEOMETRY_2DA = 4,
    HSA_EXT_BRIG_IMAGE_GEOMETRY_1DB = 5,
    HSA_EXT_BRIG_IMAGE_GEOMETRY_2DB = 6,
    HSA_EXT_BRIG_IMAGE_GEOMETRY_2DDEPTH = 7,
    HSA_EXT_BRIG_IMAGE_GEOMETRY_FIRST_USER_DEFINED = 128
} hsa_ext brig_image_geometry_t;

Values 8 through 127 are reserved, but values 128 to 255 are available for implementation defined extensions.

18.3.36 hsa_ext brig_image_query_t

hsa_ext brig_image_query_t 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 (on page 242).

typedef uint8_t hsa_ext brig_image_query8_t;
typedef enum {
    HSA_EXT_BRIG_IMAGE_QUERY_WIDTH = 0,
    HSA_EXT_BRIG_IMAGE_QUERY_HEIGHT = 1,
    HSA_EXT_BRIG_IMAGE_QUERY_DEPTH = 2,
    HSA_EXT_BRIG_IMAGE_QUERY_ARRAY = 3,
    HSA_EXT_BRIG_IMAGE_QUERY_CHANNELORDER = 4,
    HSA_EXT_BRIG_IMAGE_QUERY_CHANNELTYPE = 5,
    HSA_EXT_BRIG_IMAGE_QUERY_FIRST_USER_DEFINED = 128
} hsa_ext brig_image_query_t;

Values 6 through 127 are reserved, but values 128 to 255 are available for implementation defined extensions.
18.3.37 hsa_ext_brig_sampler_addressing_t

hsa_ext_brig_sampler_addressing_t 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 (on page 223).

typedef uint8_t hsa_ext_brig_sampler_addressing8_t;
typedef enum {
    HSA_EXT_BRIG_SAMPLER_ADDRESSING_UNDEFINED = 0,
    HSA_EXT_BRIG_SAMPLER_ADDRESSING_CLAMP_TO_EDGE = 1,
    HSA_EXT_BRIG_SAMPLER_ADDRESSING_CLAMP_TO_BORDER = 2,
    HSA_EXT_BRIG_SAMPLER_ADDRESSING_REPEAT = 3,
    HSA_EXT_BRIG_SAMPLER_ADDRESSING_MIRRORED_REPEAT = 4,
    HSA_EXT_BRIG_SAMPLER_ADDRESSING_FIRST_USER_DEFINED = 128
} hsa_ext_brig_sampler_addressing_t;

Values 5 through 127 are reserved, but values 128 to 255 are available for implementation defined extensions.

18.3.38 hsa_ext_brig_sampler_coord_normalization_t

hsa_ext_brig_sampler_coord_normalization_t 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 (on page 222).

typedef uint8_t hsa_ext_brig_sampler_coord_normalization8_t;
typedef enum {
    HSA_EXT_BRIG_SAMPLER_COORD_NORMALIZATION_UNNORMALIZED = 0,
    HSA_EXT_BRIG_SAMPLER_COORD_NORMALIZATION_NORMALIZED = 1
} hsa_ext_brig_sampler_coord_normalization_t;

18.3.39 hsa_ext_brig_sampler_filter_t

hsa_ext_brig_sampler_filter_t is used to specify the setting for the filter field in the sampler object. For more information, see 7.1.6.3 Filter mode (on page 225).

typedef uint8_t hsa_ext_brig_sampler_filter8_t;
typedef enum {
    HSA_EXT_BRIG_SAMPLER_FILTER_NEAREST = 0,
    HSA_EXT_BRIG_SAMPLER_FILTER_LINEAR = 1,
    HSA_EXT_BRIG_SAMPLER_FILTER_FIRST_USER_DEFINED = 128
} hsa_ext_brig_sampler_filter_t;

Values 2 through 127 are reserved, but values 128 to 255 are available for implementation defined extensions.

18.3.40 hsa_ext_brig_sampler_query_t

hsa_ext_brig_sampler_query_t 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 (on page 242).

typedef uint8_t hsa_ext_brig_sampler_query8_t;
typedef enum {
    HSA_EXT_BRIG_SAMPLER_QUERY_ADDRESSING = 0,
    HSA_EXT_BRIG_SAMPLER_QUERY_COORD = 1,
    HSA_EXT_BRIG_SAMPLER_QUERY_FILTER = 2,
    HSA_EXT_BRIG_SAMPLER_QUERY_FIRST_USER_DEFINED = 128
} hsa_ext_brig_sampler_query_t;

Values 3 through 127 are reserved, but values 128 to 255 are available for implementation defined extensions.
18.4 hsa_data section

The hsa_data section must start with a hsa Brig section header_t entry. The name of the section must be hsa_data. See 18.3.25 hsa Brig section header_t (on page 351).

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 (on page 340).

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 hsa Brig data_t, not the data, which starts at bytes within hsa Brig data_t.

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 hsa Brig data_t 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 hsa Brig data_t entry, rather than create duplicate hsa Brig data_t entries.

Syntax is:

```c
typedef struct hsa Brig data_s {
    uint32_t byte_count;
    uint8_t bytes[1];
} hsa Brig data_t;
```

Fields are:

- uint32_t byte_count — Number of bytes in the data. Does not include the size byte_count field, or any padding bytes that have to be added to ensure the next hsa Brig data_t starts on a 4-byte boundary. Therefore, to locate the start of the next hsa Brig data_t, the value ((7 + byte_count) / 4) * 4 must be added to the offset of the current hsa Brig data_t.

- uint8_t bytes[1] — Variable-sized. Must be allocated with (((byte_count + 3) / 4) * 4) elements. Any elements after byte_count - 1 must be 0. Bytes 0 to byte_count - 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 `hsa Brig Section Header_t` entry. The name of the section must be `hsa_code`. See 18.3.25 `hsa Brig Section Header_t` (on page 351).

All entries in the `hsa_code` section must start with a `hsa Brig Base_t` structure (see 18.3.6 `hsa Brig Base_t` (on page 342)). The `kind` field of `hsa Brig Base_t` 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 (on page 370)).

The entries for directives and instructions that are part of a kernel or function code block are ordered after a `hsa Brig Directive_Executable_t` entry for the kernel or function, and before the entry referenced by the `next Module Entry Field` of the `hsa Brig Directive_Executable_t` 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 machine code.

The `kind` field of the `hsa Brig Base_t` structure at the start of every `hsa Brig Directive_*` must be in the right-open interval [HSA_BRIG_KIND DIRECTIVE BEGIN, HSA_BRIG_KIND DIRECTIVE END]. See 18.3.12 `hsa Brig Kind_t` (on page 344).

The table below shows the possible formats for the directives. Every directive uses one of these formats.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hsa Brig Directive_Arg Block_t</code></td>
<td>Start and end of an arg block. See 18.5.1.2 <code>hsa Brig Directive_Arg Block_t</code> (on the next page).</td>
</tr>
<tr>
<td><code>hsa Brig Directive_Comment_t</code></td>
<td>Comment string. See 18.5.1.3 <code>hsa Brig Directive_Comment_t</code> (on the next page).</td>
</tr>
<tr>
<td><code>hsa Brig Directive_Control_t</code></td>
<td>Assorted finalizer controls. See 18.5.1.4 <code>hsa Brig Directive_Control_t</code> (on the next page).</td>
</tr>
<tr>
<td><code>hsa Brig Directive_Executable_t</code></td>
<td>Describes a kernel, function, or signature. See 18.5.1.5 <code>hsa Brig Directive_Executable_t</code> (on page 363).</td>
</tr>
<tr>
<td><code>hsa Brig Directive_Extension_t</code></td>
<td>Used to enable extensions that do not specify an explicit version. See 18.5.1.6 <code>hsa Brig Directive_Extension_t</code> (on page 365).</td>
</tr>
<tr>
<td><code>hsa Brig Directive_Extension_Version_t</code></td>
<td>Used to enable extensions that do specify an explicit version. See 18.5.1.7 <code>hsa Brig Directive_Extension_Version_t</code> (on page 365).</td>
</tr>
<tr>
<td><code>hsa Brig Directive_FBarrier_t</code></td>
<td>Used for fbarrier definitions. See 18.5.1.8 <code>hsa Brig Directive_FBarrier_t</code> (on page 365).</td>
</tr>
<tr>
<td><code>hsa Brig Directive_Label_t</code></td>
<td>Declare a label. See 18.5.1.9 <code>hsa Brig Directive_Label_t</code> (on page 366).</td>
</tr>
<tr>
<td><code>hsa Brig Directive_Location_t</code></td>
<td>Source-level line position. See 18.5.1.10 <code>hsa Brig Directive_Location_t</code> (on page 366).</td>
</tr>
<tr>
<td><code>hsa Brig Directive_Module_t</code></td>
<td>Module name, HSAIL version, and target information. See 18.5.1.11 <code>hsa Brig Directive_Module_t</code> (on page 367).</td>
</tr>
<tr>
<td><code>hsa Brig Directive_None_t</code></td>
<td>Special directive that is always ignored. See 18.5.1.12 <code>hsa Brig Directive_None_t</code> (on page 368).</td>
</tr>
</tbody>
</table>
### 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 `hsa_brig_directive_arg_block_t`

`hsa_brig_directive_arg_block_t` specifies the start and end of an arg block. See 4.3.6 Arg block (on page 66).

Syntax is:
```c
typedef struct hsa_brigDirective_argBlock_s {
    hsa_brig_base_t base;
} hsa_brigDirective_argBlock_t;
```

Fields are:
- `hsa_brig_base_t base` — `base.kind` must be `HSA_BRIG_KIND_DIRECTIVE_ARG_BLOCK_END` or `HSA_BRIG_KIND_DIRECTIVE_ARG_BLOCK_START`.

### 18.5.1.3 `hsa_brig_directive_comment_t`

`hsa_brig_directive_comment_t` is a comment string.

Syntax is:
```c
typedef struct hsa_brigDirective_comment_s {
    hsa_brig_base_t base;
    hsa_brig_data_offset_string32_t name;
} hsa_brigDirective_comment_t;
```

Fields are:
- `hsa_brig_base_t base` — `base.kind` must be `HSA_BRIG_KIND_DIRECTIVE_COMMENT`.
- `hsa_brig_data_offset_string32_t name` — Byte offset of the entry in the `hsa_dataSection` that contains the text of the comment (including the `//`).

### 18.5.1.4 `hsa_brig_directive_control_t`

`hsa_brig_directive_control_t` 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 13.5 Control directives for low-level performance tuning (on page 317).

Syntax is:
```c
typedef struct hsa_brigDirective_control_s {
    hsa_brig_base_t base;
    hsa_brig_controlDirective16_t control;
}
```
uint16_t reserved;
hsa_brig_data_offset_operand_list32_t operands;
} hsa_brigDirective_control_t;

Fields are:

- `hsa_brig_base_t base` — `base.kind` must be `HSA_BRIG_KIND_DIRECTIVE_CONTROL`.
- `hsa_brig_control_directive16_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 `hsa_brig_control_directive_t` (on page 343).
- `uint16_t reserved` — Must be 0.
- `hsa_brig_data_offset_operand_list32_t operands` — Byte offset of the entry in the `hsa_data` section that contains a variable-sized array of byte offsets to operands in the `hsa_operand` section. The operands must either be `HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES` or `HSA_BRIG_KIND_OPERAND_WAVESIZE`.

18.5.1.5 `hsa_brig_directive_executable_t`

`hsa_brig_directive_executable_t` describes a kernel, function, or signature.

Kernels are arranged in the `hsa_code` section as (see 4.3.2 Kernel (on page 60)):

1. `hsa_brig_directive_executable_t` with kind of `HSA_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 (on page 61) and 10.3 Function declarations, function definitions, and function signatures (on page 266)):

1. `hsa_brig_directive_executable_t` with kind of `HSA_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 (on page 267)):

1. `hsa_brig_directive_executable_t` with kind of `HSA_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 `hsa Brig Directive Variable_t` with a segment field of `HSA_BRIG_SEGMENT_KERNARG` for kernels; and `HSA_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:

```c
typedef struct hsa Brig Directive Executable_s {
    hsa Brig Base_t base;
    hsa Brig Data Offset String32_t name;
    uint16_t out_arg_count;
    uint16_t in_arg_count;
    hsa Brig Code Offset32_t first_in_arg;
    hsa Brig Code Offset32_t first_code_block_entry;
    hsa Brig Code Offset32_t next_module_entry;
    hsa Brig Executable Modifier8_t modifier;
    hsa Brig Linkage8_t linkage;
    uint16_t reserved;
} hsa Brig Directive Executable_t;
```

Fields are:

- `hsa Brig Base_t base` — `base.kind` must be `HSA_BRIG_KIND_DIRECTIVE_KERNEL`, `HSA_BRIG_KIND_DIRECTIVE_FUNCTION` or `HSA_BRIG_KIND_DIRECTIVE_SIGNATURE`.
- `hsa Brig Data Offset String32_t name` — Byte offset of the entry in the `hsa data` section that contains the name of the kernel, function, or signature.
- `uint16_t out_arg_count` — The number of output parameters from the function or signature. Must be 0 for kernels.
- `uint16_t in_arg_count` — The number of input formal arguments to the kernel, function, or signature.
- `hsa Brig Code Offset32_t first_in_arg` — 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 `first_code_block_entry`.
- `hsa Brig Code Offset32_t first_code_block_entry` — 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 `HSA_BRIG_EXECUTEABLE_MODIFIER_DEFINITION` of zero), or if the kernel or function definition code block has no entries, then this must be the same value as `next_module_entry`.
- `hsa Brig Code Offset32_t next_module_entry` — 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.
- `hsa Brig Executable Modifier8_t modifier` — Modifier for the kernel, function, or signature. The `HSA_BRIG_EXECUTEABLE_MODIFIER_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 `hsa Brig Executable Modifier_t` (on page 343).
- `hsa Brig Linkage8_t linkage` — Values are specified by the `hsa Brig Linkage_t` enumeration. Must be `HSA_BRIG_LINKAGE_NONE` for signatures; and `HSA_BRIG_LINKAGE_PROGRAM` or `HSA_BRIG_LINKAGE_MODULE` for kernels or functions depending on the linkage specified. See 18.3.13 `hsa Brig Linkage_t` (on page 345).
- `uint16_t reserved` — Must be 0.
18.5.1.6 hsa Brig_directive_extension_t

hsa Brig_directive_extension_t is used to enable an extension without specifying the version. The version used defaults to the HSAIL version specified in the module directive. Must be used for the "CORE" extension, which is not allowed to have a version. For more information, see 13.1 extension directive (on page 293).

Syntax is:

typedef struct hsa Brig_directive_extension_s {
    hsa Brig_base_t base;
    hsa Brig_data_offset_string32_t name;
} hsa Brig_directive_extension_t;

Fields are:

- hsa Brig_base_t base — base.kind must be HSA BRIG KIND DIRECTIVE EXTENSION.
- hsa Brig_data_offset_string32_t name — Byte offset of the entry in the hsa_data section that contains the name of the extension.

18.5.1.7 hsa Brig_directive_extension_version_t

hsa Brig_directive_extension_version_t is used to enable an extension with an explicit version. Must not be used for the "CORE" extension, which is not allowed to have a version; use HSA BRIG KIND DIRECTIVE_EXTENSION instead. For more information, see 13.1 extension directive (on page 293).

Syntax is:

typedef struct hsa Brig_directive_extension_version_s {
    hsa Brig_base_t base;
    hsa Brig_data_offset_string32_t name;
    hsa Brig_version32_t extension_major;
    hsa Brig_version32_t extension_minor;
} hsa Brig_directive_extension_version_t;

Fields are:

- hsa Brig_base_t base — base.kind must be HSA BRIG KIND DIRECTIVE_EXTENSION_VERSION.
- hsa Brig_data_offset_string32_t name — Byte offset of the entry in the hsa_data section that contains the name of the extension.
- hsa Brig_version32_t extension_major — The extension major version. See 18.3.31 hsa Brig_version_t (on page 355).
- hsa Brig_version32_t extension_minor — The extension minor version. See 18.3.31 hsa Brig_version_t (on page 355).

18.5.1.8 hsa Brig_directive_fbarrier_t

hsa Brig_directive_fbarrier_t is used for fbarrier declarations and definitions.

Syntax is:

typedef struct hsa Brig_directive_fbarrier_s {
    hsa Brig_base_t base;
    hsa Brig_data_offset_string32_t name;
hsa brig variable modifier8_t modifier;
hsa brig linkage8_t linkage;
uint16_t reserved;
} hsa brig directive fbarrier_t;

Fields are:

- hsa brig base t base — base.kind must be HSA_BRIG_KIND_DIRECTIVE_FBARRIER.
- hsa brig data offset string32 t name — Byte offset of the entry in the hsa data
  section that contains the name of the fbarrier.
- hsa brig variable modifier8 t modifier — Modifier for the fbarrier. The HSA_BRIG
  VARIABLE_MODIFIER_DEFINITION must be 0 if a declaration; and 1 if a definition. The values
  for other bitmask fields must be 0. See 18.3.30 hsa brig variable modifier t (on page 355).
- hsa brig linkage8_t linkage — Values are specified by the hsa brig linkage t
  enumeration. For module scope fbarriers must be HSA_BRIG_LINKAGE_PROGRAM or HSA_ 
  BRIG_LINKAGE_MODULE depending on the linkage specified; and for function scope fbarriers
  must be HSA_BRIG_LINKAGE_FUNCTION. See 4.6.2 Scope (on page 82) and 18.3.13 hsabrig_ 
  linkage_t (on page 345)
- uint16_t reserved — Must be 0.

18.5.1.9 hsa brig directive label t

hsa brig directive label t 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:
typedef struct hsa brig directive label s {
  hsa brig base t base;
  hsa brig data offset string32 t name;
} hsa brig directive label_t;

Fields are:

- hsa brig base t base — base.kind must be HSA_BRIG_KIND_DIRECTIVE_LABEL.
- hsa brig data offset string32 t name — Byte offset of the entry in the hsa data
  section that contains the name of the label.

18.5.1.10 hsa brig directive loc t

hsa brig directive loc t specifies the source-level line position. The entries starting at next entry
until the next hsa brig directive loc t 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 (on page 296).

Syntax is:
typedef struct hsa brig directive loc s {
  hsa brig base t base;
  hsa brig data offset string32 t filename;
  uint32_t line;
  uint32_t column;
} hsa brig directive loc_t;
Fields are:

- `hsa brig base t base — base.kind must be HSA BRIG KIND DIRECTIVE LOC.
- `hsa brig data offset string32 t filename — Byte offset of the entry in the hsa data section that contains the name of the file. If the HSA loc directive did not specify a file name then must reference an entry for the same string used in the nearest preceding loc directive within the module that does specify a file name, or an entry for 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 `hsa Brig Directive Loc t 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 `hsa Brig Directive Loc t statements can refer to the same column.

### 18.5.1.11 `hsa Brig Directive Module t`

`hsa Brig Directive Module t` specifies the module name, HSAIL virtual ISA specification version, and target information. For more information, see Chapter 14 module Header (on page 323).

There must be exactly one `hsa Brig Directive Module t` directive in the `hsa code section`. It may be optionally preceded only by `hsa Brig Directive Comment t, hsa Brig Directive Loc t and hsa Brig Directive Pragma t directives.

Syntax is:

```c
typedef struct hsa Brig Directive Module s {
    hsa Brig Base t base;
    hsa Brig Data Offset String32 t name;
    hsa Brig Version32 t hsa il major;
    hsa Brig Version32 t hsa il minor;
    hsa Brig Profile8 t profile;
    hsa Brig Machine Model8 t machine model;
    hsa Brig Round8 t default float round;
    uint8 t reserved;
} hsa Brig Directive Module t;
```

Fields are:

- `hsa Brig Base t base — base.kind must be HSA BRIG KIND DIRECTIVE MODULE.
- `hsa Brig Data Offset String32 t name — Byte offset of the entry in the hsa data section that contains the name of the module.
- `hsa Brig Version32 t hsa il major — The HSAIL major version. When generating BRIG, must be HSA BRIG VERSION HSAIL MAJOR. When consuming BRIG, must be HSA BRIG VERSION HSAIL MAJOR to be compatible with this revision of the HSAIL virtual ISA specification. See 18.3.31 `hsa Brig Version t (on page 355).
- `hsa Brig Version32 t hsa il minor — The HSAIL minor version. When generating BRIG, must be HSA BRIG VERSION HSAIL MINOR. When consuming BRIG, hsa il major must be HSA BRIG VERSION HSAIL MAJOR and hsa il minor must be less than or equal to HSA BRIG VERSION HSAIL MINOR to be compatible with this revision of the HSAIL virtual ISA specification. See 18.3.31 `hsa Brig Version t (on page 355).
- `hsa Brig profile8_t profile` - The profile. A member of the `hsa Brig profile_t` enumeration. See 18.3.21 `hsa Brig profile_t` (on page 350).

- `hsa Brig machine_model8_t machine_model` - The machine model. A member of the `hsa Brig machine_model_t` enumeration. See 18.3.14 `hsa Brig machine_model_t` (on page 345).

- `hsa Brig round8_t default_float_round` - The default floating-point rounding mode. A member of the `hsa Brig round_t` enumeration: only `HSABRIG_ROUND_FLOAT_DEFAULT, HSBRIG_ROUND_FLOAT_NEAREST, HSBRIG_ROUND_FLOAT_ZERO` are allowed. See 18.3.23 `hsa Brig round_t` (on page 350).

- `uint8_t reserved;` — Must be 0.

### 18.5.1.12 `hsa Brig directive none_t`

The `hsa Brig directive none_t` 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 `hsa Brig directive none_t` format. For more information, see 13.3 pad directive (on page 296).

`hsa Brig directive none_t` 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 `hsa Brig directive none_t` structure must be set to 0.

Syntax:

```c
typedef struct hsa Brig directive none_s {
    hsa Brig base_t base;
} hsa Brig directive none_t;
```

Fields are:

- `hsa Brig base_t base` — `base.kind` must be `HSABRIG_KIND_NONE` (which has the value 0). `base.size` must be a multiple of 4. If `size` is greater than the size of the `hsa Brig directive none_t` structure (4 bytes), then any extra bytes must be set to 0. `size` corresponds to the operand of the corresponding `pad` directive: `(operand + 1) * 4`.

### 18.5.1.13 `hsa Brig directive pragma_t`

`hsa Brig directive pragma_t` allows additional information to be given to control the finalizer and other consumers of HSAIL. For more information, see 13.4 pragma directive (on page 297).

Syntax is:

```c
typedef struct hsa Brig directive pragma_s {
    hsa Brig base_t base;
    hsa Brig data_offset_operand_list32_t operands;
} hsa Brig directive pragma_t;
```

Fields are:

- `hsa Brig base_t base` — `base.kind` must be `HSABRIG_KIND_DIRECTIVE_PRAGMA`.

- `hsa Brig data_offset_operand_list32_t operands` — Byte offset of the entry in the `hsa data` section that contains a variable-sized array of byte offsets to operands in the `hsa operand` section. The byte count of the entry must be exactly `4 * number of operands`. The operands must either be `HSABRIG_KINDOPERAND_CONSTANTBYTES, HSBKINDOPERAND_CONSTANTEXPRESSION, HSBKINDOPERAND...`
CONSTANT_OPERAND_LIST, HSA_BRIG_KIND_OPERAND_CONSTANT_IMAGE, HSA_BRIG_KIND_OPERAND_CONSTANT_SAMPLER, HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CODE_REF, HSA_BRIG_KIND_OPERAND_STRING, or HSA_BRIG_KIND_OPERAND_WAVESIZE.

If any operand is a constant, it must be compatible with the rules in 18.6.1 Constant operands (on page 382).

### 18.5.1.14 hsa_brig_directive_variable_t

hsa_brig_directive_variable_t is used for variable declarations or definitions.

Syntax is:

```c
typedef struct hsa_brig_directive_variable_s {
    hsa_brig_base_t base;
    hsa_brig_data_offset_string32_t name;
    hsa_brig_operand_offset32_t init;
    hsa_brig_type16_t type;
    hsa_brig_segment8_t segment;
    hsa_brig_alignment8_t align;
    hsa_brig_uint64_t dim;
    hsa_brig_variable_modifier8_t modifier;
    hsa_brig_linkage8_t linkage;
    hsa_brig_allocation8_t allocation;
    uint8_t reserved;
} hsa_brig_directive_variable_t;
```

Fields are:

- `hsa_brig_base_t base` — base.kind must be HSA_BRIG_KIND DIRECTIVE_VARIABLE.
- `hsa_brig_data_offset_string32_t name` — Byte offset of the entry in the hsa_data section that contains the variable name.
- `hsa_brig_operand_offset32_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 of HSA_BRIG_KIND OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_LIST, HSA_BRIG_KIND_OPERAND_CONSTANT_IMAGE, or HSA_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 (on page 382).

- `hsa_brig_type16_t type` — The BRIG type of the variable. If the variable is an array then must be an array type. HSA_BRIG_TYPE_B1 is not allowed.
- `hsa_brig_segment8_t segment` — Segment that will hold the variable. A member of the hsa_brig_segment_t enumeration. See 18.3.27 hsa_brig_segment_t (on page 352).
- `hsa_brig_alignment8_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 hsa_brig_alignment_t (on page 340).
- `hsa_brig_uint64_t dim` — The array dimension size dim. See 18.3.29 hsa_brig_uint64_t (on page 355).
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 (on page 267)) or a declaration with no size specified, and `dim` must be set to 0.

- **hsa Brig variable modifier8_t modifier** — Modifier for the variable. See 18.3.30 hsa Brig variable modifier8_t (on page 355).

- **hsa Brig linkage8_t linkage** — Values are specified by the hsa Brig linkage8_t enumeration. For module scope variables must be HSA_Brig_LINKAGE_PROGRAM or HSA_Brig_LINKAGE_MODULE depending on the linkage specified; for function scope variables must be HSA_Brig_LINKAGE_FUNCTION; for argument scope variables must be HSA_Brig_LINKAGE_ARG; and for signature scope variables must be HSA_Brig_LINKAGE_NONE. See 4.6.2 Scope (on page 82) and 18.3.13 hsa Brig linkage8_t (on page 345).

- **hsa Brig allocation8_t allocation** — Values are specified by the hsa Brig allocation8_t enumeration. For global segment variable must be HSA_Brig_ALLOCATION_PROGRAM or HSA_Brig_ALLOCATION_AGENT depending on the allocation specified; for readonly segment variable must be HSA_Brig_ALLOCATION_AGENT; otherwise must be HSA_Brig_ALLOCATION_AUTOMATIC. See 18.3.3 hsa Brig allocation8_t (on page 340).

- **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 machine code for kernels and indirect functions.

Every `hsa Brig inst_*` must start with a `hsa Brig inst base_t`. See 18.5.2.1 hsa Brig inst base_t (on the facing page).

The table below shows the possible formats for the instructions. Every instruction uses one of these formats.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hsa Brig inst base t</td>
<td>Every other hsa Brig inst_* entry must start with this structure. See 18.5.2.1 hsa Brig inst base t (on the facing page).</td>
</tr>
<tr>
<td>hsa Brig inst addr t</td>
<td>Address instructions. See 18.5.2.2 hsa Brig inst addr t (on page 372).</td>
</tr>
<tr>
<td>hsa Brig inst atomic t</td>
<td>Atomic instructions. See 18.5.2.3 hsa Brig inst atomic t (on page 372).</td>
</tr>
<tr>
<td>hsa Brig inst basic t</td>
<td>Used for all instructions that require no extra modifier information. See 18.5.2.4 hsa Brig inst basic t (on page 373).</td>
</tr>
<tr>
<td>hsa Brig inst br t</td>
<td>Branch, call, barrier and fbarrier instructions. See 18.5.2.5 hsa Brig inst br t (on page 373).</td>
</tr>
<tr>
<td>hsa Brig inst cmp t</td>
<td>Compare instruction. See 18.5.2.6 hsa Brig inst cmp t (on page 374).</td>
</tr>
<tr>
<td>hsa Brig inst cvt t</td>
<td>Conversion instruction. See 18.5.2.7 hsa Brig inst cvt t (on page 374).</td>
</tr>
<tr>
<td>hsa Brig inst lane t</td>
<td>Cross lane instructions. See 18.5.2.8 hsa Brig inst lane t (on page 375).</td>
</tr>
</tbody>
</table>
18.5.2.1 hsa_brig_inst_base_t

Every other hsa_brig_inst_ * must start with hsa_brig_inst_base_t which in turn starts with hsa_brig_base_t.

Syntax is:

typedef struct hsa_brig_inst_base_s {
    hsa_brig_base_t base;
    hsa_brig_opcode16_t opcode;
    hsa_brig_type16_t type;
    hsa_brig_data_offset_operand_list32_t operands;
} hsa_brig_inst_base_t;

Fields are:

- hsa_brig_base_t base — The base.kind field must be in the right-open interval [HSA_BRIG_KIND_INST_BEGIN, HSA_BRIG_KIND_INST_END). See 18.3.12 hsa_brig_kind_t (on page 344).

- hsa_brig_opcode16_t opcode — Opcode associated with the instruction.

- hsa_brig_type16_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 source_type 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 HSA_BRIG_TYPE_NONE must be used.

- hsa_brig_data_offset_operand_list32_t operands — Byte offset of the entry in the hsa_data section that contains a variable-sized array of byte offsets to operands in the hsa_operand section. The byte_count of the entry must be exactly (4 * 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 (on page 382). The operand kinds allowed for each opcode value are defined in 18.7 BRIG syntax for instructions (on page 392).

18.5.2.2 hsa_brig_inst_addr_t

The hsa_brig_inst_addr_t format is used for address instructions.

Syntax is:

typedef struct hsa_brig_inst_addr_s {
    hsa_brig_inst_base_t base;
    hsa_brig_segment8_t segment;
    uint8_t reserved[3];
} hsa_brig_inst_addr_t;

Fields are:

- hsa_brig_inst_base_t base — base.base.kind must be HSA_BRIG_KIND_INST_ADDR. base.type must be the data type of the destination and source of the instruction.

- hsa_brig_segment8_t segment — Segment. A member of the hsa_brig_segment_t enumeration. If the instruction does not specify a segment, this field must be set to HSA_BRIG_SEGMENT_FLAT. See 18.3.27 hsa_brig_segment_t (on page 352).

  - uint8_t reserved[3] — Must be 0.

18.5.2.3 hsa_brig_inst_atomic_t

The hsa_brig_inst_atomic_t format is used for atomic and atomic no return instructions.

Syntax is:

typedef struct hsa_brig_inst_atomic_s {
    hsa_brig_inst_base_t base;
    hsa_brig_segment8_t segment;
    hsa_brig_memory_order8_t memory_order;
    hsa_brig_memory_scope8_t memory_scope;
    hsa_brig_atomic_operation8_t atomic_operation;
    uint8_t equiv_class;
    uint8_t reserved[3];
} hsa_brig_inst_atomic_t;

Fields are:

- hsa_brig_inst_base_t base — base.base.kind must be HSA_BRIG_KIND_INST_ATOMIC. base.opcode must be HSA_BRIG_OPCODE_ATOMIC or HSA_BRIG_OPCODE_ATOMICNORET. base.type must be the data type of the destination and source of the atomic instruction.

- hsa_brig_segment8_t segment — Segment. A member of the hsa_brig_segment_t enumeration. If the instruction does not specify a segment, this field must be set to HSA_BRIG_SEGMENT_FLAT. Otherwise must be HSA_BRIG_SEGMENT_GLOBAL or HSA_BRIG_SEGMENT_GROUP. See 18.3.27 hsa_brig_segment_t (on page 352).

  - hsa_brig_memory_order8_t memory_order — Memory order of the atomic instruction. See 18.3.16 hsa_brig_memory_order_t (on page 345).

  - hsa_brig_memory_scope8_t memory_scope — Memory scope of the atomic instruction. If segment is HSA_BRIG_SEGMENT_GLOBAL or HSA_BRIG_SEGMENT_FLAT then must be HSA_
BRIG_MEMORY_SCOPE_WAVEFRONT, HSA_BRIG_MEMORY_SCOPE_WORKGROUP, HSA_BRIG_MEMORY_SCOPE_AGENT, or HSA_BRIG_MEMORY_SCOPE_SYSTEM. If segment is HSA_BRIG_SEGMENT_GROUP then must be HSA_BRIG_MEMORY_SCOPE_WAVEFRONT or HSA_BRIG_MEMORY_SCOPE_WORKGROUP. See 18.3.17 hsa_brig_memory_scope_t (on page 346).

- hsa_brig_atomic_operation8_t atomic_operation — The atomic instruction such as add or or. The wait atomic instructions are not allowed. See 18.3.5 hsa_brig_atomic_operation_t (on page 341).

- uint8_t equiv_class — 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 (on page 182).

- uint8_t reserved[3] — Must be 0.

18.5.2.4 hsa_brig_inst_basic_t

The hsa_brig_inst_basic_t format is used for all instructions that require no extra modifier information.

Syntax is:

```c
typedef struct hsa_brig_inst_basic_s {
   hsa_brig_inst_base_t base;
} hsa_brig_inst_basic_t;
```

Fields are:

- hsa_brig_inst_base_t base — base.base.kind must be HSA_BRIG_KIND_INST_BASIC. base.type must be the data type of the destination and source of the instruction.

18.5.2.5 hsa_brig_inst_br_t

The hsa_brig_inst_br_t format is used for the branch, call, barrier and fbarrier instructions.

Syntax is:

```c
typedef struct hsa_brig_inst_br_s {
   hsa_brig_inst_base_t base;
   hsa_brig_width8_t width;
   uint8_t reserved[3];
} hsa_brig_inst_br_t;
```

Fields are:

- hsa_brig_inst_base_t base — base.base.kind must be HSA_BRIG_KIND_INST_BR. base.opcode must be HSA_BRIG_OPCODE_BR, HSA_BRIG_OPCODE_CBR, HSA_BRIG_OPCODE_SBR, HSA_BRIG_OPCODE_CALL, HSA_BRIG_OPCODESCALL, HSA_BRIG_OPCODE_ICALL, HSA_BRIG_OPCODE_BARRIER, HSA_BRIG_OPCODE_WAVEBARRIER, HSA_BRIG_OPCODE_ARRIVEFBAR, HSA_BRIG_OPCODE_JOINFBAR, HSA_BRIG_OPCODE_LEAVEFBAR or HSA_BRIG_OPCODE_WAITFBAR. base.type must be the source operand data type, or HSA_BRIG_TYPE_NONE if the instruction has no typed operands.

- hsa_brig_width8_t width — The width modifier. If the instruction does not support the width modifier, then this must be HSA_BRIG_WIDTH_ALL for the direct branch and direct call instructions, and HSA_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 `HSA_BRIG_WIDTH_1`; for barrier instructions, it is `HSA_BRIG_WIDTH_ALL`; and for the `fbarrier` instructions, it is `HSA_BRIG_WIDTH_WAVESIZE`. Otherwise, this must be the value from `hsa brig width_t` that corresponds to the specified width modifier. See 18.3.32 `hsa brig width_t` (on page 356).

- `uint8_t reserved[3]` — Must be 0.

18.5.2.6 `hsa brig inst cmp t`

The `hsa brig inst cmp t` 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:

```c
typedef struct hsa brig inst cmp s {  
  hsa brig inst base_t base;  
  hsa brig type16_t source_type;  
  hsa brig alu modifier8_t modifier;  
  hsa brig compare operation8_t compare;  
  hsa brig pack8_t pack;  
  uint8_t reserved[3];  
} hsa brig inst cmp t;
```

Fields are:

- `hsa brig inst base_t base` — `base.base.kind` must be `HSA_BRIG_KIND_INST_CMP`. `base.opcode` must be `HSA_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 `source_type`.
- `hsa brig type16_t source_type` — Type of the sources.
- `hsa brig alu modifier8_t modifier` — The modifier flags for this instruction. See 18.3.4 `hsa brig alu modifier t` (on page 341).
- `hsa brig compare operation8_t compare` — The specific comparison (greater than, less than, and so forth).
- `hsa brig pack8_t pack` — Packing control. See 18.3.20 `hsa brig pack t` (on page 349).
- `uint8_t reserved[3]` — Must be 0.

18.5.2.7 `hsa brig inst cvt t`

The `hsa brig inst cvt t` format is used for conversion instructions.

Syntax is:

```c
typedef struct hsa brig inst cvt s {  
  hsa brig inst base_t base;  
  hsa brig type16_t source_type;  
  hsa brig alu modifier8_t modifier;  
  hsa brig round8_t round;  
} hsa brig inst cvt t;
```

Fields are:

- `hsa brig inst base_t base` — `base.base.kind` must be `HSA_BRIG_KIND_INST_CVT`. `base.opcode` must be `HSA_BRIG_OPCODE_CVT`. `base.type` must be the data type of the destination of the conversion instruction.
• `hsa_brig_type16_t source_type` — Type of the sources.

• `hsa_brig_alu_modifier8_t modifier` — The modifier flags for this instruction. See 18.3.4 `hsa_brig_alu_modifier_t` (on page 341).

• `hsa_brig_round8_t round` — Rounding mode. See 18.3.23 `hsa_brig_round_t` (on page 350).

18.5.2.8 `hsa_brig_inst_lane_t`

The `hsa_brig_inst_lane_t` format is used for cross-lane instructions.

Syntax is:

```c
typedef struct hsa_brig_inst_lane_s {
  hsa_brig_inst_base_t base;
  hsa_brig_type16_t source_type;
  hsa_brig_width8_t width;
  uint8_t reserved;
} hsa_brig_inst_lane_t;
```

Fields are:

• `hsa_brig_inst_base_t base` — `base.base.kind` must be `HSA_BRIG_KIND_INST_LANE`. `base.type` must be the data type of the destination of the cross-lane instruction.

• `hsa_brig_type16_t source_type` — Type of the source. If the instruction does not have a source type modifier then must be `HSA_BRIG_TYPE_NONE`.

• `hsa_brig_width8_t width` — The width modifier. If the instruction does not specify the width modifier, then this must be `HSA_BRIG_WIDTH_1` (the default for the cross-lane instructions). Otherwise, this must be the value from `hsa_brig_width_t` that corresponds to the specified width modifier. See 18.3.32 `hsa_brig_width_t` (on page 356).

• `uint16_t reserved` — Must be 0.

18.5.2.9 `hsa_brig_inst_mem_t`

The `hsa_brig_inst_mem_t` format is used for memory instructions.

Syntax is:

```c
typedef struct hsa_brig_inst_mem_s {
  hsa_brig_inst_base_t base;
  hsa_brig_segment8_t segment;
  hsa_brig_alignment8_t align;
  uint8_t equiv_class;
  hsa_brig_width8_t width;
  hsa_brig_memory_modifier8_t modifier;
  uint8_t reserved[3];
} hsa_brig_inst_mem_t;
```

Fields are:

• `hsa_brig_inst_base_t base` — `base.base.kind` must be `HSA_BRIG_KIND_INST_MEM`. `base.type` must be the data type of the destination and source of the memory instruction.

• `hsa_brig_segment8_t segment` — Segment. A member of the `hsa_brig_segment_t` enumeration. If the instruction does not support the segment modifier, then this must be `HSA_BRIG_SEGMENT_NONE`. If the instruction supports the segment modifier but does not specify it, then this must be `HSA_BRIG_SEGMENT_FLAT`. See 18.3.27 `hsa_brig_segment_t` (on page 352).
• hsa_brig_alignment8_t align — The align modifier. If the instruction does not specify the align modifier, then this must be HSA_BRIG_ALIGNMENT_1 (the default for memory instructions). Otherwise, this must be the value from hsa_brig_alignment_t that corresponds to the specified align modifier. See 18.3.2 hsa_brig_alignment_t (on page 340).

• uint8_t equiv_class — 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 (on page 182).

• hsa_brig_width8_t width — The width modifier. If the instruction does not support the width modifier, then this must be HSA_BRIG_WIDTH_NONE. If the instruction supports the width modifier but does not specify it, then this must be HSA_BRIG_WIDTH_1 (the default for memory instructions). Otherwise, this must be the value from hsa_brig_width_t that corresponds to the specified width modifier. See 18.3.32 hsa_brig_width_t (on page 356).

• hsa_brig_memory_modifier8_t modifier — Memory modifier flags of the instruction. See 18.3.15 hsa_brig_memory_modifier_t (on page 345).

• uint8_t reserved[3] — Must be 0.

18.5.2.10 hsa_brig_inst_mem_fence_t

The hsa_brig_inst_mem_fence_t format is used for the memfence instruction.

Syntax is:

```c
typedef struct hsa_brig_inst_mem_fence_t {
    hsa_brig_inst_base_t base;
    hsa_brig_memory_order8_t memory_order;
    hsa_brig_memory_scope8_t global_segment_memory_scope;
    hsa_brig_memory_scope8_t group_segment_memory_scope;
    uint8_t reserved;
} hsa_brig_inst_mem_fence_t;
```

Fields are:

• hsa_brig_inst_base_t base — base.base.kind must be HSA_BRIG_KIND_INST_MEM_FENCE, base.opcode must be HSA_BRIG_OPCODE_MEMFENCE, base.type must be HSA_BRIG_TYPE_NONE as a memory fence instruction has no destination operand.

• hsa_brig_memory_order8_t memory_order — Memory order of the memory fence instruction. Must be HSA_BRIG_MEMORY_ORDER_SC_ACQUIRE, HSA_BRIG_MEMORY_ORDER_SC_RELEASE, or HSA_BRIG_MEMORY_ORDER_SC_ACQUIRE_RELEASE. See 18.3.16 hsa_brig_memory_order_t (on page 345).

• hsa_brig_memory_scope8_t global_segment_memory_scope — Memory scope for the global segment of the memory fence instruction. Must be HSA_BRIG_MEMORY_SCOPE_WAVEFRONT, HSA_BRIG_MEMORY_SCOPE_WORKGROUP, HSA_BRIG_MEMORY_SCOPE_AGENT, or HSA_BRIG_MEMORY_SCOPE_SYSTEM. See 18.3.17 hsa_brig_memory_scope_t (on page 346).

• hsa_brig_memory_scope8_t group_segment_memory_scope — Memory scope for the group segment of the memory fence instruction. Must be the same value as global_segment_memory_scope as the memory orders currently supported by memfence synchronize with both the group and global segment. See 18.3.17 hsa_brig_memory_scope_t (on page 346).

• uint8_t reserved — Must be 0.
18.5.2.11 hsa\_brig\_inst\_mod\_t

The hsa\_brig\_inst\_mod\_t format is used for ALU instructions with a modifier.

Syntax is:

```c
typedef struct hsa\_brig\_inst\_mod\_s {
    hsa\_brig\_inst\_base\_t base;
    hsa\_brig\_alu\_modifier8\_t modifier;
    hsa\_brig\_round8\_t round;
    hsa\_brig\_pack8\_t pack;
    uint8\_t reserved;
} hsa\_brig\_inst\_mod\_t;
```

Fields are:

- **hsa\_brig\_inst\_base\_t** base — base.base.kind must be HSA\_BRIG\_KIND\_INST\_MOD. base.type must be the data type of the destination of the instruction.
- **hsa\_brig\_alu\_modifier8\_t** modifier — The modifier flags for this instruction. See 18.3.4 hsa\_brig\_alu\_modifier\_t (on page 341).
- **hsa\_brig\_round8\_t** round — Rounding mode. See 18.3.23 hsa\_brig\_round\_t (on page 350).
- **hsa\_brig\_pack8\_t** pack — Packing control. If the instruction does not have a packing modifier, this must be set to HSA\_BRIG\_PACK\_NONE. See 18.3.20 hsa\_brig\_pack\_t (on page 349).
- **uint8\_t** reserved — Must be 0.

18.5.2.12 hsa\_brig\_inst\_queue\_t

The hsa\_brig\_inst\_queue\_t format is used for user mode queue instructions.

Syntax is:

```c
typedef struct hsa\_brig\_inst\_queue\_s {
    hsa\_brig\_inst\_base\_t base;
    hsa\_brig\_segment8\_t segment;
    hsa\_brig\_memory\_order8\_t memory\_order;
    uint16\_t reserved;
} hsa\_brig\_inst\_queue\_t;
```

Fields are:

- **hsa\_brig\_inst\_base\_t** base — base.base.kind must be HSA\_BRIG\_KIND\_INST\_QUEUE. base.type must be the data type of the destination of the user mode queue instruction.
- **hsa\_brig\_segment8\_t** segment — Segment. A member of the hsa\_brig\_segment\_t enumeration. If the instruction does not specify a segment, this field must be set to HSA\_BRIG\_SEGMENT\_FLAT. See 18.3.27 hsa\_brig\_segment\_t (on page 352).
- **hsa\_brig\_memory\_order8\_t** memory\_order — Memory order of the user mode queue instruction. See 18.3.16 hsa\_brig\_memory\_order\_t (on page 345).
- **uint16\_t** reserved — Must be 0.

18.5.2.13 hsa\_brig\_inst\_seg\_t

The hsa\_brig\_inst\_seg\_t format is used for instructions with memory segments.
Chapter 18. BRIG: HSAIL Binary Format  18.5 hsa_code section

Syntax is:

typedef struct hsa brig inst seg_s {
    hsa brig base_t base;
    hsa brig segment8_t segment;
    uint8_t reserved[3];
} hsa brig inst seg_t;

Fields are:

- hsa brig base_t base — base.base.kind must be HSA_BRIG_KIND_INST_SEG.
  base.type must be the data type of the destination of the instruction.

- hsa brig segment8_t segment — Segment. A member of the hsa brig segment_t
  enumeration. If the instruction does not specify a segment, this field
  must be set to HSA_BRIG_SEGMENT_FLAT. See 18.3.27 hsa brig segment_t (on
  page 352).

- uint8_t reserved[3] — Must be 0.

18.5.2.14 hsa brig inst seg cvt_t

The hsa brig inst seg cvt_t format is used for instructions which convert between
segment and flat addresses.

Syntax is:

typedef struct hsa brig inst seg cvt_s {
    hsa brig base_t base;
    hsa brig type16_t source_type;
    hsa brig segment8_t segment;
    hsa brig seg cvt modifier8_t modifier;
} hsa brig inst seg cvt_t;

Fields are:

- hsa brig base_t base — base.base.kind must be HSA_BRIG_KIND_INST_CVT.
  base.type must be the data type of the destination of the convert instruction.

- hsa brig type16_t source_type — Type of the source.

- hsa brig segment8_t segment — Segment. A member of the hsa brig segment_t
  enumeration. See 18.3.27 hsa brig segment_t (on page 352).

- hsa brig seg cvt modifier8_t modifier — Segment conversion modifier flags of
  the instruction. See 18.3.26 hsa brig seg cvt modifier_t (on page 351).

18.5.2.15 hsa brig inst signal_t

The hsa brig inst signal_t format is used for signal instructions.

Syntax is:

typedef struct hsa brig inst signal_s {
    hsa brig base_t base;
    hsa brig type16_t signal_type;
    hsa brig memory order8_t memory_order;
    hsa brig atomic operation8_t signal_operation;
} hsa brig inst signal_t;
Fields are:

- `uint16_t kind`— `base.base.kind` must be `HSA_BRIG_KIND_INST_SIGNAL`.
  - `base.opcode` must be `HSA_BRIG_OPCODE_SIGNAL` or `HSA_BRIG_OPCODE_SIGNALNORET`.
  - `base.type` must be the data type of the destination and source of the signal instruction.

- `hsa Brig_type16_t signal_type`— Type of the signal. Must be `HSA_BRIG_TYPE_SIG32` or `HSA_BRIG_TYPE_SIG64`.

- `hsa Brig_memory_order8_t memory_order`— Memory order of the signal instruction. See 18.3.16 `hsa Brig_memory_order_t` (on page 345).

- `hsa Brig_atomic_operation8_t signal_operation`— The signal instruction such as `add` or `or`. See 18.3.5 `hsa Brig_atomic_operation_t` (on page 341).

### 18.5.2.16 hsa Brig inst source type_t

The `hsa Brig inst source_type_t` format is used for instructions that have different types for their destination and source operands.

Syntax is:
```c
typedef struct hsa Brig inst source_type_s {
    hsa Brig inst base_t base;
    hsa Brig type16_t source_type;
    uint16_t reserved;
} hsa Brig inst source_type_t;
```

Fields are:

- `hsa Brig inst base_t base`— `base.base.kind` must be `HSA_BRIG_KIND_INST_SOURCE_TYPE`.
  - `base.type` must be the data type of the destination of the instruction.

- `hsa Brig type16_t source_type`— Type of the source.

- `uint16_t reserved`— Must be 0.

### 18.5.2.17 hsa ext Brig inst image t

The `hsa ext Brig inst image t` format is used for the image instructions.

Syntax is:
```c
typedef struct hsa ext Brig inst image s {
    hsa Brig inst base_t base;
    hsa Brig type16_t image_type;
    hsa Brig type16_t coord_type;
    hsa Brig image geometry8_t geometry;
    uint8_t equiv_class;
    uint16_t reserved;
} hsa ext Brig inst image_t;
```

Fields are:

- `hsa Brig inst base_t base`— `base.base.kind` must be `HSA_BRIG_KIND_INST_IMAGE`.
  - `base.type` must be the data type of the destination of the image instruction.

- `hsa Brig type16_t image_type`— Type of the image. Must be `HSA_BRIG_TYPE_ROIMG`, `HSA_BRIG_TYPE_WOIMG` or `HSA_BRIG_TYPE_RWIMG`.

- `hsa Brig type16_t coord_type`— Type of the coordinates.
18.5.2.18 hsa_ext_brig_inst_query_image_t

The hsa_ext_brig_inst_query_image_t format is used for the queryimage instruction.

Syntax is:

```c
typedef struct hsa_ext_brig_inst_query_image_s {
    hsa_brig_inst_base_t base;
    hsa_brig_type16_t image_type;
    hsa_ext_brig_image_geometry8_t geometry;
    hsa_ext_brig_image_query8_t query;
} hsa_ext_brig_inst_query_image_t;
```

Fields are:

- **hsa_brig_inst_base_t base** — base.base.kind must be HSA_BRIG_KIND_INST_QUERY_IMAGE. base.type must be the data type of the destination of the query instruction.
- **hsa_brig_type16_t image_type** — Type of the image. Must be HSA_BRIG_TYPE_RWIMG, HSA_BRIG_TYPE_WOIMG, or HSA_BRIG_TYPE_ROIMG.
- **hsa_ext_brig_image_geometry8_t geometry** — Image geometry. See 18.3.35 hsa_ext_brig_image_geometry_t (on page 358).
- **hsa_ext_brig_image_query8_t query** — Image property being queried. See 18.3.36 hsa_ext_brig_image_query_t (on page 358).

18.5.2.19 hsa_ext_brig_inst_query_sampler_t

The hsa_ext_brig_inst_query_sampler_t format is used for the queriesampler instruction.

Syntax is:

```c
typedef struct hsa_ext_brig_inst_query_sampler_s {
    hsa_brig_inst_base_t base;
    hsa_ext_brig_sampler_query8_t query;
    uint8_t reserved[3];
} hsa_ext_brig_inst_query_sampler_t;
```

Fields are:

- **hsa_brig_inst_base_t base** — base.base.kind must be HSA_BRIG_KIND_INST_QUERY_SAMPLER. base.type must be the data type of the destination of the sampler instruction.
- **hsa_ext_brig_sampler_query8_t query** — Sampler property being queried. See 18.3.40 hsa_ext_brig_sampler_query_t (on page 359).
- **uint8_t reserved[3]** — 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 hsa_brig_section_header_t entry. The name of the section must be hsa_operand. See 18.3.25 hsa_brig_section_header_t (on page 351).

All operand entries (hsa_brig_operand_*) in the hsa_operand section must start with a hsa_brig_base_t structure. The kind field of the hsa_brig_base_t structure must be in the right-open interval [HSA_BRIG_KIND_OPERAND_BEGIN, HSA_BRIG_KIND_OPERAND_END). See 18.3.12 hsa_brig_kind_t (on page 344).

To reduce the size of the hsa_operand section it is allowed, but not required, to reference an already created hsa_brig_operand_* entry, rather than create duplicate hsa_brig_operand_* 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hsa_brig_operand_address_t</td>
<td>Used for address expressions. See 18.6.2 hsa_brig_operand_address_t (on page 383).</td>
</tr>
<tr>
<td>hsa_brig_operand_align_t</td>
<td>Used for aligning aggregate data constants. See 18.6.3 hsa_brig_operand_align_t (on page 384).</td>
</tr>
<tr>
<td>hsa_brig_operand_code_list_t</td>
<td>List of references to entries in the hsa_code section. See 18.6.4 hsa_brig_operand_code_list_t (on page 384).</td>
</tr>
<tr>
<td>hsa_brig_operand_ref_t</td>
<td>A reference to an entry in the hsa_code section. See 18.6.5 hsa_brig_operand_code_ref_t (on page 384).</td>
</tr>
<tr>
<td>hsa_brig_operand_constant_bytes_t</td>
<td>Declares a constant value as an array of bytes. See 18.6.6 hsa_brig_operand_constant_bytes_t (on page 385).</td>
</tr>
<tr>
<td>hsa_brig_operand_constant_expression_t</td>
<td>Declares a constant value as an expression. See 18.6.7 hsa_brig_operand_constant_expression_t (on page 386).</td>
</tr>
<tr>
<td>hsa_brig_operand_constant_operand_list_t</td>
<td>Declares a constant as a list of operands. See 18.6.8 hsa_brig_operand_constant_operand_list_t (on page 387).</td>
</tr>
<tr>
<td>hsa_brig_operand_operand_list_t</td>
<td>List of references to entries in the hsa_operand section. See 18.6.9 hsa_brig_operand_operand_list_t (on page 388).</td>
</tr>
<tr>
<td>hsa_brig_operand_register_t</td>
<td>A register (c, s, d, or q). See 18.6.10 hsa_brig_operand_register_t (on page 389).</td>
</tr>
<tr>
<td>hsa_brig_operand_string_t</td>
<td>A textual string. See 18.6.11 hsa_brig_operand_string_t (on page 389).</td>
</tr>
<tr>
<td>hsa_brig_operand_wavesize_t</td>
<td>The wavesize operand. See 18.6.12 hsa_brig_operand_wavesize_t (on page 390).</td>
</tr>
<tr>
<td>hsa_brig_operand_zero_t</td>
<td>Used to specify zero bytes in aggregate data constants. See 18.6.13 hsa_brig_operand_zero_t (on page 390).</td>
</tr>
<tr>
<td>hsa_ext_brig_operand_constant_image_t</td>
<td>Declares the properties of an image referenced by an image handle constant. See 18.6.14 hsa_ext_brig_operand_constant_image_t (on page 390).</td>
</tr>
<tr>
<td>hsa_ext_brig_operand_constant_sampler_t</td>
<td>Declares the properties of a sampler referenced by a sample handle constant. 18.6.15 hsa_ext_brig_operand_constant_sampler_t (on page 391).</td>
</tr>
</tbody>
</table>
18.6.1 Constant operands

Constant values are represented by operands with a kind field of HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_IMAGE, HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST, or HSA_BRIG_KIND_OPERAND_CONSTANT_SAMPLER. The type of the constant is given by the type field of the hsa_brig_operand_constant.*.

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 hsa_brig_directive_variable_t.
- 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 hsa_brig_inst_*.
- 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 hsa_brig_directive_*.
- 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 (note that b1 is only allowed for integer literal constants).
- If the expected type is b128, the constant type must be a packed type of the same byte size as b128. The BRIG binary representation must use the same packed type as the packed typed constant in the HSAIL textual representation.
- If the expected type is a bit type, other than b1 or b128, the constant type must be an integer, float, or packed type of the same byte size as the expected type. The BRIG binary representation must either use:
  - The same type as the typed constant in the HSAIL textual representation.
  - An unsigned integer type for a positive integer literal constant, integer symbolic expression constant, or null pointer address constant in the HSAIL textual representation.
  - A signed integer type for a negative integer literal constant in the HSAIL textual representation.
  - A float type for a floating-point constant in the HSAIL textual representation.
- If the expected type is an integer type, the constant type must be an integer type of the same byte size as the expected type. The BRIG binary representation must either use:
  - The same type as the typed constant in the HSAIL textual representation.
  - An unsigned integer type for a positive integer literal constant, integer symbolic expression constant, or null pointer address constant in the HSAIL textual representation.
A signed integer type for a negative integer literal constant in the HSAIL textual representation.

- If the expected type is a bit type array (note that a bl array type is not allowed), the constant must be an aggregate constant represented as a hsa_brig_operand_constant_operand_list_t with a type field of HSA_BRIG_TYPE_NONE. The byte size of the aggregate constant value must be the same as the byte size of the 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 hsa_brig_operand_constant_bytes_t. 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 hsa_brig_operand_constant_operand_list_t 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 hsa_brig_operand_address_t

hsa_brig_operand_address_t is used for address expressions. See 4.18 Address expressions (on page 115).

Syntax is:

```c
typedef struct hsa_brig_operand_address_s {
    hsa_brig_base_t base;
    hsa_brig_code_offset32_t symbol;
    hsa_brig_operand_offset32_t reg;
    hsa_brig_offset64_t offset;
} hsa_brig_operand_address_t;
```

Fields are:

- `hsa_brig_base_t base`: base.kind must be HSA_BRIG_KIND_OPERAND_ADDRESS.
- `hsa_brig_code_offset32_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 (on page 362). If the address expression has no symbol name then must be 0.
- `hsa_brig_operand_offset32_t reg`: Byte offset in the hsa_operand section to a HSA_BRIG_KIND_OPERAND_REGISTER operand. If the address expression has no register then must be 0.
- `hsa_brig_offset64_t offset`: Byte offset to add to the address. See 18.3.29 hsa_brig_offset64_t (on page 355). 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 hsa_brig_offset64_t must be 0. The finalizer will
order the bytes of the offset value according to the byte endianness of the HSA platform for which
machine code is being generated.

18.6.3 hsa_brig_operand_align_t

hsa_brig_operand_align_t is used for aligning aggregate data constants. See 4.8.4 Aggregate
constants (on page 99).

Syntax is:

typedef struct hsa_brig_operand_align_s {
    hsa_brig_base_t base;
    hsa_brig_alignment8_t align;
    uint8_t reserved[3];
} hsa_brig_operand_align_t;

Fields are:

- hsa_brig_base_t base — base.kind must be HSA_BRIG_KIND_OPERAND_ALIGN.
- hsa_brig_alignment8_t align — The required alignment in bytes for the next aggregate
data constant element. Causes zero padding between aggregate constant elements, or zero fill if the last
aggregate constant element. See 18.3.2 hsa_brig_alignment_t (on page 340).
- uint8_t reserved[3] — Must be 0.

18.6.4 hsa_brig_operand_code_list_t

hsa_brig_operand_code_list_t is used for a list of references to entries in the hsa_code section.

Syntax is:

typedef struct hsa_brig_operand_code_list_s {
    hsa_brig_base_t base;
    hsa_brig_data_offset_code_list32_t elements;
} hsa_brig_operand_code_list_t;

Fields are:

- hsa_brig_base_t base — base.kind must be HSA_BRIG_KIND_OPERAND_CODE_LIST.
- hsa_brig_data_offset_code_list32_t elements — Byte offset of the entry in the hsa_data
section that contains a variable-sized array of byte offsets to entries in the hsa_code section.
The byte_count of the entry must be exactly (4 * number of elements).
  - When used as a function actual argument list, each element must reference HSA_BRIG_KIND_DIRECTIVE_VARIABLE with HSA_BRIG_SEGMENT_ARG segment.
  - When used as a function list, each element must reference a HSA_BRIG_KIND_DIRECTIVE_FUNCTION or HSA_BRIG_KIND_DIRECTIVE_INDIRECT_FUNCTION
directive. See 18.5.1.1 Declarations and definitions in the same module (on page 362).
  - When used as a label list, each element must reference a HSA_BRIG_KIND_DIRECTIVE_LABEL directive in the same function scope.

18.6.5 hsa_brig_operand_code_ref_t

hsa_brig_operand_code_ref_t is used to reference an entry in the hsa_code section.
Syntax is:

```c
typedef struct hsa Brig_operand_code_ref_s {
    hsa Brig_base_t base;
    hsa Brig_code_offset32_t ref;
} hsa Brig_operand_code_ref_t;
```

Fields are:

- `hsa Brig_base_t base` — `base.kind` must be `HSA_BRIG_KIND_OPERAND_CODE_REF`.
- `hsa Brig_code_offset32_t ref` — Byte offset to the place in the `hsa_code` section.
  - When used to reference a kernel, must reference `HSA_BRIG_KIND_DIRECTIVE_KERNEL` directive.
  - When used to reference a function, must reference `HSA_BRIG_KIND_DIRECTIVE_FUNCTION` or `HSA_BRIG_KIND_DIRECTIVE_INDIRECT_FUNCTION` directive.
  - When used to reference a signature, must reference `HSA_BRIG_KIND_DIRECTIVE_SIGNATURE` directive.
  - When used to reference a variable, must reference `HSA_BRIG_KIND_DIRECTIVE_VARIABLE` directive.
  - When used to reference a fbarrier, must reference `HSA_BRIG_KIND_DIRECTIVE_FBARRIER` directive.
  - When used to reference a label, must reference `HSA_BRIG_KIND_DIRECTIVE_LABEL` directive in the same function scope.

See 18.5.1.1 Declarations and definitions in the same module (on page 362).

### 18.6.6 `hsa Brig_operand_constant_bytes_t`

`hsa Brig_operand_constant_bytes_t` specifies a constant value as an array of bytes.

Syntax is:

```c
typedef struct hsa Brig_operand_constant_bytes_s {
    hsa Brig_base_t base;
    hsa Brig_type16_t type;
    uint16_t reserved;
    hsa Brig_data_offset_string32_t bytes;
} hsa Brig_operand_constant_bytes_t;
```

Fields are:

- `hsa Brig_base_t base` — `base.kind` must be `HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES`.
- `hsa Brig_type16_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.
- `hsa Brig_data_offset_string32_t bytes` — Byte offset of the entry in the `hsa_data` section that contains the bytes of the constant value. The `byte_count` of the entry must be the byte size of the constant value.
This operand is used to represent the value of all integer constants, float constants, integer typed constants, float typed constants, packed typed constants that do not involve integer symbolic expression constants, signal typed constants, array typed constants of integer, float, packed that do not involve integer symbolic expressions 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 `type`. 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 `type`. 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 machine code is being generated. Note that for packed typed `type` 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 `hsa brig_operand_constant_expression_t`

`hsa brig_operand_constant_expression_t` specifies a constant value as an expression.

**Syntax is:**

```c
typedef struct hsa brig_operand_constant_expression_s {
    hsa brig base_t base;
    hsa brig type16_t type;
    hsa brig expression operation16_t expression operation;
    hsa brig data offset operand list32_t operands;
} hsa brig_operand_constant_expression_t;
```

**Fields are:**

- `hsa brig base_t base` — Base `kind` must be `HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION`.
- `hsa brig type16_t type` — Data `type` of the constant.
- `hsa brig expression operation16_t expression operation` — Operation associated with the expression (see 18.3.11 `hsa brig expression operation_t` (on page 343)).
- `hsa brig data offset operand list32_t operands` — Byte offset of the entry in the `hsa_data` section that contains a variable-sized array of byte offsets to entries in the `hsa operand` section. The `byte_count` of the entry must be exactly (`4 * number of operands`). The number of operands, their kinds and `type` depends on the expression `opcode`:
  - `HSA_BRIG_EXPRESSION_OPERATION_NULLPTR_FLAT`, `HSA_BRIG_EXPRESSION_OPERATION_NULLPTR_GROUP`
PRIVATE, HSA_BRIG_EXPRESSION_OPERATION_NULLPTR_KERNARG — There must be 0 operands. type must be an integer type.

- HSA_BRIG_EXPRESSION_OPERATION_ADDR — There must be 1 or 2 operands. The first operand must be a HSA_BRIG_KIND_OPERAND_CODE_REF for an allowed program scope global, readonly segment variable, kernel, or indirect function (see Integer constants (on page 86)). If a kernel or indirect function then the second operand must not be present. If present the second operand must be a HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES with type HSA_BRIG_TYPE_U64. type must be an integer type.

See 18.5.1.1 Declarations and definitions in the same module (on page 362).

18.6.8 hsa_brig_operand_constant_operand_list_t

hsa_brig_operand_constant_operand_list_t specifies the data value.

Syntax is:

```c
typedef struct hsa_brig_operand_constant_operand_list_s {
    hsa_brig_base_t base;
    hsa_brig_type16_t type;
    uint16_t reserved;
    hsa_brig_data_offset_operand_list32_t elements;
} hsa_brig_operand_constant_operand_list_t;
```

Fields are:

- hsa_brig_base_t base — base.kind must be HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST.

- hsa_brig_type16_t type — Must be an integer array, packed array, image array, or sampler array type if an array typed constant; must be a packed type if a packed typed constant; or HSA_BRIG_TYPE_NONE if an aggregate constant.

- uint16_t reserved — Must be 0.

- hsa_brig_data_offset_operand_list32_t elements — Byte offset of the entry in the hsa_data section that contains a variable-sized array of byte offsets to operands in the hsa_operand section. The byte count of the entry must be exactly (4 * number of operands). The operands must either be HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_IMAGE, HSA_BRIG_KIND_OPERAND_CONSTANT_SAMPLER, HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST, HSA_BRIG_KIND_OPERAND_ALIGN, or HSA_BRIG_KIND_OPERAND_ZERO.
If the constant is an aggregate constant, then the type field must be `HSA_BRIG_TYPE_NONE`. The operands must either be `HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES`, `HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION`, `HSA_BRIG_KIND_OPERAND_CONSTANT_IMAGE`, `HSA_BRIG_KIND_OPERAND_CONSTANT_SAMPLER`, `HSA_BRIG_KIND_OPERAND_CONSTANT_LIST`, `HSA_BRIG_KIND_OPERAND_ALIGN`, or `HSA_BRIG_KIND_OPERAND_ZERO` that correspond, in the same order, to the elements of the aggregate constant. If an element of the aggregate constant is an array typed constant, then the array typed constant elements are represented, in the same order, directly as elements of the aggregate constant with the array element type, except adjacent `HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES` elements are collapsed into a single `HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES` with the original array type. `HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST` is only allowed for operands that are packed typed constants involving integer symbolic expression constants. The byte size of the constant is the sum of the byte sizes of the operands, accounting for any padding created by any `HSA_BRIG_KIND_OPERAND_ALIGN` operands.

If the constant is an array typed constant, then the type field must be an array type. The operands must either be `HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES`, `HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION`, `HSA_BRIG_KIND_OPERAND_CONSTANT_IMAGE`, `HSA_BRIG_KIND_OPERAND_CONSTANT_SAMPLER`, or `HSA_BRIG_KIND_OPERAND_CONSTANT_LIST` that correspond, in the same order, to the elements of the array typed constant. The type of each operand must match the array element type. `HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST` is only allowed for operands that are packed typed constants involving integer symbolic expression constants. The byte size of the constant is the byte size of the array element type multiplied by the number of operands. Except all operands of `HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST` must not be `HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES` and instead must be a single `HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES` with an array type.

If the constant is a packed typed constant, then the type field must be a packed type. The operands must either be `HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES` or `HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION` that correspond, in reverse order, to the elements of the packed type constant. The type of each operand must match the packed element type. The number of operands must match the packed element count. The byte size of the constant is the byte size of the packed type. Except all operands of `HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST` must not be `HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES` and instead must be a single `HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES` with a packed type. When finalized the elements of a packed type constant are processed according to the byte endianness of the HSA platform for which machine code is being generated. Note that 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.

### 18.6.9 `hsa_brig_operand_operand_list_t`

`hsa_brig_operand_operand_list_t` is used for a list of references to entries in the `hsa_operand` section.

Syntax is:

```c
typedef struct hsa_brig_operand_operand_list_s {
  hsa_brig_base_t base;
  hsa_brig_data_offset_operand_list32_t elements;
} hsa_brig_operand_operand_list_t;
```
Fields are:

- **hsa Brig base_t base** — base.kind must be HSA_BRIG_KIND_OPERAND_OPERAND_LIST.
- **hsa Brig data offset operand list32_t elements** — Byte offset of the entry in the hsa_data section that contains a variable-sized array of byte offsets to entries in the hsa_operand section. The byte count of the entry must be exactly (4 * number of elements).
  - When used as a destination vector operand, each element must reference a HSA_BRIG_KIND_OPERANDREGISTER directive.
  - When used as a source vector operand, each element must reference a HSA_BRIG_KIND_OPERANDREGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST, or HSA_BRIG_KIND_OPERAND_WAVESIZE directive.

### 18.6.10 hsa Brig operand_register_t

hsa Brig operand_register_t is used for a register (c, s, d, or q).

Syntax is:
```c
typedef struct hsa Brig operand_register_s {
    hsa Brig base_t base;
    hsa Brig register_kind16_t reg_kind;
    uint16_t reg_num;
} hsa Brig operand_register_t;
```

Fields are:

- **hsa Brig base_t base** — base.kind must be HSA_BRIG_KIND_OPERAND_REGISTER.
- **hsa Brig register_kind16_t reg_kind** — The register kind. Must be HSA_BRIG_REGISTER_KIND_CONTROL for c register, HSA_BRIG_REGISTER_KIND_SINGLE for s register, HSA_BRIG_REGISTER_KIND_DOUBLE for d register, and HSA_BRIG_REGISTER_KIND_QUAD for q register.
- **uint16_t reg_num** — The register number.

### 18.6.11 hsa Brig operand_string_t

hsa Brig operand_string_t is used for a textual string.

Syntax is:
```c
typedef struct hsa Brig operand_string_s {
    hsa Brig base_t base;
    hsa Brig data offset string32_t string;
} hsa Brig operand_string_t;
```

Fields are:

- **hsa Brig base_t base** — base.kind must be HSA_BRIG_KIND_OPERAND_STRING.
- **hsa Brig data offset string32_t string** — Byte offset of the entry in the hsa_data section that contains the textual string.
18.6.12 hsa brig operand wavesize_t

hsa brig operand wavesize_t is the wavesize operand, which is a compile-time value equal to the size of a wavefront.

Syntax is:

typedef struct hsa brig operand wavesize_s {
    hsa brig base t base;
} hsa brig operand wavesize_t;

Fields are:

- hsa brig base t base — base.kind must be HSA_BRIG_KIND_OPERAND_WAVESIZE.

18.6.13 hsa brig operand zero_t

hsa brig operand zero_t is used to specify zero bytes in aggregate data constants. See 4.8.4 Aggregate constants (on page 99).

Syntax is:

typedef struct hsa brig operand zero_s {
    hsa brig base t base;
    hsa brig uint64 t zero_byte_count;
} hsa brig operand zero_t;

Fields are:

- hsa brig base t base — base.kind must be HSA_BRIG_KIND_OPERAND ZERO.
- hsa brig uint64 t zero_byte_count — The number of zero bytes to add to an aggregate data constant. See 18.3.2 hsa brig alignment_t (on page 340).

18.6.14 hsa ext brig operand constant image_t

hsa ext brig operand constant image_t specifies the properties of an image referenced by an image handle constant. For more information, see 7.1.7 Image creation and image handles (on page 227).

Syntax is:

typedef struct hsa ext brig operand constant image_s {
    hsa brig base t base;
    hsa brig type16 t type;
    hsa ext brig image geometry8_t geometry;
    hsa ext brig image channel order8_t channel_order;
    hsa ext brig image channel type8_t channel_type;
    uint8 t reserved[3];
    hsa brig uint64_t width;
    hsa brig uint64_t height;
    hsa brig uint64_t depth;
    hsa brig uint64_t array;
} hsa ext brig operand constant image_t;

Fields are:

- hsa brig base t base — base.kind must be HSA_BRIG_KIND_OPERAND_CONSTANT_IMAGE.
- hsa brig type16 t type — Data type of the constant. Must be HSA_BRIG_TYPE_ROIMG, HSA_BRIG_TYPE_WOIMG, or HSA_BRIG_TYPE_RWIMG.
- hsa ext brig image geometry8_t geometry — Geometry for the image. A member of
the hsa_ext_brig_image_geometry_t enumeration. See 18.3.35 hsa_ext_brig_image_geometry_t (on page 358).

- hsa_ext_brig_image_channel_order8_t channel_order — 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 hsa_ext_brig_image_channel_order8_t enumeration. See 18.3.33 hsa_ext_brig_image_channel_order8_t (on page 357).

- hsa_ext_brig_image_channel_type8_t channel_type — Channel type for storing images. Images can be stored and accessed in assorted formats. A member of the hsa_ext_brig_image_channel_type8_t enumeration. See 18.3.34 hsa_ext_brig_image_channel_type8_t (on page 357).

- uint8_t reserved[3] — Must be 0.

- hsa_brig_uint64_t width — The image width. Must be greater than zero for all image geometries.

- hsa_brig_uint64_t height — The image height. Must be greater than zero if geometry is HSA_EXT_BRIG_IMAGE_GEOMETRY_2D, HSA_EXT_BRIG_IMAGE_GEOMETRY_3D, HSA_EXT_BRIG_IMAGE_GEOMETRY_2DA, HSA_EXT_BRIG_IMAGE_GEOMETRY_2DDEPTH, or HSA_EXT_BRIG_IMAGE_GEOMETRY_2DADEPTH; otherwise must be 0.

- hsa_brig_uint64_t depth — The image depth. Must be greater than zero if geometry is HSA_EXT_BRIG_IMAGE_GEOMETRY_3D; otherwise must be 0.

- hsa_brig_uint64_t array — The number of images in the array. Must be greater than zero if geometry is HSA_EXT_BRIG_IMAGE_GEOMETRY_1DA, HSA_EXT_BRIG_IMAGE_GEOMETRY_2DA, or HSA_EXT_BRIG_IMAGE_GEOMETRY_2DDEPTH; otherwise must be 0.

### 18.6.15 hsa_ext_brig_operand_constant_sampler_t

hsa_ext_brig_operand_constant_sampler_t specifies the properties of a sampler referenced by a sampler handle constant. For more information, see 7.1.8 Sampler creation and sampler handles (on page 232).

Syntax is:

typedef struct hsa_ext_brig_operand_constant_sampler_s {
  hsa_brig_base_t base;
  hsa_brig_type16_t type;
  hsa_ext_brig_sampler_coord_normalization8_t coord;
  hsa_ext_brig_sampler_filter8_t filter;
  hsa_ext_brig_sampler_addressing8_t addressing;
  uint8_t reserved[3];
} hsa_ext_brig_operand_constant_sampler_t;

Fields are:

- hsa_brig_base_t base — base.kind must be HSA_BRIG_KIND_OPERAND_CONSTANT_SAMPLER.

- hsa_brig_type16_t type — Data type of the constant. Must be HSA_BRIG_TYPE_SAMP.

- hsa_ext_brig_sampler_coord_normalization8_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 2DDEPTH images which always use HSA_
Chapter 18. BRIG: HSAIL Binary Format  18.7 BRIG syntax for instructions

EXT_BRIG_SAMPLERCOORD_NORMALIZATION_UNNORMALIZED. Must be a member of the hsa_ext brig_sampler_coord_normalization_t enumeration. See 18.3.38 hsa_ext_brigan_s_mple_coord_normalization_t (on page 359).

- hsa_ext_brigan_sampler_filter8_t_filter — The filter mode used to specify how image elements are selected. Must be a member of the hsa_ext_brigan_sampler_filter_t enumeration. If coord is HSA_EXT_BRIG_SAMPLERCOORD_NORMALIZATION_UNNORMALIZED then must be HSA_EXT_BRIG_SAMPLER_FILTER_NEAREST. See 18.3.39 hsa_ext_brigan_sampler_filter_t (on page 359).

- hsa_ext_brigan_sampler_addressing8_t_addressing — The addressing mode used when coordinates are out of range of the corresponding image dimension size. Must be a member of the hsa_ext_brigan_sampler_addressing_t enumeration. If coord is HSA_EXT_BRIG_SAMPLERCOORD_NORMALIZATION_UNNORMALIZED then must be HSA_EXT_BRIG_SAMPLERADDRESSING_UNDEFINED, HSA_EXT_BRIG_SAMPLERADDRESSING_CLAMP_TO_EDGE or HSA_EXT_BRIG_SAMPLERADDRESSING_CLAMP_TO_BORDER. Does not apply to the array index coordinate of 1DA, 2DA and 2DADEPTH images which always use HSA_EXT_BRIG_SAMPLERADDRESSING_CLAMP_TO_EDGE. See 18.3.37 hsa_ext_brigan_sampler_addressing_t (on page 359).

- uint8_t reserved[3] — Must be 0.

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 HSA_BRIG_KIND_INST_BASIC if all modifiers have default values, or by HSA_BRIG_KIND_INST_MOD whether or not default modifiers are used. Using HSA_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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
<th>Operand 1</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_ABS</td>
<td>HSA_BRIG_KIND_INST_BASIC (if only default modifiers are used) or HSA_BRIG_KIND_INST_MOD</td>
<td>dest</td>
<td>src</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_ADD</td>
<td>HSA_BRIG_KIND_INST_BASIC (if only default modifiers are used) or HSA_BRIG_KIND_INST_MOD</td>
<td>dest</td>
<td>src</td>
<td>src</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_BORROW</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_CARRY</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_DIV</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_MAX</td>
<td>HSA_BRIG_KIND_INST_BASIC (if only default modifiers are used) or HSA_BRIG_KIND_INST_MOD</td>
<td>dest</td>
<td>src</td>
<td>src</td>
</tr>
</tbody>
</table>
18.7.1.2 BRIG syntax for integer optimization instruction

Table 18–5 BRIG syntax for integer optimization instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_MAD</td>
<td>HSA_BRIG_KIND_INST_BASIC (if only default modifiers are used) or HSA_BRIG_KIND_INST_MOD</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td>src</td>
</tr>
</tbody>
</table>

dest: must be HSA_BRIG_KIND_OPERAND_REGISTER.

src: must be HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST, or HSA_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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_MAD24</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td>src</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_MAD24HI</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td>src</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_MUL24</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td>src</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_MUL24HI</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td>src</td>
</tr>
</tbody>
</table>

dest: must be HSA_BRIG_KIND_OPERAND_REGISTER.

src: must be HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST, or HSA_BRIG_KIND_OPERAND_WAVESIZE.
18.7.1.4 BRIG syntax for integer shift instructions

Table 18–7 BRIG syntax for integer shift instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
<th>Operand 1</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_SHL</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_SHR</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
</tr>
</tbody>
</table>

dest: must be HSA_BRIG_KIND_OPERAND_REGISTER.

src: must be HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST, or HSA_BRIG_KIND_OPERAND_WAVESIZE.

18.7.1.5 BRIG syntax for individual bit instructions

Table 18–8 BRIG syntax for individual bit instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
<th>Operand 1</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_AND</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_NOT</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_OR</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_POPCOUNT</td>
<td>HSA_BRIG_KIND_INST_SOURCE_TYPE</td>
<td>dest</td>
<td>src</td>
<td>src</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_XOR</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
</tr>
</tbody>
</table>

dest: must be HSA_BRIG_KIND_OPERAND_REGISTER.

dest: must be HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST, or HSA_BRIG_KIND_OPERAND_WAVESIZE.

18.7.1.6 BRIG syntax for bit string instructions

Table 18–9 BRIG syntax for bit string instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Oper. 0</th>
<th>Oper. 1</th>
<th>Oper. 2</th>
<th>Oper. 3</th>
<th>Oper. 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_BITEXTRACT</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td>src</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_BITINSERT</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td>src</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_BITMASK</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_BITREV</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_BITSELECT</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_FIRSTBIT</td>
<td>HSA_BRIG_KIND_INST_SOURCE_TYPE</td>
<td>dest</td>
<td>src</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_LASTBIT</td>
<td>HSA_BRIG_KIND_INST_SOURCE_TYPE</td>
<td>dest</td>
<td>src</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dest: must be HSA_BRIG_KIND_OPERAND_REGISTER.

src: must be HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST, or HSA_BRIG_KIND_OPERAND_WAVESIZE.
18.7.1.7 BRIG syntax for copy (move) instructions

Table 18–10 BRIG syntax for copy (move) instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
<th>Operand 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_COMBINE</td>
<td>HSA_BRIG_KIND_INST_SOURCE_TYPE</td>
<td>dest</td>
<td>src-vector</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_EXPAND</td>
<td>HSA_BRIG_KIND_INST_SOURCE_TYPE</td>
<td>dest-vector</td>
<td>src</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_IDA</td>
<td>HSA_BRIG_KIND_INST_ADDR</td>
<td>dest</td>
<td>address</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_MOV</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
</tr>
</tbody>
</table>

dest: must be HSA_BRIG_KIND_OPERAND_REGISTER.

src-vector: must be HSA_BRIG_KIND_OPERAND_OPERAND_LIST that references a list of HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_OPERAND_OPERAND_LIST, or HSA_BRIG_KIND_OPERAND_WAVESIZE operands.

dest-vector: must be HSA_BRIG_KIND_OPERAND_OPERAND_LIST that references a list of HSA_BRIG_KIND_OPERAND_REGISTER operands.

src: must be HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST, or HSA_BRIG_KIND_OPERAND_WAVESIZE.

address: must be HSA_BRIG_KIND_OPERAND_ADDRESS.

18.7.1.8 BRIG syntax for packed data instructions

Table 18–11 BRIG syntax for packed data instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_SHUFFLE</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td>number</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_UNPACKHI</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_UNPACKLO</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_PACK</td>
<td>HSA_BRIG_KIND_INST_SOURCE_TYPE</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td>src</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_UNPACK</td>
<td>HSA_BRIG_KIND_INST_SOURCE_TYPE</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td></td>
</tr>
</tbody>
</table>

dest: must be HSA_BRIG_KIND_OPERAND_REGISTER.

src: must be HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST, or HSA_BRIG_KIND_OPERAND_WAVESIZE.

number: must be HSA_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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_CMOV</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td>src</td>
</tr>
</tbody>
</table>
dest: must be HSA_BRIG_KIND_OPERAND_REGISTER.

src: must be HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST, or HSA_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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_ADD</td>
<td>HSA_BRIG_KIND_INST_BASIC (if only default modifiers are used) or HSA_BRIG_KIND_INST_MOD</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_CEIL</td>
<td>HSA_BRIG_KIND_INST_BASIC (if only default modifiers are used) or HSA_BRIG_KIND_INST_MOD</td>
<td>dest</td>
<td>src</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_DIV</td>
<td>HSA_BRIG_KIND_INST_BASIC (if only default modifiers are used) or HSA_BRIG_KIND_INST_MOD</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_FLOOR</td>
<td>HSA_BRIG_KIND_INST_BASIC (if only default modifiers are used) or HSA_BRIG_KIND_INST_MOD</td>
<td>dest</td>
<td>src</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_FMA</td>
<td>HSA_BRIG_KIND_INST_BASIC (if only default modifiers are used) or HSA_BRIG_KIND_INST_MOD</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td>src</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_FRAC</td>
<td>HSA_BRIG_KIND_INST_BASIC (if only default modifiers are used) or HSA_BRIG_KIND_INST_MOD</td>
<td>dest</td>
<td>src</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_MAX</td>
<td>HSA_BRIG_KIND_INST_BASIC (if only default modifiers are used) or HSA_BRIG_KIND_INST_MOD</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_MIN</td>
<td>HSA_BRIG_KIND_INST_BASIC (if only default modifiers are used) or HSA_BRIG_KIND_INST_MOD</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_MUL</td>
<td>HSA_BRIG_KIND_INST_BASIC (if only default modifiers are used) or HSA_BRIG_KIND_INST_MOD</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_RINT</td>
<td>HSA_BRIG_KIND_INST_BASIC (if only default modifiers are used) or HSA_BRIG_KIND_INST_MOD</td>
<td>dest</td>
<td>src</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_SQRT</td>
<td>HSA_BRIG_KIND_INST_BASIC (if only default modifiers are used) or HSA_BRIG_KIND_INST_MOD</td>
<td>dest</td>
<td>src</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_SUB</td>
<td>HSA_BRIG_KIND_INST_BASIC (if only default modifiers are used) or HSA_BRIG_KIND_INST_MOD</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_TRUNC</td>
<td>HSA_BRIG_KIND_INST_BASIC (if only default modifiers are used) or HSA_BRIG_KIND_INST_MOD</td>
<td>dest</td>
<td>src</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dest: must be HSA_BRIG_KIND_OPERAND_REGISTER.

src: must be HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, or HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST.
### 18.7.1.11 BRIG syntax for floating-point optimization instruction

Table 18–14 BRIG syntax for floating-point optimization instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_MAD</td>
<td>HSA_BRIG_KIND_INST_BASIC (if only default modifiers are used) or HSA_BRIG_KIND_INST_MOD</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td>src</td>
</tr>
</tbody>
</table>

**dest:** must be HSA_BRIG_KIND_OPERAND_REGISTER.

**src:** must be HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, or HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST.

### 18.7.1.12 BRIG syntax for floating-point bit instructions

Table 18–15 BRIG syntax for floating-point bit instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
<th>Operand 1</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_ABS</td>
<td>HSA_BRIG_KIND_INST_BASIC (if only default modifiers are used) or HSA_BRIG_KIND_INST_MOD</td>
<td>dest</td>
<td>src</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_CLASS</td>
<td>HSA_BRIG_KIND_SOURCE_TYPE</td>
<td>dest</td>
<td>src</td>
<td>cond</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_COPYSIGN</td>
<td>HSA_BRIG_KIND_INST_BASIC (if only default modifiers are used) or HSA_BRIG_KIND_INST_MOD</td>
<td>dest</td>
<td>src</td>
<td>src</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_NEG</td>
<td>HSA_BRIG_KIND_INST_BASIC (if only default modifiers are used) or HSA_BRIG_KIND_INST_MOD</td>
<td>dest</td>
<td>src</td>
<td></td>
</tr>
</tbody>
</table>

**dest:** must be HSA_BRIG_KIND_OPERAND_REGISTER.

**src:** must be HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, or HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST.

**cond:** must be HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST, or HSA_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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_NCOSE</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_NEXP2</td>
<td></td>
<td>dest</td>
<td>src</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_NFMA</td>
<td></td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_NLOG2</td>
<td></td>
<td>dest</td>
<td>src</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_NRCP</td>
<td></td>
<td>dest</td>
<td>src</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_NSQRT</td>
<td></td>
<td>dest</td>
<td>src</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_NSIN</td>
<td></td>
<td>dest</td>
<td>src</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_NSQRT2</td>
<td></td>
<td>dest</td>
<td>src</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
dest: must be HSA_BRIG_KIND_OPERAND_REGISTER.

src: must be HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, or HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST.

18.7.14 BRIG syntax for multimedia instructions

Table 18–17 BRIG syntax for multimedia instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_BITALIGN</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td>src</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_BYTEALIGN</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td>src</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE.Lerp</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td>src</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_PACKCVT</td>
<td>HSA_BRIG_KIND_INST_SOURCE_TYPE</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td>src</td>
<td>src</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_UNPACKCVT</td>
<td>HSA_BRIG_KIND_INST_SOURCE_TYPE</td>
<td>dest</td>
<td>src</td>
<td>number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_SAD</td>
<td>HSA_BRIG_KIND_INST_SOURCE_TYPE</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td>src</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_SADHI</td>
<td>HSA_BRIG_KIND_INST_SOURCE_TYPE</td>
<td>dest</td>
<td>src</td>
<td>src</td>
<td>src</td>
<td></td>
</tr>
</tbody>
</table>

dest: must be HSA_BRIG_KIND_OPERAND_REGISTER.

src: must be HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST, or HSA_BRIG_KIND_OPERAND_WAVESIZE.

number: must be HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES with value 0, 1, 2, or 3.

18.7.15 BRIG syntax for segment checking (segmentp) instruction

Table 18–18 BRIG syntax for segment checking (segmentp) instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
<th>Operand 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_SEGMENTP</td>
<td>HSA_BRIG_KIND_INST_SEG_CVT</td>
<td>dest</td>
<td>src</td>
</tr>
</tbody>
</table>

dest: must be HSA_BRIG_KIND_OPERAND_REGISTER.

src: must be HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST, or HSA_BRIG_KIND_OPERAND_WAVESIZE.

18.7.16 BRIG syntax for segment conversion instructions

Table 18–19 BRIG syntax for segment conversion instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
<th>Operand 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_FTOS</td>
<td>HSA_BRIG_KIND_INST_SEG_CVT</td>
<td>dest</td>
<td>src</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_STOF</td>
<td>HSA_BRIG_KIND_INST_SEG_CVT</td>
<td>dest</td>
<td>src</td>
</tr>
</tbody>
</table>

dest: must be HSA_BRIG_KIND_OPERAND_REGISTER.
src: must be HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST, or HSA_BRIG_KIND_OPERAND_WAVESIZE.

18.7.1.17 BRIG syntax for compare (cmp) instruction

Table 18–20 BRIG syntax for compare (cmp) instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
<th>Operand 1</th>
<th>Operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_CMP</td>
<td>HSA_BRIG_KIND_INST_CMP</td>
<td>dest</td>
<td>src</td>
<td>src</td>
</tr>
</tbody>
</table>

dest: must be HSA_BRIG_KIND_OPERAND_REGISTER.

src: must be HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST, or HSA_BRIG_KIND_OPERAND_WAVESIZE.

The pack field of HSA_BRIG_KIND_INST_CMP should be set to HSA_BRIG_PACK_PP for packed source types and to HSA_BRIG_PACK_NONE otherwise.

18.7.1.18 BRIG syntax for conversion (cvt) instruction

Table 18–21 BRIG syntax for conversion (cvt) instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
<th>Operand 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_CVT</td>
<td>HSA_BRIG_KIND_INST_CVT</td>
<td>dest</td>
<td>src</td>
</tr>
</tbody>
</table>

dest: must be HSA_BRIG_KIND_OPERAND_REGISTER.

src: must be HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST, or HSA_BRIG_KIND_OPERAND_WAVESIZE.

18.7.2 BRIG syntax for memory instructions

Table 18–22 BRIG syntax for memory instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_LD</td>
<td>HSA_BRIG_KIND_INST_MEM</td>
<td>reg-or-vector</td>
<td>address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_ST</td>
<td>HSA_BRIG_KIND_INST_MEM</td>
<td>reg-or-vector-or-num</td>
<td>address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_ATOMIC</td>
<td>HSA_BRIG_KIND_INST_ATOMIC</td>
<td>dest</td>
<td>address</td>
<td>src</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_ATOMIC (for atomic ld)</td>
<td>HSA_BRIG_KIND_INST_ATOMIC</td>
<td>dest</td>
<td>address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_ATOMIC (for atomic cas)</td>
<td>HSA_BRIG_KIND_INST_ATOMIC</td>
<td>dest</td>
<td>address</td>
<td>src</td>
<td>src</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_ATOMICOMICNORET</td>
<td>HSA_BRIG_KIND_INST_ATOMIC</td>
<td>address</td>
<td>src</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_SIGNAL</td>
<td>HSA_BRIG_KIND_INST_SIGNAL</td>
<td>dest</td>
<td>signal</td>
<td>src</td>
<td></td>
</tr>
</tbody>
</table>
18.7.3 BRIG syntax for image instructions

Table 18-23 BRIG syntax for image instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Oper. 0</th>
<th>Oper. 1</th>
<th>Oper. 2</th>
<th>Oper. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_RDIMAGE</td>
<td>HSA_BRIG_KIND_INST_IMAGE</td>
<td>reg-or-4-vector-reg</td>
<td>image</td>
<td>sampler</td>
<td>reg-or-vector-or-num</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_LDIMAGE</td>
<td>HSA_BRIG_KIND_INST_IMAGE</td>
<td>reg-or-4-vector-reg</td>
<td>image</td>
<td>reg-or-vector-or-num</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_STIMAGE</td>
<td>HSA_BRIG_KIND_INST_IMAGE</td>
<td>reg-or-4-vector-reg</td>
<td>image</td>
<td>reg-or-vector-or-num</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_QUERYIMAGE</td>
<td>HSA_BRIG_KIND_INST_QUERY_IMAGE</td>
<td>dest</td>
<td>image</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_QUERYSAMPLER</td>
<td>HSA_BRIG_KIND_INST_QUERY_SAMPLER</td>
<td>dest</td>
<td>sampler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_IMAGEFENCE</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**dest:** must be HSA_BRIG_KIND_OPERAND_REGISTER.

**reg-or-4-vector-reg:** must be HSA_BRIG_KIND_OPERAND_REGISTER; or HSA_BRIG_KIND_OPERAND_OPERAND_LIST that references a list of HSA_BRIG_KIND_OPERAND_REGISTER operands.

**image:** must be HSA_BRIG_KIND_OPERAND_REGISTER.
section: must be HSA_BRIG_KIND_OPERAND_REGISTER.

reg-or-vector-or-num: must be HSA_BRIG_KIND_OPERAND_REGISTER; HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES; HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION; HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST; HSA_BRIG_KIND_OPERAND_WAVESIZE, or HSA_BRIG_KIND_OPERAND_OPERAND_LIST that references a list of HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST, or HSA_BRIG_KIND_OPERAND_OPERAND_WAVESIZE operands.

### 18.7.4 BRIG syntax for branch instructions

Table 18-24 BRIG syntax for branch instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
<th>Operand 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_BR</td>
<td>HSA_BRIG_KIND_INST_BR</td>
<td>label</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_CBR</td>
<td>HSA_BRIG_KIND_INST_BR</td>
<td>condition</td>
<td>label</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_SBR</td>
<td>HSA_BRIG_KIND_INST_BR</td>
<td>index</td>
<td>labels</td>
</tr>
</tbody>
</table>

**Label**: must be HSA_BRIG_KIND_OPERAND_CODE_REF that references a HSA_BRIG_KIND_DIRECTIVE_LABEL directive in the same function scope.

**Condition**: must be HSA_BRIG_KIND_OPERAND_REGISTER for a c register, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST, or HSA_BRIG_KIND_OPERAND_WAVESIZE.

**Index**: must be HSA_BRIG_KIND_OPERAND_REGISTER for an s or d register according to the instruction type (which must be u32 or u64), HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST, or HSA_BRIG_KIND_OPERAND_WAVESIZE.

**Labels**: must be HSA_BRIG_KIND_OPERAND_CODE_LIST that references a list of HSA_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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_BARRIER</td>
<td>HSA_BRIG_KIND_INST_BR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_WAVEBARRIER</td>
<td>HSA_BRIG_KIND_INST_BR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_INITFBAR</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>fbarrier-or-reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_JOINFBAR</td>
<td>HSA_BRIG_KIND_INST_BR</td>
<td>fbarrier-or-reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_WAITFBAR</td>
<td>HSA_BRIG_KIND_INST_BR</td>
<td>fbarrier-or-reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_ARRIVEFBAR</td>
<td>HSA_BRIG_KIND_INST_BR</td>
<td>fbarrier-or-reg</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 18.7.6 BRIG syntax for function instructions

Table 18-26 BRIG syntax for instructions related to functions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_CALL</td>
<td>HSA_BRIG_KIND_INST_BR</td>
<td>out-args</td>
<td>func</td>
<td>in-args</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODESCALL</td>
<td>HSA_BRIG_KIND_INST_BR</td>
<td>out-args</td>
<td>src</td>
<td>in-args</td>
<td>funcas</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_ICALL</td>
<td>HSA_BRIG_KIND_INST_BR</td>
<td>out-args</td>
<td>reg</td>
<td>in-args</td>
<td>signature</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_RET</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_ALLOC</td>
<td>HSA_BRIG_KIND_INST_MEM</td>
<td>dest</td>
<td>src</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dest: must be HSA_BRIG_KIND_OPERAND_REGISTER.

src: must be HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST, or HSA_BRIG_KIND_OPERAND_WAVESIZE.

reg: must be HSA_BRIG_KIND_OPERAND_REGISTER.

out-args: output arguments; must be HSA_BRIG_KIND_OPERAND_CODE_LIST that references a list of HSA_BRIG_KIND_DIRECTIVE_VARIABLE directives with HSA_BRIG_SEGMENT_ARG segment in the same arg block.
in-args: input arguments; must be HSA_BRIG_KIND_OPERAND_CODE_LIST that references a list of HSA_BRIG_KIND_DIRECTIVE_VARIABLE directives with HSA_BRIG_SEGMENT_ARG segment in the same arg block.

**func:** must be HSA_BRIG_KIND_OPERAND_CODE_REF that references a HSA_BRIG_KIND_DIRECTIVE_FUNCTION or HSA_BRIG_KIND_DIRECTIVE_INDIRECT_FUNCTION directive.

**funcs:** must be HSA_BRIG_KIND_OPERAND_CODE_LIST that references a list of HSA_BRIG_KIND_DIRECTIVE_FUNCTION or HSA_BRIG_KIND_DIRECTIVE_INDIRECT_FUNCTION directives.

**signature:** must be HSA_BRIG_KIND_OPERAND_CODE_REF that references a HSA_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>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
<th>Operand 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_CURRENTWORKGROUPSIZE</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>dimNumber</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_CURRENTWORKITEMFLATID</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_DIM</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_GRIDGROUPS</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>dimNumber</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_GIRDGROUPS</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>dimNumber</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_PACKETCOMPLETIONSIG</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_PACKETID</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_PACKET_GROUPID</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>dimNumber</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_PACKETGROUPID</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>dimNumber</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_WORKITEMABSID</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>dimNumber</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_WORKITEMFLATABSID</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>dimNumber</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_WORKITEMID</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
<td>dimNumber</td>
</tr>
</tbody>
</table>

**dest:** must be HSA_BRIG_KIND_OPERAND_REGISTER.

**dimNumber:** must be HSA_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>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_CLEARDETECTEXCEPT</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>exceptionsNumber</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_GETDETECTEXCEPT</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_SETDETECTEXCEPT</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>exceptionsNumber</td>
</tr>
</tbody>
</table>

**dest:** must be HSA_BRIG_KIND_OPERAND_REGISTER.

**exceptionsNumber:** must be HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES. The value must be encoded according to hsa_brig_exceptions_t (see 18.3.9 hsa_brig_exceptions_t (on page 343)).
18.7.7.3 BRIG syntax for user mode queue instructions

Table 18–29 BRIG syntax for user mode queue instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_ADDQUEUEREINDEX</td>
<td>HSA_BRIG_KIND_INST_QUEUE</td>
<td>dest</td>
<td>address</td>
<td>src</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_CASQUEUEREINDEX</td>
<td>HSA_BRIG_KIND_INST_QUEUE</td>
<td>dest</td>
<td>address</td>
<td>src</td>
<td>src</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_LDQUEUEREADINDEX</td>
<td>HSA_BRIG_KIND_INST_QUEUE</td>
<td>dest</td>
<td>address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_LDQUEUEREWRIINDEX</td>
<td>HSA_BRIG_KIND_INST_QUEUE</td>
<td>dest</td>
<td>address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_STQUEUEREADINDEX</td>
<td>HSA_BRIG_KIND_INST_QUEUE</td>
<td>address</td>
<td>src</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_STQUEUEREWRIINDEX</td>
<td>HSA_BRIG_KIND_INST_QUEUE</td>
<td>address</td>
<td>src</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dest: must be HSA_BRIG_KIND_OPERAND_REGISTER.

src: must be HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST, or HSA_BRIG_KIND_OPERAND_WAVESIZE.

address: must be HSA_BRIG_KIND_OPERAND_ADDRESS.

18.7.7.4 BRIG syntax for miscellaneous instructions

Table 18–30 BRIG syntax for miscellaneous instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Format</th>
<th>Operand 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSA_BRIG_OPCODE_CLOCK</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_CUID</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_DEBUGTRAP</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>src</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_GROUPBASEPTR</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODEROUPSTATICSIZE</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_GROUPTOTALSIZE</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_KERNARGBASEPTR</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_LANEID</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_MAXCUID</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_MAXWAVEID</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_NOP</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td></td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_NULLPTR</td>
<td>HSA_BRIG_KIND_INST_SEG</td>
<td>dest</td>
</tr>
<tr>
<td>HSA_BRIG_OPCODE_WAVEID</td>
<td>HSA_BRIG_KIND_INST_BASIC</td>
<td>dest</td>
</tr>
</tbody>
</table>

dest: must be HSA_BRIG_KIND_OPERAND_REGISTER.

src: must be HSA_BRIG_KIND_OPERAND_REGISTER, HSA_BRIG_KIND_OPERAND_CONSTANT_BYTES, HSA_BRIG_KIND_OPERAND_CONSTANT_EXPRESSION, HSA_BRIG_KIND_OPERAND_CONSTANT_OPERAND_LIST, or HSA_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_LITERAL ::= decimalIntegerLiteral
    | hexIntegerLiteral
    | octalIntegerLiteral

decimalIntegerLiteral ::= "0" { '1-9' { '0-9' } }

hexIntegerLiteral ::= "0" { "x" | "X" } '0-9a-fA-F' { '0-9a-fA-F' }
```
octalIntegerLiteral ::= "0" '[0-7]' { '[0-7]' }

TOKEN_HALF_LITERAL ::= decimalFloatLiteral { "h" | "H" }
                        | hexFloatLiteral { "h" | "H" }
                        | ieeeHalfLiteral

TOKEN_SINGLE_LITERAL ::= decimalFloatLiteral { "f" | "F" }
                        | hexFloatLiteral { "f" | "F" }
                        | ieeeSingleLiteral

TOKEN_DOUBLE_LITERAL ::= decimalFloatLiteral { "d" | "D" }
                        | hexFloatLiteral { "d" | "D" }
                        | ieeeDoubleLiteral

decimalFloatLiteral ::= ( ( '[0-9]' { '[0-9]' } ) "." 
                        | ( [ "e" | "E" ] [ "e" | "E" ] [0-9] ) )
                        | ( [ "e" | "E" ] [ "e" | "E" ] )
                        | ( [ "e" | "E" ] [ "e" | "E" ] )
                        | ( [ "e" | "E" ] [ "e" | "E" ] )

hexFloatLiteral ::= "0" ( "x" | "X" )
                       ( [0-9a-fA-F]{4} )
                        | "0" [0-9a-fA-F]
                        | "0" [0-9a-fA-F]
                      | "0" [0-9a-fA-F]

ieeeHalfLiteral ::= "0" ( "h" | "H" ) [0-9a-fA-F]
 ieeeSingleLiteral ::= "0" ( "f" | "F" ) [0-9a-fA-F]
 ieeeDoubleLiteral ::= "0" ( "d" | "D" ) [0-9a-fA-F]

TOKEN_WAVESIZE ::= "WAVESIZE"

TOKEN_STRING_LITERAL ::= "";
                        | "not (\" or new-line)"
                        | "\";
                        | "\";
                        | "\";
                        | "\";
                        | "\";
                        | "\";

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
  | pad
  | pragma
location ::= "loc"
  | TOKEN_INTEGER_LITERAL
  | TOKEN_STRING_LITERAL
  | pragmaOperand
  | "\n"
  | ":" pad
  | ";"
  | pragmaOperand
pad ::= "pad" [ TOKEN_INTEGER_LITERAL ] ";"
pragma ::= "pragma" TOKEN_STRING_LITERAL { "," pragmaOperand } ";"
moduleHeader ::= "module"
  | TOKEN_GLOBAL_IDENTIFIER ":" TOKEN_INTEGER_LITERAL ":" TOKEN_INTEGER_LITERAL ":"
  | TOKEN_INTEGER_LITERAL ":"
profile ::= "$full"
  | "$base"
machineModel ::= "$small"
  | "$large"
defaultFloatRounding ::= "$default"
  | "$zero"
  | "$near"
moduleDirective ::= "extension"
extension ::= "extension" TOKEN_STRING_LITERAL
  | TOKEN_INTEGER_LITERAL ":" TOKEN_INTEGER_LITERAL ":"
moduleStatement ::= moduleVariable
  | moduleFbarrier
  | kernel
  | function
  | signature
moduleVariable ::= optDeclQual linkageQual variable
variable ::= optAllocQual optAlignQual optConstQual variableSegment
  | dataTypeMod nonRegisterIdentifier optArrayDimension
optInitializer ::= [ "=" initializerConstant ]
initializerConstant ::= integerConstant
  | floatConstant
  | typedConstant
  | aggregateConstant
aggregateConstant ::= [ "( " aggregateConstantItem " , " aggregateConstantItem " ) "
aggregateConstantItem ::= typedConstant
  | aggregateConstantAlign
  | aggregateConstantZero
aggregateConstantAlign ::= [ "align" "( TOKEN_INTEGER_LITERAL )" ]
aggregateConstantZero ::= [ "zero" "( TOKEN_INTEGER_LITERAL )" ]
typedConstant ::= integerTypedConstant
  | floatTypedConstant
  | packedTypedConstant
  | imageTypedConstant
  | samplerTypedConstant
  | signalTypedConstant
  | arrayTypedConstant
integerTypedConstant ::= integerType [ "(" integerConstant ")" ]
integerConstant ::= [ "+" | "-" ] TOKEN_INTEGER_LITERAL
  | "addr" [ "(" TOKEN_GLOBAL_IDENTIFIER ":" TOKEN_INTEGER_LITERAL ]
  | "nullptr" [ "_group" | "_private" | "_kernarg" ]
floatTypedConstant ::= "f16" "{ halfConstant }")"  
 | "f32" "{ singleConstant }")"  
 | "f64" "{ doubleConstant }")"  
floatConstant ::= halfConstant  
 | singleConstant  
 | doubleConstant  
halfConstant ::= [ "+" | "-" ] TOKEN_HALF_LITERAL  
singleConstant ::= [ "+" | "-" ] TOKEN_SINGLE_LITERAL  
doubleConstant ::= [ "+" | "-" ] TOKEN_DOUBLE_LITERAL  
packedTypedConstant ::= integerPackedType  
 | halfPackedType  
 | singlePackedType  
 | doublePackedType  
imageTypedConstant ::= imageType "{ imageProperty "," } imageProperty "}"  
imageProperty ::= "geometry" =~ imageGeometry  
 | "width" =~ TOKEN_INTEGER_LITERAL  
 | "height" =~ TOKEN_INTEGER_LITERAL  
 | "depth" =~ TOKEN_INTEGER_LITERAL  
 | "array" =~ TOKEN_INTEGER_LITERAL  
 | "channel_order" =~ imageChannelOrder  
 | "channel_type" =~ imageChannelType  
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"  
 | "x"  
 | "rg"  
 | "rgx"  
 | "ra"  
 | "rgb"  
 | "rgbx"  
 | "rgba"  
 | "bgra"  
 | "argb"  
 | "abgr"  
 | "srgb"  
 | "srgbx"  
 | "srgba"
functionHeader ::= functionHeader | "function" TOKEN_GLOBAL_IDENTIFIER kernFormalArgumentList
kernFormalArgumentList ::= [ { 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 ::= optAlignQual "arg" dataTypeMod
TOKEN_LOCAL_IDENTIFIER optArrayDimension

moduleFbarrier ::= optDeclQual linkageQual fbarrier
fbarrier ::= "fbarrier" nonRegisterIdentifier ;
kern ::= declQual linkageQual kernelHeader |
| linkageQual kernelHeader codeBlock ;
kernHeader ::= "kernel" TOKEN_GLOBAL_IDENTIFIER kernFormalArgumentList
packedArrayTypedConstant ::= packedType [ ]
| packedTypedConstant , |
packedArrayTypedConstant ::= packedType [ ]
| packedTypedConstant , |
imageArrayTypedConstant ::= imagePath [ ]
| imageTypedConstant , |
samplerArrayTypedConstant ::= samplerType [ ]
| samplerTypedConstant , |
signalArrayTypedConstant ::= signalType [ ]
| signalTypedConstant , |
arrayTypedConstant ::= integerArrayType [ ]
| halfArrayType [ ]
| f16ArrayType [ ]
| f32ArrayType [ ]
| f64ArrayType [ ]
| doubleArrayType [ ]
| singleArrayType [ ]
| halfArrayType [ ]
| integerArrayType [ ]
imageTypedConstant ::= imagePath [ ]
| imagePath , |
samplerTypedConstant ::= samplerType [ ]
| samplerType , |
signalTypedConstant ::= signalType [ ]
| signalType , |
samplerProperty ::= samplerAddressing |
| samplerFilter |
| samplerCoord |
samplerAddressing ::= "undefined" |
| "repeat" |
| "nearest" |
| "clamp_to_edge" |
| "clamp_to_border" |
samplerFilter ::= "normalized" |
| "unnormalized" |
samplerCoord ::= "nearest" |
| "linear" |
signature ::= signatureType [ ]
| signatureType , |
signalType ::= signalTypedConstant |
| samplerTypedConstant |
| imageTypedConstant |
| packedTypedConstant |
| doubleConstant |
| singleConstant |
| halfConstant |
| integerConstant |
signatureType ::= signatureType |
| signatureType , |
signalTypedConstant ::= signalType [ ]
| signalType , |

kernel ::= declQual linkageQual kernelHeader |
| linkageQual kernelHeader codeBlock |
samplerAddressing ::= samplerAddressing |
| samplerFilter |
| samplerCoord |
samplerFilter ::= "normalized" |
| "unnormalized" |
samplerCoord ::= "nearest" |
| "linear" |
signature ::= signatureType [ ]
| signatureType , |
signalType ::= signalTypedConstant |
| samplerTypedConstant |
| imageTypedConstant |
| packedTypedConstant |
| doubleConstant |
| singleConstant |
| halfConstant |
| integerConstant |
signatureType ::= signatureType |
| signatureType , |
signalTypedConstant ::= signalType [ ]
| signalType , |
samplerProperty ::= samplerAddressing |
| samplerFilter |
| samplerCoord |
samplerAddressing ::= samplerAddressing |
| samplerFilter |
| samplerCoord |
samplerFilter ::= "normalized" |
| "unnormalized" |
samplerCoord ::= "nearest" |
| "linear" |
signature ::= signatureType [ ]
| signatureType , |
signalType ::= signalTypedConstant |
| samplerTypedConstant |
| imageTypedConstant |
| packedTypedConstant |
| doubleConstant |
| singleConstant |
| halfConstant |
| integerConstant |
signatureType ::= signatureType |
| signatureType , |
signalTypedConstant ::= signalType [ ]
| signalType , |
samplerProperty ::= samplerAddressing |
| samplerFilter |
| samplerCoord |
samplerAddressing ::= samplerAddressing |
| samplerFilter |
| samplerCoord |
samplerFilter ::= "normalized" |
| "unnormalized" |
samplerCoord ::= "nearest" |
| "linear" |
signature ::= signatureType [ ]
| signatureType , |
signalType ::= signalTypedConstant |
| samplerTypedConstant |
| imageTypedConstant |
| packedTypedConstant |
| doubleConstant |
| singleConstant |
| halfConstant |
| integerConstant |
signatureType ::= signatureType |
| signatureType , |
signalTypedConstant ::= signalType [ ]
| signalType , |
samplerProperty ::= samplerAddressing |
| samplerFilter |
| samplerCoord |
samplerAddressing ::= samplerAddressing |
| samplerFilter |
| samplerCoord |
samplerFilter ::= "normalized" |
| "unnormalized" |
samplerCoord ::= "nearest" |
| "linear" |
signature ::= signatureType [ ]
| signatureType , |
signalType ::= signalTypedConstant |
| samplerTypedConstant |
| imageTypedConstant |
| packedTypedConstant |
| doubleConstant |
| singleConstant |
| halfConstant |
| integerConstant |
signature ::= "signature" TOKEN_GLOBAL_IDENTIFIER
    sigOutputFormalArgumentList sigInputFormalArgumentList ;
sigOutputFormalArgumentList ::= sigFormalArgumentList
sigInputFormalArgumentList ::= sigFormalArgumentList
sigFormalArgument ::= (" [ { sigFormalArgument ," } sigFormalArgument ] ")
linkageQual ::= [ "prog" ]
optDeclQual ::= { declQual }
declQual ::= "decl"
optConstQual ::= [ "const" ]
optAlignQual ::= [ "align" "( TOKEN_INTEGER_LITERAL )" ]
allocKind ::= "agent"
allocationKind ::= "[ [ TOKEN_INTEGER_LITERAL ] ]"
codeBlock ::= "{" annotations
    { codeBlockDirective annotations }
    { codeBlockDefinition annotations }
    { codeBlockStatement annotations }
    ""}
codeBlockDirective ::= control
codeBlockVariable ::= variable
codeBlockFbarrier ::= fbarrier
codeBlockStatement ::= argBlock
    label
    instruction
argBlock ::= ":{ " annotations
    { argBlockDefinition annotations }
    { argBlockStatement annotations }
    ""}
arbBlockDefinition ::= argBlockVariable
argBlockVariable ::= variable
argBlockStatement ::= label
    instruction
    call
label ::= TOKEN_LABEL_IDENTIFIER ";"
instruction ::= instruction0
    instruction1
    instruction2
    instruction3
    instruction4
    mul
    bitinsert
Chapter 19. HSAIL Grammar in Extended Backus-Naur Form

19.2 HSAIL syntax grammar in Extended Backus-Naur Form (EBNF)

| combine |
| expand |
| lda |
| mov |
| pack |
| unpack |
| packcvt |
| unpackcvt |
| sad |
| segmentConversion |
| cmp |
| cvt |
| ld |
| st |
| atomic |
| atomicnoret |
| signal |
| signalnoretn |
| memfence |
| rdimage |
| stimage |
| ldimage |
| queryimage |
| querysampler |
| branch |
| barrier |
| wavebarrier |
| fbar |
| crossLane |
| ret |
| alloca |
| packetcompletionsig |
| queue |

instruction0 ::= ( "nop" |
                  "imagefence" ) |
";"

instruction1 ::= ( instruction1Opcode optRoundingMod nonOpaqueTypeMod |
                  "nullptr" optSegmentMod nonOpaqueTypeMod |
                  "nullptr" optSegmentMod nonOpaqueTypeMod |
                  operand |
                  ) ; ;

instruction1Opcode ::= "cleardetectexcept" |
                         "clock" |
                         "cuid" |
                         "debugtrap" |
                         "dim" |
                         "getdetectexcept" |
                         "groupbaseptr" |
                         "groupstaticsize" |
                         "grouptotalsize" |
                         "kernargbaseptr" |
                         "laneid" |
                         "maxcuid" |
                         "maxwaveid" |
                         "packetid" |
                         "setdetectexcept" |
                         "waveid" |
                         "workitemflatabsid" |
                         "workitemflatid"

instruction2 ::= ( instruction2Opcode optRoundingMod optPackingMod |
                  instruction2OpcodeFtz optFtzMod optPackingMod |
                  instruction2OpcodeFtz optFtzMod nonOpaqueTypeMod |
                  "popcount" nonOpaqueTypeMod |
                  "firstbit" nonOpaqueTypeMod |
                  "lastbit" nonOpaqueTypeMod |
                  operand |
                  ) ; ;
nonOpaqueTypeMod operand "," operand ";"

instruction2 Opcode ::= "abs"
| "bitrev"
| "currentworkgroupsize"
| "currentworkitemflatid"
| "fract"
| "ncos"
| "neg"
| "nexp2"
| "nlog2"
| "nrcp"
| "nrsqrt"
| "nsin"
| "sqrt"
| "workgroupid"
| "workgroupsize"
| "workitemabsid"
| "workitemid"

instruction2 Opcode Ftz ::= "ceil"
| "floor"
| "rint"
| "trunc"

instruction3 ::= ( instruction3 Opcode optRoundingMod optPackingMod optIntegerSatmod
| instruction3 Opcode Ftz optFtzMod optPackingMod
| "class" nonOpaqueTypeMod
)
nonOpaqueTypeMod operand "," operand "," operand ";"

instruction3 Opcode ::= "add"
| "bitmask"
| "borrow"
| "carry"
| "copysign"
| "div"
| "rem"
| "sub"
| "shl"
| "shr"
| "and"
| "or"
| "xor"
| "unpackhi"
| "unpacklo"

instruction3 Opcode Ftz ::= "max"
| "min"

instruction4 ::= ( instruction4 Opcode optRoundingMod optIntegerSatmod
| instruction4 Opcode Ftz optFtzMod optPackingMod
)
nonOpaqueTypeMod operand "," operand "," operand "," operand ";"

instruction4 Opcode ::= "fma"
| "mad"
| "bitextract"
| "bitselect"
| "shuffle"
| "cmov"
| "bitalign"
| "bytealign"
| "lerp"

instruction4 Opcode Ftz ::= "nfma"

mul ::= ( "mul" optRoundingMod optPackingMod nonOpaqueTypeMod
| "float" optRoundingMod optPackingMod nonOpaqueTypeMod
| "int" optRoundingMod optPackingMod nonOpaqueTypeMod
| "varint" optRoundingMod optPackingMod nonOpaqueTypeMod
| "complex" optRoundingMod optPackingMod nonOpaqueTypeMod
| "vector" optRoundingMod optPackingMod nonOpaqueTypeMod
| "aligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "unaligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "aligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "unaligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "aligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "unaligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "aligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "unaligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "aligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "unaligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "aligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "unaligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "aligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "unaligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "aligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "unaligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "aligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "unaligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "aligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "unaligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "aligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "unaligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "aligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "unaligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "aligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "unaligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "aligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "unaligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "aligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "unaligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "aligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "unaligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "aligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "unaligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "aligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "unaligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "aligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "unaligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "aligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "unaligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "aligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "unaligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "aligned" optRoundingMod optPackingMod nonOpaqueTypeMod
| "unaligned" optRoundingMod optPackingMod nonOpaqueTypeMo
| "mulhi" optPackingMod nonOpaqueTypeMod
| "mul24hi" nonOpaqueTypeMod
| "mul24" nonOpaqueTypeMod
| "mad24" nonOpaqueTypeMod operand ","
| "mad24hi" nonOpaqueTypeMod operand ","
operand "," operand "," operand ";"

bitinsert ::= "bitinsert" nonOpaqueTypeMod operand "," operand "," operand "," operand ";"

combine ::= "combine" vectorMod nonOpaqueTypeMod nonOpaqueTypeMod operand "," vectorOperand "," operand "," operand ";"

expand ::= "expand" vectorMod nonOpaqueTypeMod nonOpaqueTypeMod vectorOperand "," operand "," operand ";"

lda ::= "lda" optSegmentMod nonOpaqueTypeMod operand "," memoryOperand ",";

mov ::= "mov" dataTypeMod operand "," operand ";"

pack ::= "pack" nonOpaqueTypeMod nonOpaqueTypeMod operand ",," operand ",," operand "," operand ";"

unpack ::= "unpack" nonOpaqueTypeMod nonOpaqueTypeMod operand ",," operand ",," operand "," operand ";"

packcvt ::= "packcvt" nonOpaqueTypeMod nonOpaqueTypeMod operand ",," operand ",," operand "," operand "," operand ";"

unpackcvt ::= "unpackcvt" nonOpaqueTypeMod nonOpaqueTypeMod operand ",," operand ",," operand "," operand ";"

sad ::= ("sad" | "sadhi") nonOpaqueTypeMod nonOpaqueTypeMod operand ",," operand ",," operand "," operand ";"

segmentConversion ::= ("segmentp" | "ftos" | "stof") segmentMod optNullMod nonOpaqueTypeMod nonOpaqueTypeMod operand ",," operand "," operand ",";

cmp ::= "cmp" comparisonOp optFtzMod optPackingMod nonOpaqueTypeMod operand ",," operand ",," operand ";"

cmpOp ::= "eq"
| "ne"
| "lt"
| "le"
| "gt"
| "ge"
| "ueq"
| "ue"n
| "ltu"
| "leu"
| "gtu"
| "geu"
| "num"
| "nan"
| "seq"
| "sne"
| "slt"
| "sle"
| "sgt"
| "sgle"
| "snum"
| "snan"
| "sequ"
| "sneu"
| "sltu"
| "sleu"
| "sgtu"
| "sgtuu"

cvt ::= "cvt" optCvtRoundingMod optPackingMod optIntegerSatmod nonOpaqueTypeMod
| nonOpaqueTypeMod operand ",," operand ",";

optCvtRoundingMod ::= [ cvtRoundingMod ]
cvtRoundingMod ::= floatRoundingMod
| "ftz"
| "_ftz" floatRoundingMod 
| integerRoundingMod
| "_ftz" integerRoundingMod

ld ::= "ld" optVectorMod optSegmentMod
optAlignMod optConstMod optEquivMod optWidthMod optNTMod dataTypeMod
possibleVectorOperand "," memoryOperand ";"

st ::= "st" optVectorMod optSegmentMod
optAlignMod optEquivMod optNTMod 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
) ";"

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
| "_wait"
  waitOp memOrderMod nonOpaqueTypeMod signalTypeMod operand "," operand "," operand
| "_waittimeout"
  waitOp memOrderMod nonOpaqueTypeMod signalTypeMod operand "," operand "," operand
) ";"

signalnoret ::= "signalnoret"
( signalOp
  memOrderMod nonOpaqueTypeMod signalTypeMod operand "," operand
| "_st"
  stMemOrderMod nonOpaqueTypeMod signalTypeMod operand "," operand
)


Chapter 19. HSAIL Grammar in Extended Backus-Naur Form  19.2 HSAIL syntax grammar in Extended Backus-Naur Form (EBNF)


signalOp ::= "add" | "and" | "exch" | "oc" | "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 operand "," operand "," operand "," possibleVectorOperand ";"
ldimage ::= "ldimage" [ "v4" ] geometryMod optEquivMod nonOpaqueTypeMod imageTypeMod operand "," operand "," possibleVectorOperand ";"
stimage ::= "stimage" [ "v4" ] geometryMod optEquivMod nonOpaqueTypeMod imageTypeMod operand "," 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 ";" | "cbz" optWidthMod nonOpaqueTypeMod TOKEN_REGISTER "," TOKEN_LABEL_IDENTIFIER ";" | "sbc" optWidthMod nonOpaqueTypeMod operand branchTargets ";"
branchTargets ::= [ \[ { TOKEN_LABEL_IDENTIFIER "," } TOKEN_LABEL_IDENTIFIER ] ]
barrier ::= "barrier" optWidthMod ";"
wavebarrier ::= "wavebarrier" ";"
fbar ::= "initfbar" operand ";" | "joinfbar" optWidthMod operand ";" | "waitfbar" optWidthMod operand ";" | "arrivefbar" optWidthMod operand ";" | "leavefbar" optWidthMod operand ";" | "releasefbar" operand ";"
19.2 HSAIL Syntax Grammar in Extended Backus-Naur Form (EBNF)

```
crossLane ::= "activelaneid" optWidthMod nonOpaqueTypeMod operand ";"
            | "activelanecount" optWidthMod nonOpaqueTypeMod nonOpaqueTypeMod operand ";"
            | "activelanemask" "_v4" optWidthMod nonOpaqueTypeMod nonOpaqueTypeMod vectorOperand ";" operand ";"
            | "activelanepermute" optWidthMod nonOpaqueTypeMod operand ";" operand ";" operand ";" operand ";"
call ::= "call" TOKEN_GLOBAL_IDENTIFIER callOutputActualArguments callInputActualArguments ";"
        | "scal" optWidthMod nonOpaqueTypeMod operand callOutputActualArguments callInputActualArguments callTargets ";"
        | "icall" optWidthMod nonOpaqueTypeMod operand callOutputActualArguments callInputActualArguments TOKENLOBAL_IDENTIFIER ";"
callOutputActualArguments ::= callActualArguments
callInputActualArguments ::= callActualArguments
callActualArguments ::= "[" operandList "]"
callTargets ::= "[" { TOKENLOBAL_IDENTIFIER "," } TOKENLOBAL_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 ";"
        | ( "ldqueuereadindex" | "ldqueuewriteindex" ) segmentMod memOrderMod nonOpaqueTypeMod operand "," memoryOperand ";"
        | ( "stqueuereadindex" | "stqueuewriteindex" ) segmentMod memOrderMod nonOpaqueTypeMod memoryOperand "," operand ";"
pragmaOperand ::= TOKEN_STRING_LITERAL
        | immediateOperand
        | aggregateConstant
        | identifierOperand
        | TOKEN_LABEL_IDENTIFIER
operand ::= immediateOperand
        | identifierOperand
immediateOperand ::= integerConstant
        | floatConstant
        | typedConstant
        | TOKEN_WAVESIZE
memoryOperand ::= symbolicAddressableOperand
        | offsetAddressableOperand
symbolicAddressableOperand ::= "[" nonRegisterIdentifier "]"
        | symbolicAddressableOperand offsetAddressableOperand
offsetAddressableOperand ::= "[" registerIdentifier "+" TOKEN_INTEGER_LITERAL "]"
        | "[" registerIdentifier "-" TOKEN_INTEGER_LITERAL "]"
        | "[" registerIdentifier "]"
        | "[" TOKEN_INTEGER_LITERAL "]"
        | "[" "+" TOKEN_INTEGER_LITERAL "]"
        | "[" "-" TOKEN_INTEGER_LITERAL "]"
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_LITERAL ")" ]

optConstMod ::= [ "_const" ]

optEquivMod ::= [ "_equiv" "(" TOKEN_INTEGER_LITERAL ")" ]

optNullMod ::= [ "_nonnull" ]

optWidthMod ::= [ "_width" "("
                           { "all"
                           | TOKEN_WAVESIZE
                           | TOKEN_INTEGER_LITERAL
                           ) ")" ]

optNTMod ::= [ "_nt" ]

optVectorMod ::= [ vectorMod ]

vectorMod ::= "_v2"
            | "_v3"
            | "_v4"

optRoundingMod ::= optFtzMod [ floatRoundingMod ]

optFtzMod ::= [ "_ftz" ]

floatRoundingMod ::= "_up"
                  | "_down"
                  | "_zero"
                  | "_near"

integerRoundingMod ::= "_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"

optIntegerSatMod ::= ["sat"]

dataTypeMod ::= 
| "_" dataType
| packedType 
| opaqueType

dataType ::= baseType
| packedType 
| opaqueType 

nonOpaqueTypeMod ::= 
| "_" nonOpaqueType

nonOpaqueType ::= baseType
| packedType 
| opaqueType

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"
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 of HSAIL must support. See HSA Platform System Architecture Specification Version 1.2, Appendix A Limits for additional system architecture limits.

- **c registers**: The c registers in HSAIL are a single pool of resources per function scope. It is an error if the value \((c_{\text{max}} + 1)\) exceeds 128 for any kernel or function definition, where \(c_{\text{max}}\) 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 \(c_{\text{max}}\) 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 \((s_{\text{max}} + 1) + 2*(d_{\text{max}} + 1) + 4*(q_{\text{max}} + 1)\) exceeds 2048 for any kernel or function definition, where \(s_{\text{max}}\), \(d_{\text{max}}\), and \(q_{\text{max}}\) 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 \(s_{\text{max}}\) 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.

- **Number of fbarriers**: Every kernel agent of an implementation must support at least 32 fbarriers per work-group.

- **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 (on page 263)).
APPENDIX B.
Glossary of HSAIL Terms

acquire synchronizing operation
A memory operation that specifies an acquire memory ordering. See 6.2.1 Memory order (on page 183).

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? (on page 21).

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 runtime executables into which zero or more HSA runtime code objects have been loaded for zero or more kernel agents. See 4.2 Program, code object, and executable (on page 50).

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, kernel code handle, 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 operations visible. AQL packets are queued on user mode queues. See HSA Platform System Architecture Specification Version 1.2, 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 (on page 32) and 10.2 Function call argument passing (on page 263).

BRIG
The HSAIL binary format. See Chapter 18 BRIG: HSAIL Binary Format (on page 337).

code object
An HSAIL program can be finalized to produce two kinds of HSA runtime code objects. A program code object contains the definitions of program scope global segment variables. An agent code object contains the definitions of agent allocation global and readonly segment variables and the machine code for all kernels for a specific instruction set architecture. A code object can be loaded into an HSA runtime executable. See 4.2 Program, code object, and executable (on page 50).

compound type
A type made up of a base data type and a length. See 4.13.1 Base data types (on page 108).
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 (on page 24).

communication operations
Operations that involve potential information transfer between threads of execution. See 2.12 Divergent control flow (on page 42).

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 (on page 28).

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 (on page 24).

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 (on page 42).

executable
An HSA runtime executable manages the allocation of global and readonly segment variables defined by HSA runtime code objects, manages installing machine code from agent code objects into kernel agents, and manages linking references to global and readonly segment variables in one code object to definitions in another code object or in the host application. An application can use the HSA runtime to create zero or more HSA runtime executables, to which zero or more code objects can be loaded. The kernels defined by agent code objects can then be executed on the kernel agent on which they have been installed. See 4.2 Program, code object, and executable (on page 50).

extension
An HSAIL operation specific to a finalizer. Extensions are enabled by the extension directive and accessed like all HSAIL instructions. See 13.1 extension directive (on page 293).

fbarrier
A fine-grain barrier that applies to a subset of a work-group. See 9.2 Fine-grain barrier (fbarrier) instructions (on page 249).

finalizer
A finalizer is part of the optional HSA runtime finalizer extension and translates HSAIL code in the form of BRIG into HSA runtime code objects. When an application uses the HSA runtime it can optionally include the finalizer extension.

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 (on page 28).

global segment
A memory segment in which memory is visible to all units of execution in all agents. See 2.8.1 Types of segments (on page 32).

grid
A multidimensional, rectangular structure containing work-groups. A grid is formed when a program launches a kernel. See 1.2 HSAIL virtual language (on page 22).
group segment
A memory segment in which memory is visible to a single work-group. See 2.8.1 Types of segments (on page 32).

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 operations 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? (on page 21).

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 Version 1.2.

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 (on page 50) and 4.3 Module (on page 57).

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 runtime executable must be used to provide the definition, such as to a host application variable. See 4.2 Program, code object, and executable (on page 50) and 4.3 Module (on page 57).

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 (on page 227).
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 tables attributes. If a kernel agent accesses an invalid address, system software shall be notified. See HSA Platform System Architecture Specification Version 1.2, 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 (on page 32).

kernel
A section of code executed in a data-parallel way by a kernel agent. Kernels are written in HSAIL and are translated by a finalizer to machine code. See 1.1 What is HSAIL? (on page 21).

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 operations to construct and enqueue AQL packets. A kernel agent is composed of one or more compute units. See 1.1 What is HSAIL? (on page 21).

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 (on page 29).

loader
A loader is part of the HSA runtime and can load HSA runtime code objects into HSA runtime executables. 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 agent code objects that have been loaded onto a kernel agent that is part of the HSA system.

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 (on page 107).

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) (on page 120).

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 (on page 73).
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 (on page 32).

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 runtime executable. See 4.12.1 Program linkage (on page 106).

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 (on page 32).

release synchronizing operation
A memory operation that specifies a release memory ordering. See 6.2.1 Memory order (on page 183).

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 (on page 232).

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 (on page 32).

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 (on page 202).

spill segment
A memory segment used to load or store register spills. See 2.8.1 Types of segments (on page 32).

texel
An image access specifies a filter mode that can result in more than one image element being accessed: these elements are known as texels. See 7.1.6.3 Filter mode (on page 225).
unit of execution
A unit of execution is a program-ordered sequence of operations through a processing element. A unit of execution can be any thread of execution on an agent, a work-item, or any method of sending operations through a processing element in an HSA-compatible device.

Unit of Least Precision (ULP)
A concept used in determining the accuracy of a floating-point operation. See 4.19.6 Unit of least precision (ULP) (on page 121).

uniform instruction
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 (on page 42).

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 1.2, section 2.8 Requirement: User mode queuing.

wavefront
A group of work-items executing on a single program counter. See 2.6 Wavefronts, lanes, and wavefront sizes (on page 29).

WAVESIZE
An implementation defined constant specifying the size of a wavefront for a kernel agent. See 2.6.2 Wavefront size (on page 31) and 2.6 Wavefronts, lanes, and wavefront sizes (on page 29).

work-group
A work-group is a partitioning of the grid of work-items formed by a kernel dispatch. It is an instance of execution in a compute unit. See 2.2 Work-groups (on page 26).

work-group ID
The identifier of a work-group expressed in three dimensions. See 2.2.1 Work-group ID (on page 26).

work-group flattened ID
The work-group ID flattened into one dimension. See 2.2.2 Work-group flattened ID (on page 27).

work-item
A single unit of execution of the grid formed by a kernel dispatch. See 2.3 Work-items (on page 27).

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 (on page 28).

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 (on page 28).

work-item flattened absolute ID
The work-item absolute ID flattened into one dimension. See 2.3.4 Work-item flattened absolute ID (on page 28).
work-item ID
   The identifier of a work-item (within the work-group) expressed in three dimensions. See 2.3.1 Work-item ID (on page 27).

write atomicity
   A condition of a store such that it must be written in its entirety.
Index

2
24-bit integer optimization instructions 133, 393
 mad24 134, 413
 mad24hi 134, 413
 mul24 134, 320-321, 413
 mul24hi 134, 413

A
acquire memory order 421
active work-group 29, 421
active work-item 29, 44-45, 189, 421
addressing mode 214, 222-227, 232-234, 392
agent allocation 185
agent threads 41
aggregate constant 99-100
application program 421
Architected Queuing Language (AQL) 21, 25, 421, 423-424
argument scope 78, 83, 108, 262-264, 275, 370, 420
argument scope arg segment 55
arguments 335
arithmetic instructions 118, 127
atomic memory instruction 184-187, 194-195, 200, 284, 345

B
base data type 108, 110, 421
base profile 47, 90, 109, 112, 117, 119-120, 141-142, 144, 154-155, 175, 178, 187, 191-192, 324, 327-328
basic type 308
BasicType (pragma) 308
bit conditional move instruction 151, 395
cmov 151, 395
bit string instructions 138
 bitextract 139
 bitinsert 139
 bitmask 139
 bitrev 140
 bitselect 140
 firstbit 140
 lastbit 140
bits per pixel 213, 219
branch instructions 42-43, 45-46, 103, 113, 143, 246-247, 415
brn 246
cbr 42-43, 45-46, 246, 415
sbr 42, 45, 103, 143, 247, 415
BREAK 121, 290, 292, 318, 335-336
BRIG binary format 50, 101, 213, 295, 337, 339, 351, 355, 362, 392, 403, 421

C
call instruction 235
channel order 210, 213-216, 219-221, 224-225, 391
channel type 210, 213, 216, 219-221, 229, 234, 391
clock special instruction 44, 284
closed interval 215
code object 421
communication operations 422
compare instructions 84, 166-167, 327-328, 374, 411, 413
cmp 144, 166-167, 327-328, 411, 413
compilation unit 304
compile-time macro 31, 286
CompileUnit (pragma) 304
composite type 310
CompositeType (pragma) 310
compound type 108-109, 114, 137, 145-147, 187, 253, 352, 421
calculate 335
compute unit 25-26, 40, 249, 286-287, 290, 421-422, 424
Constu (pragma) 316
control 410
control (c) register 84, 136, 158
control directive 25, 65-66, 259, 281, 290, 317-322
enablebreakexceptions 318
enabledetectexceptions 290, 319, 410
maxdynamicgroupszie 319
maxflatgridsize 320
maxflatworkgroupsize 320
requireddim 320
requiredgridsize 321
requiredgroupbasepseudalign 321
requiredworkgroupsize 321
requiredpartialwavefronts 322
requiredpartialworkgroups 322
control flow 246, 264
control flow divergence 247, 271, 273, 332
control flow optimization 332
conversion instructions 109, 115, 171, 187, 190, 194, 289, 327-330, 374, 411, 413
cvt 115, 120, 171, 187, 190, 194, 289, 327-330, 413
coordinate normalization mode 222, 227, 232
copy instructions 35, 75-78, 113, 126, 130, 141-143, 171, 202, 235-236, 266, 295, 411, 413
combine instruction 142
expand instruction 142
lda 35, 75-76, 78, 113, 126, 142-143, 202, 236, 266, 411, 413
mov 143, 171, 202, 235, 295, 411, 413
current work-item flattened ID 28, 422
currentWorkGroupsSize special instruction 277

D
debug information 303
type ID 303
reference description 312
source line 312
debugtrap special instruction 285, 291
declaration 61
default_float_rounding 324
Dereference (pragma) 316
derived type 309
DerivedType (pragma) 309
DETECT 117, 121, 280-281, 290, 319, 336
extension directive 58, 67, 110, 209, 293-294, 422
dispatch packet instructions 28
workItemFlatabsid 28
divergent control flow 42-43, 45, 332, 422
dominator 46

e
entity ID 303
enumeration type 310
EnumerationType (pragma) 310
Enumerator (pragma) 308
exception instructions 279-280, 290, 336, 411
cleardetextexcept 280
getdetextexcept 280, 290, 336, 411
setdetextexcept 280, 290, 411
executable 422
experimental features 23
expression (pragma group) 315
expression pragma 315
extension 422
extension directive 293
major 293
minor 293
name 293

F
features, experimental 23
File (pragma) 304
filter mode 222-227, 232, 234, 236, 392
finalization 51
finalizer extension 67
fine-grain barrier 26, 44, 65, 67, 72, 106, 249-255, 361-362, 365-366, 402, 409-411, 422
flat addressing 335
Flat memory 31
flattened absolute ID 28, 422, 426
add instruction 153, 289
ceil 153
div 129, 153-154, 289, 331, 412
divmod 154
fma 154, 289, 328-329, 412
fract 154
max 127, 129, 154, 198, 218-219, 412
min 129, 154, 198, 218-219, 412
mul 130, 154, 289, 410, 412
rint 155
sqrt 155, 289
sub 130, 155, 197, 289, 412
trunc 155, 412
floating-point bit instructions 120, 128-130, 157-158, 182-183, 271, 328, 373, 376, 380, 412
abs 158
class 158, 182-183, 271, 373, 376, 380, 412
copiesign 158
neg 158
floating-point optimization instruction 155
forward progress 46
Fragment (pragma) 317
ftz modifier 119, 157, 159, 167, 175, 178, 289, 329
full profile 47, 120, 154-155, 324, 327-329
declaration 107
definition 107-108
function declaration 61-62, 74-75, 78, 83, 106, 266
function definition 61-62, 74-75, 78, 263-266, 364
indirect function 55, 267, 269
function instructions 42, 45, 48-49, 52, 64, 66, 69, 72, 79, 235, 262-275, 295, 371, 411, 416
alloca 264, 274-275, 411, 416
call instruction 66, 262-267, 269, 271
icall 42, 45, 64, 267, 272-273, 416
G

G

H

hardware registers 35, 40, 55, 78
Heterogeneous System Architecture (HSA) 21

definition 21

host CPU 21-22, 33-34, 38, 40, 185, 272, 286, 327, 421-423

HSA See Heterogeneous System Architecture (HSA)

HSA implementation 21-22, 25, 29, 39-42, 44-45, 67, 75,
77-78, 82, 119-121, 126, 133, 159, 184, 186-187,
189, 193, 195, 202, 204-206, 210, 212, 217-218,
220, 222, 224, 227, 229-230, 232, 247-249, 253,
259, 263, 269, 271, 275, 280-281, 284-285, 288-
289, 298, 323, 327-329, 331-332, 351, 356, 420,
423

HSA runtime 287, 423

hsa.tools.debug.assert expr 315

Constu 316

dereference 316

Fragment 317

PlusUConst 316

hsa.tools.debug.List 303

hsa.tools.debug.ref 312

GlobalAddr 313

GlobalImplicit 313

LocalAddr 313

LocalImplicit 314

LocalRegister 315

SourceLine 312

Subprogram 315

hsa.tools.debug.src 303

BasicType 308

CompileUnit 304

CompositeType 310

DerivedType 309

EnumerationType 310

Enumerator 308

File 304

GlobalVariable 307

IndexableType 309

LexicalBlock 306

LocalVariable 307

Location 305

MacroDefine 311

MacroInclude 312

MacroUndefine 311

Namespace 306

Subprogram 305

SubprogramType 311

Subrange 308

Text 305

hsa.tools.debug.tool 303

hsa.tools.debug.version 303

HSAIL 423

HSAIL implementation 24

HSAIL memory 31

HSAIL module 46, 423

HSAIL program 24, 423

I

illegal instruction 423

image 86, 110, 209-214, 219-220, 224-225, 227, 229-232,
234-236, 239, 294, 333, 357-358, 379-380, 390-391

image access permission 210, 228-230, 235

image coordinates 210, 222, 225, 237-242

image data 210-214, 216, 227-230, 234, 236-238, 240-
242, 423

image data layout 210

image element 210, 212-213, 217, 222-225, 227, 236-
237, 241-242

image format 210

image geometry 210-211, 220-222, 225, 228-229, 231,
234, 379-380

image handler 235

image layouts 230

image memory model 32, 183, 211, 236

image sampler 222-223, 233-236, 238-240, 391, 400-
401

image size 86, 209-210, 212, 228-229

pixel 162, 213
texel 225

image data memory 41

image format 210, 213, 219, 228-230, 242

image instructions 120, 183, 209-212, 215, 217-220, 222-
225, 227-230, 234-238, 240, 242, 294, 333, 379,
400

ingefence 244

ldimage 183, 222, 225-226, 230, 234-235, 239-240, 294,
411, 415


queriesampler 235, 242, 244, 295, 359, 380, 411, 415

rdimage 210, 212, 222-223, 226, 232, 234-240, 294,
411, 415, 425

stimage 183, 222, 225-226, 234-235, 241-242, 295, 411,
415

imagefence 295
indexable type 309
IndexableType (pragma) 309
indirect function descriptor 42
individual bit instructions 136-137
   and instruction 137
   not instruction 137
   or instruction 137
   xor instruction 137
instruction set architecture (ISA) 21, 331, 355-356, 367
integer arithmetic instructions 127-130
   abs instruction 128
   add instruction 129
   borrow instruction 129
   carry instruction 129
   div instruction 129
   max instruction 127, 129
   min instruction 129
   mul instruction 130
   mulhi instruction 130
   neg instruction 130
   rem instruction 129
   sub instruction 130
integer optimization instruction 132-133
   mad 133, 412
integer shift instructions 134-135
   shl 135, 412
   shr 135, 412
interval 205-206, 215, 424
   closed interval 218-219, 424
   half-open interval 223, 424
   left-open interval 424
   open interval 424
   right-open interval 296, 323, 361, 371, 381, 424
invalid address 288, 424
ISA  See instruction set architecture (ISA)

K
kernarg segment 34, 36, 39, 56, 76-77, 105, 107, 125-126, 143, 180, 185, 286, 424
kernel descriptor 106
kernel dispatch 47, 421, 424-425
kernel dispatch packet instructions 27, 276-277, 369, 411, 416
current workgroup size 277
dim 277, 369, 411
gridgroups 277
gridsize 277
packet completions sig 277, 416
packetid 277
workgroupid 278
workgroup size 278
workitem asid 278
workitem flat asid 278, 411
workitem flat id 278
workitem id 278

L
lane 29-30, 260-261, 286, 375, 424
lexical block 306
LexicalBlock (pragma) 306
library 325-326, 423
limits 29, 41, 209, 212, 229, 232, 318, 327-328
linkage 60-62, 64, 68-69, 106, 345, 364, 366, 369-370
   arg linkage 68-69, 75, 108
   function linkage 61-62, 68-69, 72, 75, 107-108
   module linkage 60, 62, 69, 72, 75, 107, 325, 424
program linkage 60, 62, 69, 72, 75, 83, 106-107, 325, 425
List (pragma) 303
loader 424
local variable 307
LocalAddr (pragma) 313
LocalImplicit (pragma) 314
LocalRegister (pragma) 315
LocalVariable (pragma) 307
Location (pragma) 305

M
machine instructions 44
machine model 40-41, 110, 141-142, 165-166, 180-181, 202, 253, 315, 324, 335
macro
   define 311
   include file 312
   undefine 311
MacroDefine (pragma) 311
MacroInclude (pragma) 312
MacroUndefine (pragma) 311
memory access 333
memory fence 180, 183-184, 207, 209, 252, 256-257, 346, 376
   atomic 184, 194-196, 202, 204, 248, 283, 373, 411, 414
   atomicnoret 195, 197-198, 200-201, 204, 230, 411, 414
   memfence 207-208, 376, 411, 415
   signalnoret 204, 411, 414
   st 183, 191-192, 194, 197, 235, 265, 295, 411, 414
memory model 21-22, 33, 183, 185, 202, 236, 334, 421, 423
memory scope 103, 184, 186-187, 194-195, 237, 346, 372
memory segment 27, 36-37, 164, 181, 185, 352, 421-425
synchronizing memory instruction 187, 333-334
memory type 31
miscellaneous instructions 283-284
cuid 285, 411
groupbaseptr 285, 411
groupstaticsize 285, 411
grouptotalsize 286, 411
kernargbaseptr 125, 284, 286, 411
laneid 286, 411
maxcuid 285-286, 411
maxwaveid 286, 411
nop 286
nullptr 165, 286, 411
packetid 411
waveid 287, 411
module header 41, 57, 86, 323, 327, 356
module linkage 424
multimedia instructions 161
bitalign 161, 412
bytealign 162, 412
lerp 162, 412
packcv 162, 411, 413
sad 162-163, 411, 413
sadhi 163, 413
unpackcv 162, 411, 413

N

Namespace (pragma) 306
native floating-point instructions 120, 159, 289
ncos 160, 412
nexp2 160, 412
nfma 160
nlog2 160, 289, 412
nrcp 160, 331, 412
nrsc 160, 412
nsin 160, 412
nsqrt 117, 160, 412
natural alignment 76, 125, 232, 369, 424

O

optimization, control flow 332

P

packed data 84, 110, 112, 144, 349
ranges 111
packed data instructions 144
pack 146
shuffle 145, 412
unpack 146
unpackhi 146, 412
unpacklo 145, 412

packet 25
packet ID 47, 425
packet processor 425
packing control 110, 112, 349, 374, 377
ranges 111
padding bytes 351, 360
partial lane 45
partial wavefront 45
partial work-group 26-27, 277
performance tuning 317
persistence rules 39
pixel 210
PlusUconst (pragma) 316
popcount 137
pragma
syntax 303, 312, 315
pragma category
hsa.tools.debug.expr See hsa.tools.debug.expr
hsa.tools.debug.ref See hsa.tools.debug.ref
hsa.tools.debug.src See hsa.tools.debug.src
pragma directive 296-298
private segment 23, 29, 33-37, 39-40, 55, 77, 105-106, 263,
265-266, 269, 271-272, 274-275, 331, 425
profile 22, 41, 109, 120, 159, 217-218, 323-324, 327-328,
350, 368, 407
base profile 328
full profile 329
profile event trigger pragma 301
program linkage 425

Q

queue ID 25, 277, 425

R

race condition 41, 215, 253-254, 256
ranges 111
read atomicity 425
readonly segment 34, 38, 68, 75-77, 103-107, 185, 189,
203, 230, 232-233, 334, 355, 370, 425
register pressure 331
registers 31
release synchronizing operation 425
runtime 21-22, 34, 50-51, 56, 121, 187, 205-206, 220, 229-
230, 232, 236, 288, 290-291, 297, 318, 423
runtime library 318

S

sampler 232-236, 391
sampler handle 110, 210-211, 232-233, 235-236, 391
sampler handle 425
segment 32, 36-37, 39, 75-76, 109-110, 116, 143, 165-166,
180-181, 184-185, 187, 230, 232-233, 235, 286-287,
335, 369, 372-373, 375, 377-378, 425
segment checking instructions 109, 164
segmentp 39, 109, 164, 181, 413
segment conversion instructions 165, 413
  ftos 143, 165-166, 181, 413
  stof 143, 165-166, 181, 413
segment modifier 116, 184
shared virtual memory 36, 424
shuffle instruction 148
signature 407
signed or unsigned 139
small model 40-41
source description (pragma group) 303
source line debug information 312
source line location 305
source reference (pragma group) 312
SourceLine (pragma) 312
special instructions 27, 44, 276, 283, 288
  addqueuewriteindex 282, 416
casqueuewriteindex 282
ldqueuereadindex 282, 416
ldqueuewriteindex 282, 416
stqueuereadindex 283, 416
stqueuewriteindex 283
spill segment 35, 40-41, 55, 76, 78, 105-106, 143, 331, 425
stack machine model 315
subprogram 305
  definition 315
type 311
Subprogram (pragma) 305, 315
SubprogramType (pragma) 311
Subrange (pragma) 308

T
texel 425
Text (pragma) 305
tool (pragma) 303
translation examples to HSAIL
  transpose 49
  vector add 48

U
uniform instruction 426
unit of execution 426
unit of least precision (ULP) 426
URL 426
user mode queue 47, 426
user mode queue instructions 281-282
  addqueuewriteindex 282
casqueuewriteindex 282
ldqueuereadindex 282
ldqueuewriteindex 282
stqueuereadindex 283
stqueuewriteindex 283

V
variadic function 267
vector operand 115

version (pragma) 303
virtual machine 20-21, 298, 423

W
wavefront 29-30, 40, 43, 251-252, 260, 274, 280, 291-292,
  390, 424, 426
wavefront size 29-31, 42, 44, 46, 248, 259-260, 286, 320,
  322, 332, 356, 424
WAVESIZE 31, 45-46, 103, 113-115, 189, 191, 207, 254-256,
  259-261, 286, 296, 356, 390, 406, 426
width modifier 44-46, 189-190, 247-249, 271, 273, 356, 373-376
work-group 25-29, 33, 36, 39, 44, 47, 55, 123, 184, 236-237,
  248-249, 252-253, 255, 274, 290, 319-320, 362-363,
  422-423, 426-427
work-group absolute ID 36, 422
work-group flattened ID 27, 426
work-group ID 26, 279, 426
work-item 22, 27-28, 33-34, 38-41, 44-45, 55, 105-106, 184,
  259-261, 286, 296, 356, 390, 406, 426
  183-190, 205, 236-237, 250-251, 253, 260, 262,
  280, 284-286, 292, 320, 425-427
work-item absolute ID 26, 28, 36, 279, 422, 426
work-item flattened absolute ID 28, 426
work-item flattened ID 28, 30, 426
work-item flattened ID, current 28
work-item ID 27-28, 278-279, 427
workitemflatabsid 278
write atomicity 427