User Guide for AMDGPU Backend

Introduction

The AMDGPU backend provides ISA code generation for AMD GPUs, starting with the R600 family up until the current GCN families. It lives in the `llvm/lib/Target/AMDGPU` directory.

LLVM

Target Triples

Use the Clang option `-target <Architecture>-<Vendor>-<OS>-<Environment>` to specify the target triple:

AMDGPU Architectures

Architecture	Description
`r600`	AMD GPUs HD2XXX-HD6XXX for graphics and compute shaders.
`amdgcn`	AMD GPUs GCN GFX6 onwards for graphics and compute shaders.

AMDGPU Vendors

Vendor	Description
`amd`	Can be used for all AMD GPU usage.
`mesa3d`	Can be used if the OS is `mesa3d`.

AMDGPU Operating Systems

OS	Description

<empty>	Defaults to the *unknown* OS.
`amdhsa`	Compute kernels executed on HSA [HSA] compatible runtimes such as: • AMD's ROCm™ runtime [AMD-ROCm] using the *rocm-amdhsa* loader on Linux. See *AMD ROCm Platform Release Notes* [AMD-ROCm-Release-Notes] for supported hardware and software. • AMD's PAL runtime using the *pal-amdhsa* loader on Windows.
`amdpal`	Graphic shaders and compute kernels executed on AMD's PAL runtime using the *pal-amdpal* loader on Windows and Linux Pro.
`mesa3d`	Graphic shaders and compute kernels executed on AMD's Mesa 3D runtime using the *mesa-mesa3d* loader on Linux.

AMDGPU Environments

Environment	Description
<empty>	Default.

Processors

Use the Clang options `-mcpu=<target-id>` or `--offload-arch=<target-id>` to specify the AMDGPU processor together with optional target features. See Target ID and Target Features for AMD GPU target specific information.

Every processor supports every OS ABI (see AMDGPU Operating Systems) with the following exceptions:

- `amdhsa` is not supported in `r600` architecture (see AMDGPU Architectures).

AMDGPU Processors

Processor	Alternative Processor	Target Triple Architecture	dGPU/ APU	Target Features Supported	Target Properties	OS Support *(see amdgpu-os and corresponding runtime release notes for current information and level of support)*	Example Products
Radeon HD 2000/3000 Series (R600) [AMD-RADEON-HD-2000-3000]							

r600	r600	dGPU	◦ Does not support generic address space
r630	r600	dGPU	◦ Does not support generic address space
rs880	r600	dGPU	◦ Does not support generic address space
rv670	r600	dGPU	◦ Does not support generic address space
Radeon HD 4000 Series (R700) [AMD-RADEON-HD-4000]			
rv710	r600	dGPU	◦ Does not support generic address space
rv730	r600	dGPU	◦ Does not support generic address space
rv770	r600	dGPU	◦ Does not support generic address space
Radeon HD 5000 Series (Evergreen) [AMD-RADEON-HD-5000]			

cedar	r600	dGPU	◦ Does not support generic address space
cypress	r600	dGPU	◦ Does not support generic address space
juniper	r600	dGPU	◦ Does not support generic address space
redwood	r600	dGPU	◦ Does not support generic address space
sumo	r600	dGPU	◦ Does not support generic address space
Radeon HD 6000 Series (Northern Islands) [AMD-RADEON-HD-6000]			
barts	r600	dGPU	◦ Does not support generic address space
caicos	r600	dGPU	◦ Does not support generic address space
cayman	r600	dGPU	◦ Does not

				• Does not support generic address space	
`turks`		`r600`	dGPU	• Does not support generic address space	
GCN GFX6 (Southern Islands (SI)) [AMD-GCN-GFX6]					
`gfx600`	• `tahiti`	`amdgcn`	dGPU	• Does not support generic address space	• *pal-amdpal*
`gfx601`	• `pitcairn` • `verde`	`amdgcn`	dGPU	• Does not support generic address space	• *pal-amdpal*
`gfx602`	• `hainan` • `oland`	`amdgcn`	dGPU	• Does not support generic address space	• *pal-amdpal*
GCN GFX7 (Sea Islands (CI)) [AMD-GCN-GFX7]					
`gfx700`	• `kaveri`	`amdgcn`	APU	• Offset flat scratch	• *rocm-amdhsa* • *pal-amdhsa* • *pal-amdpal*	• A6-7000 • A6 Pro-7050B • A8-7100 • A8 Pro-7150B • A10-7300 • A10 Pro-7350B • FX-7500 • A8-7200P

						◦ A10-7400P ◦ FX-7600P
gfx701	◦ hawaii	amdgcn	dGPU	◦ Offset flat scratch	◦ *rocm-amdhsa* ◦ *pal-amdhsa* ◦ *pal-amdpal*	◦ FirePro W8100 ◦ FirePro W9100 ◦ FirePro S9150 ◦ FirePro S9170
gfx702		amdgcn	dGPU	◦ Offset flat scratch	◦ *rocm-amdhsa* ◦ *pal-amdhsa* ◦ *pal-amdpal*	◦ Radeon R9 290 ◦ Radeon R9 290x ◦ Radeon R390 ◦ Radeon R390x
gfx703	◦ kabini ◦ mullins	amdgcn	APU	◦ Offset flat scratch	◦ *pal-amdhsa* ◦ *pal-amdpal*	◦ E1-2100 ◦ E1-2200 ◦ E1-2500 ◦ E2-3000 ◦ E2-3800 ◦ A4-5000 ◦ A4-5100 ◦ A6-5200 ◦ A4 Pro-3340B
gfx704	◦ bonaire	amdgcn	dGPU	◦ Offset flat scratch	◦ *pal-amdhsa* ◦ *pal-amdpal*	◦ Radeon HD 7790 ◦ Radeon HD 8770 ◦ R7 260 ◦ R7 260X
gfx705		amdgcn	APU	◦ Offset flat scratch	◦ *pal-amdhsa*	*TBA*

						◦ *pal-amdpal*	
GCN GFX8 (Volcanic Islands (VI)) [AMD-GCN-GFX8]							
gfx801	◦ carrizo	amdgcn	APU	◦ xnack	◦ Offset flat scratch	◦ *rocm-amdhsa* ◦ *pal-amdhsa* ◦ *pal-amdpal*	◦ A6-8500P ◦ Pro A6-8500B ◦ A8-8600P ◦ Pro A8-8600B ◦ FX-8800P ◦ Pro A12-8800B ◦ A10-8700P ◦ Pro A10-8700B ◦ A10-8780P ◦ A10-9600P ◦ A10-9630P ◦ A12-9700P ◦ A12-9730P ◦ FX-9800P ◦ FX-9830P ◦ E2-9010 ◦ A6-9210 ◦ A9-9410
gfx802	◦ iceland ◦ tonga	amdgcn	dGPU		◦ Offset flat scratch	◦ *rocm-amdhsa* ◦ *pal-amdhsa* ◦ *pal-amdpal*	◦ Radeon R9 285 ◦ Radeon R9 380 ◦ Radeon R9 385

gfx803	fiji	amdgcn	dGPU			• *rocm-amdhsa* • *pal-amdhsa* • *pal-amdpal*	• Radeon R9 Nano • Radeon R9 Fury • Radeon R9 FuryX • Radeon Pro Duo • FirePro S9300x2 • Radeon Instinct MI8
	• polaris10	amdgcn	dGPU		• Offset flat scratch	• *rocm-amdhsa* • *pal-amdhsa* • *pal-amdpal*	• Radeon RX 470 • Radeon RX 480 • Radeon Instinct MI6
	• polaris11	amdgcn	dGPU		• Offset flat scratch	• *rocm-amdhsa* • *pal-amdhsa* • *pal-amdpal*	• Radeon RX 460
gfx805	• tongapro	amdgcn	dGPU		• Offset flat scratch	• *rocm-amdhsa* • *pal-amdhsa* • *pal-amdpal*	• FirePro S7150 • FirePro S7100 • FirePro W7100 • Mobile FirePro M7170
gfx810	• stoney	amdgcn	APU	• xnack	• Offset flat scratch	• *rocm-amdhsa*	*TBA*

					◦ *pal-amdhsa* ◦ *pal-amdpal*	
GCN GFX9 (Vega) [AMD-GCN-GFX900-GFX904-VEGA] [AMD-GCN-GFX906-VEGA7NM] [AMD-GCN-GFX908-CDNA1] [AMD-GCN-GFX90A-CDNA2]						
`gfx900`	`amdgcn`	dGPU	◦ xnack	◦ Absolute flat scratch	◦ *rocm-amdhsa* ◦ *pal-amdhsa* ◦ *pal-amdpal*	◦ Radeon Vega Frontier Edition ◦ Radeon RX Vega 56 ◦ Radeon RX Vega 64 ◦ Radeon RX Vega 64 Liquid ◦ Radeon Instinct MI25
`gfx902`	`amdgcn`	APU	◦ xnack	◦ Absolute flat scratch	◦ *rocm-amdhsa* ◦ *pal-amdhsa* ◦ *pal-amdpal*	◦ Ryzen 3 2200G ◦ Ryzen 5 2400G
`gfx904`	`amdgcn`	dGPU	◦ xnack		◦ *rocm-amdhsa* ◦ *pal-amdhsa* ◦ *pal-amdpal*	*TBA*
`gfx906`	`amdgcn`	dGPU	◦ sramecc ◦ xnack	◦ Absolute flat scratch	◦ *rocm-amdhsa* ◦ *pal-amdhsa* ◦ *pal-*	◦ Radeon Instinct MI50 ◦ Radeon Instinct

					amdpal	MI60 ◦ Radeon VII ◦ Radeon Pro VII
`gfx908`	`amdgcn`	dGPU	◦ sramecc ◦ xnack	◦ Absolute flat scratch	◦ *rocm-amdhsa*	◦ AMD Instinct MI100 Accelerator
`gfx909`	`amdgcn`	APU	◦ xnack	◦ Absolute flat scratch	◦ *pal-amdpal*	*TBA*
`gfx90a`	`amdgcn`	dGPU	◦ sramecc ◦ tgsplit ◦ xnack	◦ Absolute flat scratch ◦ Packed work-item IDs	◦ *rocm-amdhsa*	*TBA*
`gfx90c`	`amdgcn`	APU	◦ xnack	◦ Absolute flat scratch	◦ *pal-amdpal*	◦ Ryzen 7 4700G ◦ Ryzen 7 4700GE ◦ Ryzen 5 4600G ◦ Ryzen 5 4600GE ◦ Ryzen 3 4300G ◦ Ryzen 3 4300GE ◦ Ryzen Pro 4000G ◦ Ryzen 7 Pro 4700G ◦ Ryzen 7 Pro 4750GE ◦ Ryzen 5 Pro 4650G ◦ Ryzen 5

						Pro 4650GE ◦ Ryzen 3 Pro 4350G ◦ Ryzen 3 Pro 4350GE
`gfx940`	`amdgcn`	dGPU	◦ sramecc ◦ tgsplit ◦ xnack	◦ Architected flat scratch ◦ Packed work-item IDs		*TBA*
`gfx941`	`amdgcn`	dGPU	◦ sramecc ◦ tgsplit ◦ xnack	◦ Architected flat scratch ◦ Packed work-item IDs		*TBA*
`gfx942`	`amdgcn`	dGPU	◦ sramecc ◦ tgsplit ◦ xnack	◦ Architected flat scratch ◦ Packed work-item IDs		*TBA*
GCN GFX10.1 (RDNA 1) [AMD-GCN-GFX10-RDNA1]						
`gfx1010`	`amdgcn`	dGPU	◦ cumode ◦ wavefrontsize64 ◦ xnack	◦ Absolute flat scratch	◦ *rocm-amdhsa* ◦ *pal-amdhsa* ◦ *pal-amdpal*	◦ Radeon RX 5700 ◦ Radeon RX 5700 XT ◦ Radeon Pro 5600 XT ◦ Radeon Pro 5600M
`gfx1011`	`amdgcn`	dGPU	◦ cumode ◦ wavefrontsize64 ◦ xnack	◦ Absolute flat scratch	◦ *rocm-amdhsa* ◦ *pal-amdhsa* ◦ *pal-amdpal*	◦ Radeon Pro V520

`gfx1012`	`amdgcn`	dGPU	◦ cumode ◦ wavefrontsize64 ◦ xnack	◦ Absolute flat scratch	◦ *rocm-amdhsa* ◦ *pal-amdhsa* ◦ *pal-amdpal*	◦ Radeon RX 5500 ◦ Radeon RX 5500 XT
`gfx1013`	`amdgcn`	APU	◦ cumode ◦ wavefrontsize64 ◦ xnack	◦ Absolute flat scratch	◦ *rocm-amdhsa* ◦ *pal-amdhsa* ◦ *pal-amdpal*	*TBA*
GCN GFX10.3 (RDNA 2) [AMD-GCN-GFX10-RDNA2]						
`gfx1030`	`amdgcn`	dGPU	◦ cumode ◦ wavefrontsize64	◦ Absolute flat scratch	◦ *rocm-amdhsa* ◦ *pal-amdhsa* ◦ *pal-amdpal*	◦ Radeon RX 6800 ◦ Radeon RX 6800 XT ◦ Radeon RX 6900 XT
`gfx1031`	`amdgcn`	dGPU	◦ cumode ◦ wavefrontsize64	◦ Absolute flat scratch	◦ *rocm-amdhsa* ◦ *pal-amdhsa* ◦ *pal-amdpal*	◦ Radeon RX 6700 XT
`gfx1032`	`amdgcn`	dGPU	◦ cumode ◦ wavefrontsize64	◦ Absolute flat scratch	◦ *rocm-amdhsa* ◦ *pal-amdhsa* ◦ *pal-amdpal*	*TBA*
`gfx1033`	`amdgcn`	APU	◦ cumode ◦ wavefrontsize64	◦ Absolute flat scratch	◦ *pal-amdpal*	*TBA*
`gfx1034`	`amdgcn`	dGPU	◦ cumode ◦ wavefrontsize64	◦ Absolute flat scratch	◦ *pal-amdpal*	*TBA*
`gfx1035`	`amdgcn`	APU	◦ cumode ◦ wavefrontsize64	◦ Absolute flat scratch	◦ *pal-amdpal*	*TBA*

`gfx1036`	`amdgcn`	APU	◦ cumode ◦ wavefrontsize64	◦ Absolute flat scratch	◦ *pal-amdpal*	*TBA*
GCN GFX11 (RDNA 3) [AMD-GCN-GFX11-RDNA3]						
`gfx1100`	`amdgcn`	dGPU	◦ cumode ◦ wavefrontsize64	◦ Architected flat scratch ◦ Packed work-item IDs	◦ *pal-amdpal*	*TBA*
`gfx1101`	`amdgcn`	dGPU	◦ cumode ◦ wavefrontsize64	◦ Architected flat scratch ◦ Packed work-item IDs		*TBA*
`gfx1102`	`amdgcn`	dGPU	◦ cumode ◦ wavefrontsize64	◦ Architected flat scratch ◦ Packed work-item IDs		*TBA*
`gfx1103`	`amdgcn`	APU	◦ cumode ◦ wavefrontsize64	◦ Architected flat scratch ◦ Packed work-item IDs		*TBA*

Target Features

Target features control how code is generated to support certain processor specific features. Not all target features are supported by all processors. The runtime must ensure that the features supported by the device used to execute the code match the features enabled when generating the code. A mismatch of features may result in incorrect execution, or a reduction in performance.

The target features supported by each processor is listed in AMDGPU Processors.

Target features are controlled by exactly one of the following Clang options:

`-mcpu=<target-id>` or `--offload-arch=<target-id>`

> The `-mcpu` and `--offload-arch` can specify the target feature as optional components of the target ID. If omitted, the target feature has the `any` value. See Target ID.

`-m[no-]<target-feature>`

Target features not specified by the target ID are specified using a separate option. These target features can have an `on` or `off` value. `on` is specified by omitting the `no-` prefix, and `off` is specified by including the `no-` prefix. The default if not specified is `off`.

For example:

`-mcpu=gfx908:xnack+`

Enable the `xnack` feature.

`-mcpu=gfx908:xnack-`

Disable the `xnack` feature.

`-mcumode`

Enable the `cumode` feature.

`-mno-cumode`

Disable the `cumode` feature.

AMDGPU Target Features

Target Feature Name	Clang Option to Control	Description
cumode	• `-m[no-]cumode`	Control the wavefront execution mode used when generating code for kernels. When disabled native WGP wavefront execution mode is used, when enabled CU wavefront execution mode is used (see Memory Model).
sramecc	• `-mcpu` • `--offload-arch`	If specified, generate code that can only be loaded and executed in a process that has a matching setting for SRAMECC. If not specified for code object V2 to V3, generate code that can be loaded and executed in a process with SRAMECC enabled. If not specified for code object V4 or above, generate code that can be loaded and executed in a process with either setting of SRAMECC.
tgsplit	`-m[no-]tgsplit`	Enable/disable generating code that assumes

		work-groups are launched in threadgroup split mode. When enabled the waves of a work-group may be launched in different CUs.
wavefrontsize64	• `-m[no-]wavefrontsize64`	Control the wavefront size used when generating code for kernels. When disabled native wavefront size 32 is used, when enabled wavefront size 64 is used.
xnack	• `-mcpu` • `--offload-arch`	If specified, generate code that can only be loaded and executed in a process that has a matching setting for XNACK replay. If not specified for code object V2 to V3, generate code that can be loaded and executed in a process with XNACK replay enabled. If not specified for code object V4 or above, generate code that can be loaded and executed in a process with either setting of XNACK replay. XNACK replay can be used for demand paging and page migration. If enabled in the device, then if a page fault occurs the code may execute incorrectly unless generated with XNACK replay enabled, or generated for code object V4 or above without specifying XNACK replay. Executing code that was generated with XNACK replay enabled, or generated for code object V4 or above without specifying XNACK replay, on a device that does not have XNACK replay enabled will execute correctly but may be less performant than code generated for XNACK replay disabled.

Target ID

AMDGPU supports target IDs. See Clang Offload Bundler for a general description. The AMDGPU target specific information is:

processor

Is an AMDGPU processor or alternative processor name specified in AMDGPU Processors. The non-canonical form target ID allows both the primary processor and alternative processor

names. The canonical form target ID only allow the primary processor name.

target-feature

Is a target feature name specified in AMDGPU Target Features that is supported by the processor. The target features supported by each processor is specified in AMDGPU Processors. Those that can be specified in a target ID are marked as being controlled by `-mcpu` and `--offload-arch`. Each target feature must appear at most once in a target ID. The non-canonical form target ID allows the target features to be specified in any order. The canonical form target ID requires the target features to be specified in alphabetic order.

Code Object V2 to V3 Target ID

The target ID syntax for code object V2 to V3 is the same as defined in Clang Offload Bundler except when used in the .amdgcn_target <target-triple> "-" <target-id> assembler directive and the bundle entry ID. In those cases it has the following BNF syntax:

```
<target-id> ::== <processor> ( "+" <target-feature> )*
```

Where a target feature is omitted if *Off* and present if *On* or *Any*.

Note

The code object V2 to V3 cannot represent *Any* and treats it the same as *On*.

Embedding Bundled Code Objects

AMDGPU supports the HIP and OpenMP languages that perform code object embedding as described in Clang Offload Bundler.

Note

The target ID syntax used for code object V2 to V3 for a bundle entry ID differs from that used elsewhere. See Code Object V2 to V3 Target ID.

Address Spaces

The AMDGPU architecture supports a number of memory address spaces. The address space names use the OpenCL standard names, with some additions.

The AMDGPU address spaces correspond to target architecture specific LLVM address space numbers used in LLVM IR.

The AMDGPU address spaces are described in AMDGPU Address Spaces. Only 64-bit process address spaces are supported for the `amdgcn` target.

AMDGPU Address Spaces

				64-Bit Process Address Space	
Address Space Name	LLVM IR Address Space Number	HSA Segment Name	Hardware Name	Address Size	NULL Value
Generic	0	flat	flat	64	0x0000000000000000
Global	1	global	global	64	0x0000000000000000
Region	2	N/A	GDS	32	*not implemented for AMDHSA*
Local	3	group	LDS	32	0xFFFFFFFF
Constant	4	constant	*same as global*	64	0x0000000000000000
Private	5	private	scratch	32	0xFFFFFFFF
Constant 32-bit	6	*TODO*			0x00000000
Buffer Fat Pointer (experimental)	7	*TODO*			
Buffer Resource (experimental)	8	*TODO*			
Streamout Registers	128	N/A	GS_REGS		

Generic

The generic address space is supported unless the *Target Properties* column of AMDGPU Processors specifies *Does not support generic address space*.

The generic address space uses the hardware flat address support for two fixed ranges of virtual addresses (the private and local apertures), that are outside the range of addressable global memory, to map from a flat address to a private or local address. This uses FLAT instructions that can take a flat address and access global, private (scratch), and group (LDS) memory depending on if the address is within one of the aperture ranges.

Flat access to scratch requires hardware aperture setup and setup in the kernel prologue (see Flat Scratch). Flat access to LDS requires hardware aperture setup and M0 (GFX7-GFX8) register setup (see M0).

To convert between a private or group address space address (termed a segment address) and a flat address the base address of the corresponding aperture can be used. For GFX7-GFX8 these are available in the HSA AQL Queue the address of which can be obtained with Queue Ptr SGPR (see Initial Kernel Execution State). For GFX9-GFX11 the aperture base addresses are directly available as inline constant registers `SRC_SHARED_BASE/LIMIT` and `SRC_PRIVATE_BASE/LIMIT`. In 64-bit address mode the aperture sizes are 2^32 bytes and the base is aligned to 2^32 which makes it easier to convert from flat to segment or segment to flat.

A global address space address has the same value when used as a flat address so no conversion is needed.

Global and Constant

The global and constant address spaces both use global virtual addresses, which are the same virtual address space used by the CPU. However, some virtual addresses may only be accessible to the CPU, some only accessible by the GPU, and some by both.

Using the constant address space indicates that the data will not change during the execution of the kernel. This allows scalar read instructions to be used. As the constant address space could only be modified on the host side, a generic pointer loaded from the constant address space is safe to be assumed as a global pointer since only the device global memory is visible and managed on the host side. The vector and scalar L1 caches are invalidated of volatile data before each kernel dispatch execution to allow constant memory to change values between kernel dispatches.

Region

The region address space uses the hardware Global Data Store (GDS). All wavefronts executing on the same device will access the same memory for any given region address. However, the same region address accessed by wavefronts executing on different devices will access different memory. It is higher performance than global memory. It is allocated by the runtime. The data store (DS) instructions can be used to access it.

Local

The local address space uses the hardware Local Data Store (LDS) which is automatically allocated when the hardware creates the wavefronts of a work-group, and freed when all the wavefronts of a work-group have terminated. All wavefronts belonging to the same work-group will access the same memory for any given local address. However, the same local address accessed by wavefronts belonging to different work-groups will access different memory. It is higher performance than global memory. The data store (DS) instructions can be used to access it.

Private

The private address space uses the hardware scratch memory support which automatically allocates memory when it creates a wavefront and frees it when a wavefronts terminates. The memory accessed by a lane of a wavefront for any given private address will be different to the memory accessed by another lane of the same or different wavefront for the same private address.

If a kernel dispatch uses scratch, then the hardware allocates memory from a pool of backing memory allocated by the runtime for each wavefront. The lanes of the wavefront access this using dword (4 byte) interleaving. The mapping used from private address to backing memory address is:

```
wavefront-scratch-base + ((private-address / 4) * wavefront-size * 4) +
(wavefront-lane-id * 4) + (private-address % 4)
```

If each lane of a wavefront accesses the same private address, the interleaving results in adjacent dwords being accessed and hence requires fewer cache lines to be fetched.

There are different ways that the wavefront scratch base address is determined by a wavefront (see Initial Kernel Execution State).

Scratch memory can be accessed in an interleaved manner using buffer instructions with the scratch buffer descriptor and per wavefront scratch offset, by the scratch instructions, or by flat instructions. Multi-dword access is not supported except by flat and scratch instructions in GFX9-GFX11.

Code that manipulates the stack values in other lanes of a wavefront, such as by `addrspacecast`-ing stack pointers to generic ones and taking offsets that reach other lanes or by explicitly constructing the scratch buffer descriptor, triggers undefined behavior when it modifies the scratch values of other lanes. The compiler may assume that such modifications do not occur.

Constant 32-bit

TODO

Buffer Fat Pointer

The buffer fat pointer is an experimental address space that is currently unsupported in the backend. It exposes a non-integral pointer that is in the future intended to support the modelling of 128-bit buffer descriptors plus a 32-bit offset into the buffer (in total encapsulating a 160-bit *pointer*), allowing normal LLVM load/store/atomic operations to be used to model the buffer descriptors used heavily in graphics workloads targeting the backend.

The buffer descriptor used to construct a buffer fat pointer must be *raw*: the stride must be 0, the "add tid" flag bust be 0, the swizzle enable bits must be off, and the extent must be measured in bytes. (On subtargets where bounds checking may be disabled, buffer fat pointers may choose to enable it or not).

Buffer Resource

The buffer resource pointer, in address space 8, is the newer form for representing buffer descriptors in AMDGPU IR, replacing their previous representation as `<4 x i32>`. It is a non-integral pointer that represents a 128-bit buffer descriptor resource (*V#*).

Since, in general, a buffer resource supports complex addressing modes that cannot be easily represented in LLVM (such as implicit swizzled access to structured buffers), it is illegal to perform non-trivial address computations, such as `getelementptr` operations, on buffer resources. They may be passed to AMDGPU buffer intrinsics, and they may be converted to and from `i128`.

Casting a buffer resource to a buffer fat pointer is permitted and adds an offset of 0.

Buffer resources can be created from 64-bit pointers (which should be either generic or global) using the *llvm.amdgcn.make.buffer.rsrc* intrinsic, which takes the pointer, which becomes the base of the resource, the 16-bit stride (and swzizzle control) field stored in bits *63:48* of a *V#*, the 32-bit NumRecords/extent field (bits *95:64*), and the 32-bit flags field (bits *127:96*). The specific interpretation of these fields varies by the target architecture and is detailed in the ISA descriptions.

Streamout Registers

Dedicated registers used by the GS NGG Streamout Instructions. The register file is modelled as a memory in a distinct address space because it is indexed by an address-like offset in place of named registers, and because register accesses affect LGKMcnt. This is an internal address space used only by the compiler. Do not use this address space for IR pointers.

Memory Scopes

This section provides LLVM memory synchronization scopes supported by the AMDGPU backend memory model when the target triple OS is `amdhsa` (see Memory Model and Target Triples).

The memory model supported is based on the HSA memory model [HSA] which is based in turn on HRF-indirect with scope inclusion [HRF]. The happens-before relation is transitive over the synchronizes-with relation independent of scope and synchronizes-with allows the memory scope instances to be inclusive (see table AMDHSA LLVM Sync Scopes).

This is different to the OpenCL [OpenCL] memory model which does not have scope inclusion and

requires the memory scopes to exactly match. However, this is conservatively correct for OpenCL.

AMDHSA LLVM Sync Scopes

LLVM Sync Scope	Description
none	The default: `system`. Synchronizes with, and participates in modification and seq_cst total orderings with, other operations (except image operations) for all address spaces (except private, or generic that accesses private) provided the other operation's sync scope is: • `system`. • `agent` and executed by a thread on the same agent. • `workgroup` and executed by a thread in the same work-group. • `wavefront` and executed by a thread in the same wavefront.
`agent`	Synchronizes with, and participates in modification and seq_cst total orderings with, other operations (except image operations) for all address spaces (except private, or generic that accesses private) provided the other operation's sync scope is: • `system` or `agent` and executed by a thread on the same agent. • `workgroup` and executed by a thread in the same work-group. • `wavefront` and executed by a thread in the same wavefront.
`workgroup`	Synchronizes with, and participates in modification and seq_cst total orderings with, other operations (except image operations) for all address spaces (except private, or generic that accesses private) provided the other operation's sync scope is: • `system`, `agent` or `workgroup` and executed by a thread in the same work-group.

	• `wavefront` and executed by a thread in the same wavefront.
`wavefront`	Synchronizes with, and participates in modification and seq_cst total orderings with, other operations (except image operations) for all address spaces (except private, or generic that accesses private) provided the other operation's sync scope is: • `system`, `agent`, `workgroup` or `wavefront` and executed by a thread in the same wavefront.
`singlethread`	Only synchronizes with and participates in modification and seq_cst total orderings with, other operations (except image operations) running in the same thread for all address spaces (for example, in signal handlers).
`one-as`	Same as `system` but only synchronizes with other operations within the same address space.
`agent-one-as`	Same as `agent` but only synchronizes with other operations within the same address space.
`workgroup-one-as`	Same as `workgroup` but only synchronizes with other operations within the same address space.
`wavefront-one-as`	Same as `wavefront` but only synchronizes with other operations within the same address space.
`singlethread-one-as`	Same as `singlethread` but only synchronizes with other operations within the same address space.

LLVM IR Intrinsics

The AMDGPU backend implements the following LLVM IR intrinsics.

This section is WIP.

AMDGPU LLVM IR Intrinsics

LLVM Intrinsic	Description
llvm.amdgcn.log	Provides direct access to v_log_f32 and v_log_f16 (on targets with half support). Peforms log2 function.
llvm.amdgcn.exp2	Provides direct access to v_exp_f32 and v_exp_f16 (on targets with half support). Performs exp2 function.

LLVM IR Attributes

The AMDGPU backend supports the following LLVM IR attributes.

AMDGPU LLVM IR Attributes

LLVM Attribute	Description
“amdgpu-flat-work-group-size”=”min,max”	Specify the minimum and maximum flat work group sizes that will be specified when the kernel is dispatched. Generated by the `amdgpu_flat_work_group_size` CLANG attribute [CLANG-ATTR]. The implied default value is 1,1024.
“amdgpu-implicitarg-num-bytes”=”n”	Number of kernel argument bytes to add to the kernel argument block size for the implicit arguments. This varies by OS and language (for OpenCL see OpenCL kernel implicit arguments appended for AMDHSA OS).
“amdgpu-num-sgpr”=”n”	Specifies the number of SGPRs to use. Generated by the `amdgpu_num_sgpr` CLANG attribute [CLANG-ATTR].
“amdgpu-num-vgpr”=”n”	Specifies the number of VGPRs to use. Generated by the `amdgpu_num_vgpr` CLANG attribute [CLANG-ATTR].
“amdgpu-waves-per-eu”=”m,n”	Specify the minimum and maximum number of waves per execution unit. Generated by the `amdgpu_waves_per_eu` CLANG attribute [CLANG-ATTR]. This is an optimization hint, and the backend may not be able to satisfy the request. If the specified range is incompatible with the function’s “amdgpu-flat-work-group-size” value, the implied occupancy bounds by the workgroup size takes precedence.
“amdgpu-ieee” true/false.	Specify whether the function expects the IEEE field of the mode register to be set on entry. Overrides the default for the calling convention.
“amdgpu-dx10-clamp” true/false.	Specify whether the function expects the DX10_CLAMP field of the mode register to be set on entry. Overrides the default for the calling convention.
“amdgpu-no-workitem-id-x”	Indicates the function does not depend on the value of the llvm.amdgcn.workitem.id.x intrinsic. If a function is marked with this attribute, or reached through a call

	site marked with this attribute, the value returned by the intrinsic is undefined. The backend can generally infer this during code generation, so typically there is no benefit to frontends marking functions with this.
“amdgpu-no-workitem-id-y”	The same as amdgpu-no-workitem-id-x, except for the llvm.amdgcn.workitem.id.y intrinsic.
“amdgpu-no-workitem-id-z”	The same as amdgpu-no-workitem-id-x, except for the llvm.amdgcn.workitem.id.z intrinsic.
“amdgpu-no-workgroup-id-x”	The same as amdgpu-no-workitem-id-x, except for the llvm.amdgcn.workgroup.id.x intrinsic.
“amdgpu-no-workgroup-id-y”	The same as amdgpu-no-workitem-id-x, except for the llvm.amdgcn.workgroup.id.y intrinsic.
“amdgpu-no-workgroup-id-z”	The same as amdgpu-no-workitem-id-x, except for the llvm.amdgcn.workgroup.id.z intrinsic.
“amdgpu-no-dispatch-ptr”	The same as amdgpu-no-workitem-id-x, except for the llvm.amdgcn.dispatch.ptr intrinsic.
“amdgpu-no-implicitarg-ptr”	The same as amdgpu-no-workitem-id-x, except for the llvm.amdgcn.implicitarg.ptr intrinsic.
“amdgpu-no-dispatch-id”	The same as amdgpu-no-workitem-id-x, except for the llvm.amdgcn.dispatch.id intrinsic.
“amdgpu-no-queue-ptr”	Similar to amdgpu-no-workitem-id-x, except for the llvm.amdgcn.queue.ptr intrinsic. Note that unlike the other ABI hint attributes, the queue pointer may be required in situations where the intrinsic call does not directly appear in the program. Some subtargets require the queue pointer for to handle some addrspacecasts, as well as the llvm.amdgcn.is.shared, llvm.amdgcn.is.private, llvm.trap, and llvm.debug intrinsics.
“amdgpu-no-hostcall-ptr”	Similar to amdgpu-no-implicitarg-ptr, except specific to the implicit kernel argument that holds the pointer to the hostcall buffer. If this attribute is absent, then the amdgpu-no-implicitarg-ptr is also removed.
“amdgpu-no-heap-ptr”	Similar to amdgpu-no-implicitarg-ptr, except specific to the implicit kernel argument that holds the pointer to an initialized memory buffer that conforms to the

	requirements of the malloc/free device library V1 version implementation. If this attribute is absent, then the amdgpu-no-implicitarg-ptr is also removed.
“amdgpu-no-multigrid-sync-arg”	Similar to amdgpu-no-implicitarg-ptr, except specific to the implicit kernel argument that holds the multigrid synchronization pointer. If this attribute is absent, then the amdgpu-no-implicitarg-ptr is also removed.
“amdgpu-no-default-queue”	Similar to amdgpu-no-implicitarg-ptr, except specific to the implicit kernel argument that holds the default queue pointer. If this attribute is absent, then the amdgpu-no-implicitarg-ptr is also removed.
“amdgpu-no-completion-action”	Similar to amdgpu-no-implicitarg-ptr, except specific to the implicit kernel argument that holds the completion action pointer. If this attribute is absent, then the amdgpu-no-implicitarg-ptr is also removed.

Calling Conventions

The AMDGPU backend supports the following calling conventions:

AMDGPU Calling Conventions

Calling Convention	Description
`ccc`	The C calling convention. Used by default. See Non-Kernel Functions for more details.
`fastcc`	The fast calling convention. Mostly the same as the `ccc`.
`coldcc`	The cold calling convention. Mostly the same as the `ccc`.
`amdgpu_cs`	Used for Mesa/AMDPAL compute shaders. ..TODO:: Describe.
`amdgpu_cs_chain`	Similar to `amdgpu_cs`, with differences described below. Functions with this calling convention cannot be called directly. They must instead be launched via the `llvm.amdgcn.cs.chain` intrinsic. Arguments are passed in SGPRs, starting at s0, if they have the `inreg` attribute, and in VGPRs otherwise, starting at v8. Using more SGPRs or VGPRs than available in the subtarget is not allowed. On subtargets that use a scratch buffer

	descriptor (as opposed to `scratch_{load,store}_*` instructions), the scratch buffer descriptor is passed in s[48:51]. This limits the SGPR / `inreg` arguments to the equivalent of 48 dwords; using more than that is not allowed. The return type must be void. Varargs, sret, byval, byref, inalloca, preallocated are not supported. Values in scalar registers as well as v0-v7 are not preserved. Values in VGPRs starting at v8 are not preserved for the active lanes, but must be saved by the callee for inactive lanes when using WWM. Wave scratch is "empty" at function boundaries. There is no stack pointer input or output value, but functions are free to use scratch starting from an initial stack pointer. Calls to `amdgpu_gfx` functions are allowed and behave like they do in `amdgpu_cs` functions. All counters (`lgkmcnt`, `vmcnt`, `storecnt`, etc.) are presumed in an unknown state at function entry. A function may have multiple exits (e.g. one chain exit and one plain `ret void` for when the wave ends), but all `llvm.amdgcn.cs.chain` exits must be in uniform control flow.
`amdgpu_cs_chain_preserve`	Same as `amdgpu_cs_chain`, but active lanes for VGPRs starting at v8 are preserved.
`amdgpu_es`	Used for AMDPAL shader stage before geometry shader if geometry is in use. So either the domain (= tessellation evaluation) shader if tessellation is in use, or otherwise the vertex shader. ..TODO:: Describe.
`amdgpu_gfx`	Used for AMD graphics targets. Functions with this calling convention cannot be used as entry points. ..TODO:: Describe.
`amdgpu_gs`	Used for Mesa/AMDPAL geometry shaders. ..TODO:: Describe.
`amdgpu_hs`	Used for Mesa/AMDPAL hull shaders (= tessellation control shaders). ..TODO:: Describe.

`amdgpu_kernel`	See Kernel Functions
`amdgpu_ls`	Used for AMDPAL vertex shader if tessellation is in use. ..TODO:: Describe.
`amdgpu_ps`	Used for Mesa/AMDPAL pixel shaders. ..TODO:: Describe.
`amdgpu_vs`	Used for Mesa/AMDPAL last shader stage before rasterization (vertex shader if tessellation and geometry are not in use, or otherwise copy shader if one is needed). ..TODO:: Describe.

ELF Code Object

The AMDGPU backend generates a standard ELF [ELF] relocatable code object that can be linked by `lld` to produce a standard ELF shared code object which can be loaded and executed on an AMDGPU target.

Header

The AMDGPU backend uses the following ELF header:

AMDGPU ELF Header

Field	Value
`e_ident[EI_CLASS]`	`ELFCLASS64`
`e_ident[EI_DATA]`	`ELFDATA2LSB`
`e_ident[EI_OSABI]`	• `ELFOSABI_NONE` • `ELFOSABI_AMDGPU_HSA` • `ELFOSABI_AMDGPU_PAL` • `ELFOSABI_AMDGPU_MESA3D`
`e_ident[EI_ABIVERSION]`	• `ELFABIVERSION_AMDGPU_HSA_V2` • `ELFABIVERSION_AMDGPU_HSA_V3` • `ELFABIVERSION_AMDGPU_HSA_V4` • `ELFABIVERSION_AMDGPU_HSA_V5` • `ELFABIVERSION_AMDGPU_PAL` • `ELFABIVERSION_AMDGPU_MESA3D`
`e_type`	• `ET_REL` • `ET_DYN`

e_machine	EM_AMDGPU
e_entry	0
e_flags	See AMDGPU ELF Header e_flags for Code Object V2, AMDGPU ELF Header e_flags for Code Object V3, and AMDGPU ELF Header e_flags for Code Object V4 and After

AMDGPU ELF Header Enumeration Values

Name	Value
EM_AMDGPU	224
ELFOSABI_NONE	0
ELFOSABI_AMDGPU_HSA	64
ELFOSABI_AMDGPU_PAL	65
ELFOSABI_AMDGPU_MESA3D	66
ELFABIVERSION_AMDGPU_HSA_V2	0
ELFABIVERSION_AMDGPU_HSA_V3	1
ELFABIVERSION_AMDGPU_HSA_V4	2
ELFABIVERSION_AMDGPU_HSA_V5	3
ELFABIVERSION_AMDGPU_PAL	0
ELFABIVERSION_AMDGPU_MESA3D	0

e_ident[EI_CLASS]

The ELF class is:

- ELFCLASS32 for r600 architecture.
- ELFCLASS64 for amdgcn architecture which only supports 64-bit process address space applications.

e_ident[EI_DATA]

All AMDGPU targets use ELFDATA2LSB for little-endian byte ordering.

e_ident[EI_OSABI]

One of the following AMDGPU target architecture specific OS ABIs (see AMDGPU Operating Systems):

- ELFOSABI_NONE for *unknown* OS.
- ELFOSABI_AMDGPU_HSA for amdhsa OS.
- ELFOSABI_AMDGPU_PAL for amdpal OS.
- ELFOSABI_AMDGPU_MESA3D for mesa3D OS.

`e_ident[EI_ABIVERSION]`

The ABI version of the AMDGPU target architecture specific OS ABI to which the code object conforms:

- `ELFABIVERSION_AMDGPU_HSA_V2` is used to specify the version of AMD HSA runtime ABI for code object V2. Specify using the Clang option `-mcode-object-version=2`.
- `ELFABIVERSION_AMDGPU_HSA_V3` is used to specify the version of AMD HSA runtime ABI for code object V3. Specify using the Clang option `-mcode-object-version=3`.
- `ELFABIVERSION_AMDGPU_HSA_V4` is used to specify the version of AMD HSA runtime ABI for code object V4. Specify using the Clang option `-mcode-object-version=4`. This is the default code object version if not specified.
- `ELFABIVERSION_AMDGPU_HSA_V5` is used to specify the version of AMD HSA runtime ABI for code object V5. Specify using the Clang option `-mcode-object-version=5`.
- `ELFABIVERSION_AMDGPU_PAL` is used to specify the version of AMD PAL runtime ABI.
- `ELFABIVERSION_AMDGPU_MESA3D` is used to specify the version of AMD MESA 3D runtime ABI.

`e_type`

Can be one of the following values:

`ET_REL`

The type produced by the AMDGPU backend compiler as it is relocatable code object.

`ET_DYN`

The type produced by the linker as it is a shared code object.

The AMD HSA runtime loader requires a `ET_DYN` code object.

`e_machine`

The value `EM_AMDGPU` is used for the machine for all processors supported by the `r600` and `amdgcn` architectures (see AMDGPU Processors). The specific processor is specified in the `NT_AMD_HSA_ISA_VERSION` note record for code object V2 (see Code Object V2 Note Records) and in the `EF_AMDGPU_MACH` bit field of the `e_flags` for code object V3 and above (see AMDGPU ELF Header e_flags for Code Object V3 and AMDGPU ELF Header e_flags for Code Object V4 and After).

`e_entry`

The entry point is 0 as the entry points for individual kernels must be selected in order to invoke them through AQL packets.

`e_flags`

The AMDGPU backend uses the following ELF header flags:

AMDGPU ELF Header `e_flags` for Code Object V2

Name	Value	Description
`EF_AMDGPU_FEATURE_XNACK_V2`	0x01	Indicates if the `xnack` target feature is enabled for all code contained in the code object. If the processor does not support the `xnack` target feature then must be 0. See Target Features.
`EF_AMDGPU_FEATURE_TRAP_HANDLER_V2`	0x02	Indicates if the trap handler is enabled for all code contained in the code object. If the processor does not support a trap handler then must be 0. See Target Features.

AMDGPU ELF Header `e_flags` for Code Object V3

Name	Value	Description
`EF_AMDGPU_MACH`	0x0ff	AMDGPU processor selection mask for `EF_AMDGPU_MACH_xxx` values defined in AMDGPU EF_AMDGPU_MACH Values.
`EF_AMDGPU_FEATURE_XNACK_V3`	0x100	Indicates if the `xnack` target feature is enabled for all code contained in the code object. If the processor does not support the `xnack` target feature then must be 0. See Target Features.
`EF_AMDGPU_FEATURE_SRAMECC_V3`	0x200	Indicates if the `sramecc` target feature is enabled for all code contained in the code object. If the processor does not support the `sramecc` target feature then must be 0. See Target Features.

AMDGPU ELF Header `e_flags` for Code Object V4 and After

Name	Value	Description
`EF_AMDGPU_MACH`	0x0ff	AMDGPU processor selection mask for `EF_AMDGPU_MACH_xxx` values defined in AMDGPU EF_AMDGPU_MACH Values.
`EF_AMDGPU_FEATURE_XNACK_V4`	0x300	XNACK selection mask for

		`EF_AMDGPU_FEATURE_XNACK_*_V4` values.
`EF_AMDGPU_FEATURE_XNACK_UNSUPPORTED_V4`	0x000	XNACK unsupported.
`EF_AMDGPU_FEATURE_XNACK_ANY_V4`	0x100	XNACK can have any value.
`EF_AMDGPU_FEATURE_XNACK_OFF_V4`	0x200	XNACK disabled.
`EF_AMDGPU_FEATURE_XNACK_ON_V4`	0x300	XNACK enabled.
`EF_AMDGPU_FEATURE_SRAMECC_V4`	0xc00	SRAMECC selection mask for `EF_AMDGPU_FEATURE_SRAMECC_*_V4` values.
`EF_AMDGPU_FEATURE_SRAMECC_UNSUPPORTED_V4`	0x000	SRAMECC unsupported.
`EF_AMDGPU_FEATURE_SRAMECC_ANY_V4`	0x400	SRAMECC can have any value.
`EF_AMDGPU_FEATURE_SRAMECC_OFF_V4`	0x800	SRAMECC disabled,
`EF_AMDGPU_FEATURE_SRAMECC_ON_V4`	0xc00	SRAMECC enabled.

AMDGPU `EF_AMDGPU_MACH` Values

Name	Value	Description (see AMDGPU Processors)
`EF_AMDGPU_MACH_NONE`	0x000	*not specified*
`EF_AMDGPU_MACH_R600_R600`	0x001	`r600`
`EF_AMDGPU_MACH_R600_R630`	0x002	`r630`
`EF_AMDGPU_MACH_R600_RS880`	0x003	`rs880`
`EF_AMDGPU_MACH_R600_RV670`	0x004	`rv670`
`EF_AMDGPU_MACH_R600_RV710`	0x005	`rv710`
`EF_AMDGPU_MACH_R600_RV730`	0x006	`rv730`
`EF_AMDGPU_MACH_R600_RV770`	0x007	`rv770`
`EF_AMDGPU_MACH_R600_CEDAR`	0x008	`cedar`
`EF_AMDGPU_MACH_R600_CYPRESS`	0x009	`cypress`
`EF_AMDGPU_MACH_R600_JUNIPER`	0x00a	`juniper`
`EF_AMDGPU_MACH_R600_REDWOOD`	0x00b	`redwood`
`EF_AMDGPU_MACH_R600_SUMO`	0x00c	`sumo`
`EF_AMDGPU_MACH_R600_BARTS`	0x00d	`barts`
`EF_AMDGPU_MACH_R600_CAICOS`	0x00e	`caicos`
`EF_AMDGPU_MACH_R600_CAYMAN`	0x00f	`cayman`
`EF_AMDGPU_MACH_R600_TURKS`	0x010	`turks`
reserved	0x011 -	Reserved for `r600` architecture

	0x01f	processors.
EF_AMDGPU_MACH_AMDGCN_GFX600	0x020	gfx600
EF_AMDGPU_MACH_AMDGCN_GFX601	0x021	gfx601
EF_AMDGPU_MACH_AMDGCN_GFX700	0x022	gfx700
EF_AMDGPU_MACH_AMDGCN_GFX701	0x023	gfx701
EF_AMDGPU_MACH_AMDGCN_GFX702	0x024	gfx702
EF_AMDGPU_MACH_AMDGCN_GFX703	0x025	gfx703
EF_AMDGPU_MACH_AMDGCN_GFX704	0x026	gfx704
reserved	0x027	Reserved.
EF_AMDGPU_MACH_AMDGCN_GFX801	0x028	gfx801
EF_AMDGPU_MACH_AMDGCN_GFX802	0x029	gfx802
EF_AMDGPU_MACH_AMDGCN_GFX803	0x02a	gfx803
EF_AMDGPU_MACH_AMDGCN_GFX810	0x02b	gfx810
EF_AMDGPU_MACH_AMDGCN_GFX900	0x02c	gfx900
EF_AMDGPU_MACH_AMDGCN_GFX902	0x02d	gfx902
EF_AMDGPU_MACH_AMDGCN_GFX904	0x02e	gfx904
EF_AMDGPU_MACH_AMDGCN_GFX906	0x02f	gfx906
EF_AMDGPU_MACH_AMDGCN_GFX908	0x030	gfx908
EF_AMDGPU_MACH_AMDGCN_GFX909	0x031	gfx909
EF_AMDGPU_MACH_AMDGCN_GFX90C	0x032	gfx90c
EF_AMDGPU_MACH_AMDGCN_GFX1010	0x033	gfx1010
EF_AMDGPU_MACH_AMDGCN_GFX1011	0x034	gfx1011
EF_AMDGPU_MACH_AMDGCN_GFX1012	0x035	gfx1012
EF_AMDGPU_MACH_AMDGCN_GFX1030	0x036	gfx1030
EF_AMDGPU_MACH_AMDGCN_GFX1031	0x037	gfx1031
EF_AMDGPU_MACH_AMDGCN_GFX1032	0x038	gfx1032
EF_AMDGPU_MACH_AMDGCN_GFX1033	0x039	gfx1033
EF_AMDGPU_MACH_AMDGCN_GFX602	0x03a	gfx602
EF_AMDGPU_MACH_AMDGCN_GFX705	0x03b	gfx705
EF_AMDGPU_MACH_AMDGCN_GFX805	0x03c	gfx805
EF_AMDGPU_MACH_AMDGCN_GFX1035	0x03d	gfx1035
EF_AMDGPU_MACH_AMDGCN_GFX1034	0x03e	gfx1034
EF_AMDGPU_MACH_AMDGCN_GFX90A	0x03f	gfx90a

EF_AMDGPU_MACH_AMDGCN_GFX940	0x040	gfx940
EF_AMDGPU_MACH_AMDGCN_GFX1100	0x041	gfx1100
EF_AMDGPU_MACH_AMDGCN_GFX1013	0x042	gfx1013
reserved	0x043	Reserved.
EF_AMDGPU_MACH_AMDGCN_GFX1103	0x044	gfx1103
EF_AMDGPU_MACH_AMDGCN_GFX1036	0x045	gfx1036
EF_AMDGPU_MACH_AMDGCN_GFX1101	0x046	gfx1101
EF_AMDGPU_MACH_AMDGCN_GFX1102	0x047	gfx1102
reserved	0x048	Reserved.
reserved	0x049	Reserved.
reserved	0x04a	Reserved.
EF_AMDGPU_MACH_AMDGCN_GFX941	0x04b	gfx941
EF_AMDGPU_MACH_AMDGCN_GFX942	0x04c	gfx942

Sections

An AMDGPU target ELF code object has the standard ELF sections which include:

AMDGPU ELF Sections

Name	Type	Attributes
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.debug_*	SHT_PROGBITS	*none*
.dynamic	SHT_DYNAMIC	SHF_ALLOC
.dynstr	SHT_PROGBITS	SHF_ALLOC
.dynsym	SHT_PROGBITS	SHF_ALLOC
.got	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.hash	SHT_HASH	SHF_ALLOC
.note	SHT_NOTE	*none*
.rela*name*	SHT_RELA	*none*
.rela.dyn	SHT_RELA	*none*
.rodata	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	*none*
.strtab	SHT_STRTAB	*none*

.symtab	SHT_SYMTAB	*none*
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR

These sections have their standard meanings (see [ELF]) and are only generated if needed.

`.debug*`

The standard DWARF sections. See DWARF Debug Information for information on the DWARF produced by the AMDGPU backend.

`.dynamic`, `.dynstr`, `.dynsym`, `.hash`

The standard sections used by a dynamic loader.

`.note`

See Note Records for the note records supported by the AMDGPU backend.

`.rela`*name*, `.rela.dyn`

For relocatable code objects, *name* is the name of the section that the relocation records apply. For example, `.rela.text` is the section name for relocation records associated with the `.text` section.

For linked shared code objects, `.rela.dyn` contains all the relocation records from each of the relocatable code object's `.rela`*name* sections.

See Relocation Records for the relocation records supported by the AMDGPU backend.

`.text`

The executable machine code for the kernels and functions they call. Generated as position independent code. See Code Conventions for information on conventions used in the isa generation.

Note Records

The AMDGPU backend code object contains ELF note records in the `.note` section. The set of generated notes and their semantics depend on the code object version; see Code Object V2 Note Records and Code Object V3 and Above Note Records.

As required by `ELFCLASS32` and `ELFCLASS64`, minimal zero-byte padding must be generated after the `name` field to ensure the `desc` field is 4 byte aligned. In addition, minimal zero-byte padding must be generated to ensure the `desc` field size is a multiple of 4 bytes. The `sh_addralign` field of the `.note` section must be at least 4 to indicate at least 8 byte alignment.

Code Object V2 Note Records

Warning

Code object V2 is not the default code object version emitted by this version of LLVM.

The AMDGPU backend code object uses the following ELF note record in the `.note` section when compiling for code object V2.

The note record vendor field is “AMD”.

Additional note records may be present, but any which are not documented here are deprecated and should not be used.

AMDGPU Code Object V2 ELF Note Records

Name	Type	Description
“AMD”	`NT_AMD_HSA_CODE_OBJECT_VERSION`	Code object version.
“AMD”	`NT_AMD_HSA_HSAIL`	HSAIL properties generated by the HSAIL Finalizer and not the LLVM compiler.
“AMD”	`NT_AMD_HSA_ISA_VERSION`	Target ISA version.
“AMD”	`NT_AMD_HSA_METADATA`	Metadata null terminated string in YAML [YAML] textual format.
“AMD”	`NT_AMD_HSA_ISA_NAME`	Target ISA name.

AMDGPU Code Object V2 ELF Note Record Enumeration Values

Name	Value
`NT_AMD_HSA_CODE_OBJECT_VERSION`	1
`NT_AMD_HSA_HSAIL`	2
`NT_AMD_HSA_ISA_VERSION`	3
reserved	4-9
`NT_AMD_HSA_METADATA`	10
`NT_AMD_HSA_ISA_NAME`	11

`NT_AMD_HSA_CODE_OBJECT_VERSION`

Specifies the code object version number. The description field has the following layout:

```
struct amdgpu_hsa_note_code_object_version_s {
  uint32_t major_version;
```

```
  uint32_t minor_version;
};
```

The `major_version` has a value less than or equal to 2.

`NT_AMD_HSA_HSAIL`

Specifies the HSAIL properties used by the HSAIL Finalizer. The description field has the following layout:

```
struct amdgpu_hsa_note_hsail_s {
  uint32_t hsail_major_version;
  uint32_t hsail_minor_version;
  uint8_t profile;
  uint8_t machine_model;
  uint8_t default_float_round;
};
```

`NT_AMD_HSA_ISA_VERSION`

Specifies the target ISA version. The description field has the following layout:

```
struct amdgpu_hsa_note_isa_s {
  uint16_t vendor_name_size;
  uint16_t architecture_name_size;
  uint32_t major;
  uint32_t minor;
  uint32_t stepping;
  char vendor_and_architecture_name[1];
};
```

`vendor_name_size` and `architecture_name_size` are the length of the vendor and architecture names respectively, including the NUL character.

`vendor_and_architecture_name` contains the NUL terminates string for the vendor, immediately followed by the NUL terminated string for the architecture.

This note record is used by the HSA runtime loader.

Code object V2 only supports a limited number of processors and has fixed settings for target features. See AMDGPU Code Object V2 Supported Processors and Fixed Target Feature Settings for a list of processors and the corresponding target ID. In the table the note record ISA name is a concatenation of the vendor name, architecture name, major, minor, and stepping separated by a ":".

The target ID column shows the processor name and fixed target features used by the LLVM compiler. The LLVM compiler does not generate a `NT_AMD_HSA_HSAIL` note record.

A code object generated by the Finalizer also uses code object V2 and always generates a `NT_AMD_HSA_HSAIL` note record. The processor name and `sramecc` target feature is as shown in AMDGPU Code Object V2 Supported Processors and Fixed Target Feature Settings but the `xnack` target feature is specified by the `EF_AMDGPU_FEATURE_XNACK_V2` `e_flags` bit.

`NT_AMD_HSA_ISA_NAME`

Specifies the target ISA name as a non-NUL terminated string.

This note record is not used by the HSA runtime loader.

See the `NT_AMD_HSA_ISA_VERSION` note record description of the code object V2's limited support of processors and fixed settings for target features.

See AMDGPU Code Object V2 Supported Processors and Fixed Target Feature Settings for a mapping from the string to the corresponding target ID. If the `xnack` target feature is supported and enabled, the string produced by the LLVM compiler will may have a `+xnack` appended. The Finlizer did not do the appending and instead used the `EF_AMDGPU_FEATURE_XNACK_V2` `e_flags` bit.

`NT_AMD_HSA_METADATA`

Specifies extensible metadata associated with the code objects executed on HSA [HSA] compatible runtimes (see AMDGPU Operating Systems). It is required when the target triple OS is `amdhsa` (see Target Triples). See Code Object V2 Metadata for the syntax of the code object metadata string.

AMDGPU Code Object V2 Supported Processors and Fixed Target Feature Settings

Note Record ISA Name	Target ID
`AMD:AMDGPU:6:0:0`	`gfx600`
`AMD:AMDGPU:6:0:1`	`gfx601`
`AMD:AMDGPU:6:0:2`	`gfx602`
`AMD:AMDGPU:7:0:0`	`gfx700`
`AMD:AMDGPU:7:0:1`	`gfx701`
`AMD:AMDGPU:7:0:2`	`gfx702`
`AMD:AMDGPU:7:0:3`	`gfx703`
`AMD:AMDGPU:7:0:4`	`gfx704`
`AMD:AMDGPU:7:0:5`	`gfx705`
`AMD:AMDGPU:8:0:0`	`gfx802`
`AMD:AMDGPU:8:0:1`	`gfx801:xnack+`
`AMD:AMDGPU:8:0:2`	`gfx802`

AMD:AMDGPU:8:0:3	gfx803
AMD:AMDGPU:8:0:4	gfx803
AMD:AMDGPU:8:0:5	gfx805
AMD:AMDGPU:8:1:0	gfx810:xnack+
AMD:AMDGPU:9:0:0	gfx900:xnack-
AMD:AMDGPU:9:0:1	gfx900:xnack+
AMD:AMDGPU:9:0:2	gfx902:xnack-
AMD:AMDGPU:9:0:3	gfx902:xnack+
AMD:AMDGPU:9:0:4	gfx904:xnack-
AMD:AMDGPU:9:0:5	gfx904:xnack+
AMD:AMDGPU:9:0:6	gfx906:sramecc-:xnack-
AMD:AMDGPU:9:0:7	gfx906:sramecc-:xnack+
AMD:AMDGPU:9:0:12	gfx90c:xnack-

Code Object V3 and Above Note Records

The AMDGPU backend code object uses the following ELF note record in the `.note` section when compiling for code object V3 and above.

The note record vendor field is “AMDGPU”.

Additional note records may be present, but any which are not documented here are deprecated and should not be used.

AMDGPU Code Object V3 and Above ELF Note Records

Name	Type	Description
“AMDGPU”	`NT_AMDGPU_METADATA`	Metadata in Message Pack [MsgPack] binary format.

AMDGPU Code Object V3 and Above ELF Note Record Enumeration Values

Name	Value
reserved	0-31
`NT_AMDGPU_METADATA`	32

`NT_AMDGPU_METADATA`

Specifies extensible metadata associated with an AMDGPU code object. It is encoded as a map

in the Message Pack [MsgPack] binary data format. See Code Object V3 Metadata, Code Object V4 Metadata and Code Object V5 Metadata for the map keys defined for the `amdhsa` OS.

Symbols

Symbols include the following:

AMDGPU ELF Symbols

Name	Type	Section	Description
link-name	`STT_OBJECT`	• `.data` • `.rodata` • `.bss`	Global variable
link-name`.kd`	`STT_OBJECT`	• `.rodata`	Kernel descriptor
link-name	`STT_FUNC`	• `.text`	Kernel entry point
link-name	`STT_OBJECT`	• SHN_AMDGPU_LDS	Global variable in LDS

Global variable

Global variables both used and defined by the compilation unit.

If the symbol is defined in the compilation unit then it is allocated in the appropriate section according to if it has initialized data or is readonly.

If the symbol is external then its section is `STN_UNDEF` and the loader will resolve relocations using the definition provided by another code object or explicitly defined by the runtime.

If the symbol resides in local/group memory (LDS) then its section is the special processor specific section name `SHN_AMDGPU_LDS`, and the `st_value` field describes alignment requirements as it does for common symbols.

Kernel descriptor

Every HSA kernel has an associated kernel descriptor. It is the address of the kernel descriptor that is used in the AQL dispatch packet used to invoke the kernel, not the kernel entry point. The layout of the HSA kernel descriptor is defined in Kernel Descriptor.

Kernel entry point

Every HSA kernel also has a symbol for its machine code entry point.

Relocation Records

AMDGPU backend generates `Elf64_Rela` relocation records. Supported relocatable fields are:

`word32`
: This specifies a 32-bit field occupying 4 bytes with arbitrary byte alignment. These values use the same byte order as other word values in the AMDGPU architecture.

`word64`
: This specifies a 64-bit field occupying 8 bytes with arbitrary byte alignment. These values use the same byte order as other word values in the AMDGPU architecture.

Following notations are used for specifying relocation calculations:

A
: Represents the addend used to compute the value of the relocatable field.

G
: Represents the offset into the global offset table at which the relocation entry's symbol will reside during execution.

GOT
: Represents the address of the global offset table.

P
: Represents the place (section offset for `et_rel` or address for `et_dyn`) of the storage unit being relocated (computed using `r_offset`).

S
: Represents the value of the symbol whose index resides in the relocation entry. Relocations not using this must specify a symbol index of `STN_UNDEF`.

B
: Represents the base address of a loaded executable or shared object which is the difference between the ELF address and the actual load address. Relocations using this are only valid in executable or shared objects.

The following relocation types are supported:

AMDGPU ELF Relocation Records

Relocation Type	Kind	Value	Field	Calculation

R_AMDGPU_NONE		0	*none*	*none*
R_AMDGPU_ABS32_LO	Static, Dynamic	1	word32	(S + A) & 0xFFFFFFFF
R_AMDGPU_ABS32_HI	Static, Dynamic	2	word32	(S + A) >> 32
R_AMDGPU_ABS64	Static, Dynamic	3	word64	S + A
R_AMDGPU_REL32	Static	4	word32	S + A - P
R_AMDGPU_REL64	Static	5	word64	S + A - P
R_AMDGPU_ABS32	Static, Dynamic	6	word32	S + A
R_AMDGPU_GOTPCREL	Static	7	word32	G + GOT + A - P
R_AMDGPU_GOTPCREL32_LO	Static	8	word32	(G + GOT + A - P) & 0xFFFFFFFF
R_AMDGPU_GOTPCREL32_HI	Static	9	word32	(G + GOT + A - P) >> 32
R_AMDGPU_REL32_LO	Static	10	word32	(S + A - P) & 0xFFFFFFFF
R_AMDGPU_REL32_HI	Static	11	word32	(S + A - P) >> 32
reserved		12		
R_AMDGPU_RELATIVE64	Dynamic	13	word64	B + A
R_AMDGPU_REL16	Static	14	word16	((S + A - P) - 4) / 4

R_AMDGPU_ABS32_LO and R_AMDGPU_ABS32_HI are only supported by the mesa3d OS, which does not support R_AMDGPU_ABS64.

There is no current OS loader support for 32-bit programs and so R_AMDGPU_ABS32 is not used.

Loaded Code Object Path Uniform Resource Identifier (URI)

The AMD GPU code object loader represents the path of the ELF shared object from which the code object was loaded as a textual Uniform Resource Identifier (URI). Note that the code object is the in memory loaded relocated form of the ELF shared object. Multiple code objects may be loaded at different memory addresses in the same process from the same ELF shared object.

The loaded code object path URI syntax is defined by the following BNF syntax:

```
code_object_uri ::== file_uri | memory_uri
file_uri        ::== "file://" file_path [ range_specifier ]
memory_uri      ::== "memory://" process_id range_specifier
```

```
range_specifier ::== [ "#" | "?" ] "offset=" number "&" "size=" number
file_path       ::== URI_ENCODED_OS_FILE_PATH
process_id      ::== DECIMAL_NUMBER
number          ::== HEX_NUMBER | DECIMAL_NUMBER | OCTAL_NUMBER
```

number

Is a C integral literal where hexadecimal values are prefixed by “0x” or “0X”, and octal values by “0”.

file_path

Is the file’s path specified as a URI encoded UTF-8 string. In URI encoding, every character that is not in the regular expression `[a-zA-Z0-9/_.~-]` is encoded as two uppercase hexadecimal digits proceeded by “%”. Directories in the path are separated by “/”.

offset

Is a 0-based byte offset to the start of the code object. For a file URI, it is from the start of the file specified by the `file_path`, and if omitted defaults to 0. For a memory URI, it is the memory address and is required.

size

Is the number of bytes in the code object. For a file URI, if omitted it defaults to the size of the file. It is required for a memory URI.

process_id

Is the identity of the process owning the memory. For Linux it is the C unsigned integral decimal literal for the process ID (PID).

For example:

```
file:///dir1/dir2/file1
file:///dir3/dir4/file2#offset=0x2000&size=3000
memory://1234#offset=0x20000&size=3000
```

DWARF Debug Information

Warning

This section describes provisional support for AMDGPU DWARF [DWARF] that is not currently fully implemented and is subject to change.

AMDGPU generates DWARF [DWARF] debugging information ELF sections (see ELF Code Object) which contain information that maps the code object executable code and data to the source

language constructs. It can be used by tools such as debuggers and profilers. It uses features defined in DWARF Extensions For Heterogeneous Debugging that are made available in DWARF Version 4 and DWARF Version 5 as an LLVM vendor extension.

This section defines the AMDGPU target architecture specific DWARF mappings.

Register Identifier

This section defines the AMDGPU target architecture register numbers used in DWARF operation expressions (see DWARF Version 5 section 2.5 and A.2.5.4 DWARF Operation Expressions) and Call Frame Information instructions (see DWARF Version 5 section 6.4 and A.6.4 Call Frame Information).

A single code object can contain code for kernels that have different wavefront sizes. The vector registers and some scalar registers are based on the wavefront size. AMDGPU defines distinct DWARF registers for each wavefront size. This simplifies the consumer of the DWARF so that each register has a fixed size, rather than being dynamic according to the wavefront size mode. Similarly, distinct DWARF registers are defined for those registers that vary in size according to the process address size. This allows a consumer to treat a specific AMDGPU processor as a single architecture regardless of how it is configured at run time. The compiler explicitly specifies the DWARF registers that match the mode in which the code it is generating will be executed.

DWARF registers are encoded as numbers, which are mapped to architecture registers. The mapping for AMDGPU is defined in AMDGPU DWARF Register Mapping. All AMDGPU targets use the same mapping.

AMDGPU DWARF Register Mapping

DWARF Register	AMDGPU Register	Bit Size	Description
0	PC_32	32	Program Counter (PC) when executing in a 32-bit process address space. Used in the CFI to describe the PC of the calling frame.
1	EXEC_MASK_32	32	Execution Mask Register when executing in wavefront 32 mode.
2-15	*Reserved*		*Reserved for highly accessed registers using DWARF shortcut.*
16	PC_64	64	Program Counter (PC) when executing in a 64-bit process address space. Used in the CFI to describe the PC of the calling frame.
17	EXEC_MASK_64	64	Execution Mask Register when executing in

			wavefront 64 mode.
18-31	*Reserved*		*Reserved for highly accessed registers using DWARF shortcut.*
32-95	SGPR0-SGPR63	32	Scalar General Purpose Registers.
96-127	*Reserved*		*Reserved for frequently accessed registers using DWARF 1-byte ULEB.*
128	STATUS	32	Status Register.
129-511	*Reserved*		*Reserved for future Scalar Architectural Registers.*
512	VCC_32	32	Vector Condition Code Register when executing in wavefront 32 mode.
513-767	*Reserved*		*Reserved for future Vector Architectural Registers when executing in wavefront 32 mode.*
768	VCC_64	64	Vector Condition Code Register when executing in wavefront 64 mode.
769-1023	*Reserved*		*Reserved for future Vector Architectural Registers when executing in wavefront 64 mode.*
1024-1087	*Reserved*		*Reserved for padding.*
1088-1129	SGPR64-SGPR105	32	Scalar General Purpose Registers.
1130-1535	*Reserved*		*Reserved for future Scalar General Purpose Registers.*
1536-1791	VGPR0-VGPR255	32*32	Vector General Purpose Registers when executing in wavefront 32 mode.
1792-2047	*Reserved*		*Reserved for future Vector General Purpose Registers when executing in wavefront 32 mode.*
2048-2303	AGPR0-AGPR255	32*32	Vector Accumulation Registers when executing in wavefront 32 mode.
2304-2559	*Reserved*		*Reserved for future Vector Accumulation Registers when executing in wavefront 32 mode.*
2560-2815	VGPR0-VGPR255	64*32	Vector General Purpose Registers when executing in wavefront 64 mode.

2816-3071	*Reserved*		*Reserved for future Vector General Purpose Registers when executing in wavefront 64 mode.*
3072-3327	AGPR0-AGPR255	64*32	Vector Accumulation Registers when executing in wavefront 64 mode.
3328-3583	*Reserved*		*Reserved for future Vector Accumulation Registers when executing in wavefront 64 mode.*

The vector registers are represented as the full size for the wavefront. They are organized as consecutive dwords (32-bits), one per lane, with the dword at the least significant bit position corresponding to lane 0 and so forth. DWARF location expressions involving the `DW_OP_LLVM_offset` and `DW_OP_LLVM_push_lane` operations are used to select the part of the vector register corresponding to the lane that is executing the current thread of execution in languages that are implemented using a SIMD or SIMT execution model.

If the wavefront size is 32 lanes then the wavefront 32 mode register definitions are used. If the wavefront size is 64 lanes then the wavefront 64 mode register definitions are used. Some AMDGPU targets support executing in both wavefront 32 and wavefront 64 mode. The register definitions corresponding to the wavefront mode of the generated code will be used.

If code is generated to execute in a 32-bit process address space, then the 32-bit process address space register definitions are used. If code is generated to execute in a 64-bit process address space, then the 64-bit process address space register definitions are used. The `amdgcn` target only supports the 64-bit process address space.

Memory Space Identifier

The DWARF memory space represents the source language memory space. See DWARF Version 5 section 2.12 which is updated by the *DWARF Extensions For Heterogeneous Debugging* section A.2.14 Memory Spaces.

The DWARF memory space mapping used for AMDGPU is defined in AMDGPU DWARF Memory Space Mapping.

AMDGPU DWARF Memory Space Mapping

DWARF		AMDGPU
Memory Space Name	Value	Memory Space
`DW_MSPACE_LLVM_none`	0x0000	Generic (Flat)
`DW_MSPACE_LLVM_global`	0x0001	Global

DW_MSPACE_LLVM_constant	0x0002	Global
DW_MSPACE_LLVM_group	0x0003	Local (group/LDS)
DW_MSPACE_LLVM_private	0x0004	Private (Scratch)
DW_MSPACE_AMDGPU_region	0x8000	Region (GDS)

The DWARF memory space values defined in the *DWARF Extensions For Heterogeneous Debugging* section A.2.14 Memory Spaces are used.

In addition, DW_ADDR_AMDGPU_region is encoded as a vendor extension. This is available for use for the AMD extension for access to the hardware GDS memory which is scratchpad memory allocated per device.

For AMDGPU if no DW_AT_LLVM_memory_space attribute is present, then the default memory space of DW_MSPACE_LLVM_none is used.

See Address Space Identifier for information on the AMDGPU mapping of DWARF memory spaces to DWARF address spaces, including address size and NULL value.

Address Space Identifier

DWARF address spaces correspond to target architecture specific linear addressable memory areas. See DWARF Version 5 section 2.12 and *DWARF Extensions For Heterogeneous Debugging* section A.2.13 Address Spaces.

The DWARF address space mapping used for AMDGPU is defined in AMDGPU DWARF Address Space Mapping.

AMDGPU DWARF Address Space Mapping

DWARF				AMDGPU	Notes
Address Space Name	Value	Address	Bit Size	LLVM IR Address Space	
		64-bit process address space	32-bit process address space		
DW_ASPACE_LLVM_none	0x00	64	32	Global	*default address space*
DW_ASPACE_AMDGPU_generic	0x01	64	32	Generic (Flat)	
DW_ASPACE_AMDGPU_region	0x02	32	32	Region (GDS)	
DW_ASPACE_AMDGPU_local	0x03	32	32	Local	

				(group/LDS)	
Reserved	0x04				
`DW_ASPACE_AMDGPU_private_lane`	0x05	32	32	Private (Scratch)	*focused lane*
`DW_ASPACE_AMDGPU_private_wave`	0x06	32	32	Private (Scratch)	*unswizzled wavefront*

See Address Spaces for information on the AMDGPU LLVM IR address spaces including address size and NULL value.

The `DW_ASPACE_LLVM_none` address space is the default target architecture address space used in DWARF operations that do not specify an address space. It therefore has to map to the global address space so that the `DW_OP_addr*` and related operations can refer to addresses in the program code.

The `DW_ASPACE_AMDGPU_generic` address space allows location expressions to specify the flat address space. If the address corresponds to an address in the local address space, then it corresponds to the wavefront that is executing the focused thread of execution. If the address corresponds to an address in the private address space, then it corresponds to the lane that is executing the focused thread of execution for languages that are implemented using a SIMD or SIMT execution model.

Note

CUDA-like languages such as HIP that do not have address spaces in the language type system, but do allow variables to be allocated in different address spaces, need to explicitly specify the `DW_ASPACE_AMDGPU_generic` address space in the DWARF expression operations as the default address space is the global address space.

The `DW_ASPACE_AMDGPU_local` address space allows location expressions to specify the local address space corresponding to the wavefront that is executing the focused thread of execution.

The `DW_ASPACE_AMDGPU_private_lane` address space allows location expressions to specify the private address space corresponding to the lane that is executing the focused thread of execution for languages that are implemented using a SIMD or SIMT execution model.

The `DW_ASPACE_AMDGPU_private_wave` address space allows location expressions to specify the unswizzled private address space corresponding to the wavefront that is executing the focused thread of execution. The wavefront view of private memory is the per wavefront unswizzled backing memory layout defined in Address Spaces, such that address 0 corresponds to the first location for the backing memory of the wavefront (namely the address is not offset by `wavefront-scratch-base`). The following formula can be used to convert from a `DW_ASPACE_AMDGPU_private_lane` address to a `DW_ASPACE_AMDGPU_private_wave` address:

```
private-address-wavefront =
  ((private-address-lane / 4) * wavefront-size * 4) +
  (wavefront-lane-id * 4) + (private-address-lane % 4)
```

If the `DW_ASPACE_AMDGPU_private_lane` address is dword aligned, and the start of the dwords for each lane starting with lane 0 is required, then this simplifies to:

```
private-address-wavefront =
  private-address-lane * wavefront-size
```

A compiler can use the `DW_ASPACE_AMDGPU_private_wave` address space to read a complete spilled vector register back into a complete vector register in the CFI. The frame pointer can be a private lane address which is dword aligned, which can be shifted to multiply by the wavefront size, and then used to form a private wavefront address that gives a location for a contiguous set of dwords, one per lane, where the vector register dwords are spilled. The compiler knows the wavefront size since it generates the code. Note that the type of the address may have to be converted as the size of a `DW_ASPACE_AMDGPU_private_lane` address may be smaller than the size of a `DW_ASPACE_AMDGPU_private_wave` address.

Lane identifier

DWARF lane identifies specify a target architecture lane position for hardware that executes in a SIMD or SIMT manner, and on which a source language maps its threads of execution onto those lanes. The DWARF lane identifier is pushed by the `DW_OP_LLVM_push_lane` DWARF expression operation. See DWARF Version 5 section 2.5 which is updated by *DWARF Extensions For Heterogeneous Debugging* section A.2.5.4 DWARF Operation Expressions.

For AMDGPU, the lane identifier corresponds to the hardware lane ID of a wavefront. It is numbered from 0 to the wavefront size minus 1.

Operation Expressions

DWARF expressions are used to compute program values and the locations of program objects. See DWARF Version 5 section 2.5 and A.2.5.4 DWARF Operation Expressions.

DWARF location descriptions describe how to access storage which includes memory and registers. When accessing storage on AMDGPU, bytes are ordered with least significant bytes first, and bits are ordered within bytes with least significant bits first.

For AMDGPU CFI expressions, `DW_OP_LLVM_select_bit_piece` is used to describe unwinding vector

registers that are spilled under the execution mask to memory: the zero-single location description is the vector register, and the one-single location description is the spilled memory location description. The `DW_OP_LLVM_form_aspace_address` is used to specify the address space of the memory location description.

In AMDGPU expressions, `DW_OP_LLVM_select_bit_piece` is used by the `DW_AT_LLVM_lane_pc` attribute expression where divergent control flow is controlled by the execution mask. An undefined location description together with `DW_OP_LLVM_extend` is used to indicate the lane was not active on entry to the subprogram. See DW_AT_LLVM_lane_pc for an example.

Debugger Information Entry Attributes

This section describes how certain debugger information entry attributes are used by AMDGPU. See the sections in DWARF Version 5 section 3.3.5 and 3.1.1 which are updated by *DWARF Extensions For Heterogeneous Debugging* section A.3.3.5 Low-Level Information and A.3.1.1 Full and Partial Compilation Unit Entries.

`DW_AT_LLVM_lane_pc`

For AMDGPU, the `DW_AT_LLVM_lane_pc` attribute is used to specify the program location of the separate lanes of a SIMT thread.

If the lane is an active lane then this will be the same as the current program location.

If the lane is inactive, but was active on entry to the subprogram, then this is the program location in the subprogram at which execution of the lane is conceptual positioned.

If the lane was not active on entry to the subprogram, then this will be the undefined location. A client debugger can check if the lane is part of a valid work-group by checking that the lane is in the range of the associated work-group within the grid, accounting for partial work-groups. If it is not, then the debugger can omit any information for the lane. Otherwise, the debugger may repeatedly unwind the stack and inspect the `DW_AT_LLVM_lane_pc` of the calling subprogram until it finds a non-undefined location. Conceptually the lane only has the call frames that it has a non-undefined `DW_AT_LLVM_lane_pc`.

The following example illustrates how the AMDGPU backend can generate a DWARF location list expression for the nested `IF/THEN/ELSE` structures of the following subprogram pseudo code for a target with 64 lanes per wavefront.

```
1 SUBPROGRAM X
2 BEGIN
```

```
 3   a;
 4   IF (c1) THEN
 5     b;
 6     IF (c2) THEN
 7       c;
 8     ELSE
 9       d;
10     ENDIF
11     e;
12   ELSE
13     f;
14   ENDIF
15   g;
16 END
```

The AMDGPU backend may generate the following pseudo LLVM MIR to manipulate the execution mask (`EXEC`) to linearize the control flow. The condition is evaluated to make a mask of the lanes for which the condition evaluates to true. First the `THEN` region is executed by setting the `EXEC` mask to the logical `AND` of the current `EXEC` mask with the condition mask. Then the `ELSE` region is executed by negating the `EXEC` mask and logical `AND` of the saved `EXEC` mask at the start of the region. After the `IF/THEN/ELSE` region the `EXEC` mask is restored to the value it had at the beginning of the region. This is shown below. Other approaches are possible, but the basic concept is the same.

```
 1 $lex_start:
 2   a;
 3   %1 = EXEC
 4   %2 = c1
 5 $lex_1_start:
 6   EXEC = %1 & %2
 7 $if_1_then:
 8     b;
 9     %3 = EXEC
10     %4 = c2
11 $lex_1_1_start:
12     EXEC = %3 & %4
13 $lex_1_1_then:
14       c;
15     EXEC = ~EXEC & %3
16 $lex_1_1_else:
17       d;
18     EXEC = %3
19 $lex_1_1_end:
20     e;
21   EXEC = ~EXEC & %1
22 $lex_1_else:
23     f;
```

```
24   EXEC = %1
25 $lex_1_end:
26   g;
27 $lex_end:
```

To create the DWARF location list expression that defines the location description of a vector of lane program locations, the LLVM MIR `DBG_VALUE` pseudo instruction can be used to annotate the linearized control flow. This can be done by defining an artificial variable for the lane PC. The DWARF location list expression created for it is used as the value of the `DW_AT_LLVM_lane_pc` attribute on the subprogram's debugger information entry.

A DWARF procedure is defined for each well nested structured control flow region which provides the conceptual lane program location for a lane if it is not active (namely it is divergent). The DWARF operation expression for each region conceptually inherits the value of the immediately enclosing region and modifies it according to the semantics of the region.

For an `IF/THEN/ELSE` region the divergent program location is at the start of the region for the `THEN` region since it is executed first. For the `ELSE` region the divergent program location is at the end of the `IF/THEN/ELSE` region since the `THEN` region has completed.

The lane PC artificial variable is assigned at each region transition. It uses the immediately enclosing region's DWARF procedure to compute the program location for each lane assuming they are divergent, and then modifies the result by inserting the current program location for each lane that the `EXEC` mask indicates is active.

By having separate DWARF procedures for each region, they can be reused to define the value for any nested region. This reduces the total size of the DWARF operation expressions.

The following provides an example using pseudo LLVM MIR.

```
 1 $lex_start:
 2   DEFINE_DWARF %__uint_64 = DW_TAG_base_type[
 3     DW_AT_name = "__uint64";
 4     DW_AT_byte_size = 8;
 5     DW_AT_encoding = DW_ATE_unsigned;
 6   ];
 7   DEFINE_DWARF %__active_lane_pc = DW_TAG_dwarf_procedure[
 8     DW_AT_name = "__active_lane_pc";
 9     DW_AT_location = [
10       DW_OP_regx PC;
11       DW_OP_LLVM_extend 64, 64;
12       DW_OP_regval_type EXEC, %uint_64;
13       DW_OP_LLVM_select_bit_piece 64, 64;
14     ];
15   ];
```

```
  DEFINE_DWARF %__divergent_lane_pc = DW_TAG_dwarf_procedure[
    DW_AT_name = "__divergent_lane_pc";
    DW_AT_location = [
      DW_OP_LLVM_undefined;
      DW_OP_LLVM_extend 64, 64;
    ];
  ];
  DBG_VALUE $noreg, $noreg, %DW_AT_LLVM_lane_pc, DIExpression[
    DW_OP_call_ref %__divergent_lane_pc;
    DW_OP_call_ref %__active_lane_pc;
  ];
  a;
  %1 = EXEC;
  DBG_VALUE %1, $noreg, %__lex_1_save_exec;
  %2 = c1;
$lex_1_start:
  EXEC = %1 & %2;
$lex_1_then:
    DEFINE_DWARF %__divergent_lane_pc_1_then = DW_TAG_dwarf_procedure[
      DW_AT_name = "__divergent_lane_pc_1_then";
      DW_AT_location = DIExpression[
        DW_OP_call_ref %__divergent_lane_pc;
        DW_OP_addrx &lex_1_start;
        DW_OP_stack_value;
        DW_OP_LLVM_extend 64, 64;
        DW_OP_call_ref %__lex_1_save_exec;
        DW_OP_deref_type 64, %__uint_64;
        DW_OP_LLVM_select_bit_piece 64, 64;
      ];
    ];
    DBG_VALUE $noreg, $noreg, %DW_AT_LLVM_lane_pc, DIExpression[
      DW_OP_call_ref %__divergent_lane_pc_1_then;
      DW_OP_call_ref %__active_lane_pc;
    ];
    b;
    %3 = EXEC;
    DBG_VALUE %3, %__lex_1_1_save_exec;
    %4 = c2;
$lex_1_1_start:
    EXEC = %3 & %4;
$lex_1_1_then:
      DEFINE_DWARF %__divergent_lane_pc_1_1_then = DW_TAG_dwarf_procedure[
        DW_AT_name = "__divergent_lane_pc_1_1_then";
        DW_AT_location = DIExpression[
          DW_OP_call_ref %__divergent_lane_pc_1_then;
          DW_OP_addrx &lex_1_1_start;
          DW_OP_stack_value;
          DW_OP_LLVM_extend 64, 64;
          DW_OP_call_ref %__lex_1_1_save_exec;
          DW_OP_deref_type 64, %__uint_64;
```

```
66            DW_OP_LLVM_select_bit_piece 64, 64;
67          ];
68        ];
69        DBG_VALUE $noreg, $noreg, %DW_AT_LLVM_lane_pc, DIExpression[
70          DW_OP_call_ref %__divergent_lane_pc_1_1_then;
71          DW_OP_call_ref %__active_lane_pc;
72        ];
73        c;
74      EXEC = ~EXEC & %3;
75  $lex_1_1_else:
76        DEFINE_DWARF %__divergent_lane_pc_1_1_else = DW_TAG_dwarf_procedure[
77          DW_AT_name = "__divergent_lane_pc_1_1_else";
78          DW_AT_location = DIExpression[
79            DW_OP_call_ref %__divergent_lane_pc_1_then;
80            DW_OP_addrx &lex_1_1_end;
81            DW_OP_stack_value;
82            DW_OP_LLVM_extend 64, 64;
83            DW_OP_call_ref %__lex_1_1_save_exec;
84            DW_OP_deref_type 64, %__uint_64;
85            DW_OP_LLVM_select_bit_piece 64, 64;
86          ];
87        ];
88        DBG_VALUE $noreg, $noreg, %DW_AT_LLVM_lane_pc, DIExpression[
89          DW_OP_call_ref %__divergent_lane_pc_1_1_else;
90          DW_OP_call_ref %__active_lane_pc;
91        ];
92        d;
93      EXEC = %3;
94  $lex_1_1_end:
95      DBG_VALUE $noreg, $noreg, %DW_AT_LLVM_lane_pc, DIExpression[
96        DW_OP_call_ref %__divergent_lane_pc;
97        DW_OP_call_ref %__active_lane_pc;
98      ];
99      e;
100   EXEC = ~EXEC & %1;
101 $lex_1_else:
102     DEFINE_DWARF %__divergent_lane_pc_1_else = DW_TAG_dwarf_procedure[
103       DW_AT_name = "__divergent_lane_pc_1_else";
104       DW_AT_location = DIExpression[
105         DW_OP_call_ref %__divergent_lane_pc;
106         DW_OP_addrx &lex_1_end;
107         DW_OP_stack_value;
108         DW_OP_LLVM_extend 64, 64;
109         DW_OP_call_ref %__lex_1_save_exec;
110         DW_OP_deref_type 64, %__uint_64;
111         DW_OP_LLVM_select_bit_piece 64, 64;
112       ];
113     ];
114     DBG_VALUE $noreg, $noreg, %DW_AT_LLVM_lane_pc, DIExpression[
115       DW_OP_call_ref %__divergent_lane_pc_1_else;
```

```
116       DW_OP_call_ref %__active_lane_pc;
117     ];
118     f;
119   EXEC = %1;
120 $lex_1_end:
121   DBG_VALUE $noreg, $noreg, %DW_AT_LLVM_lane_pc DIExpression[
122     DW_OP_call_ref %__divergent_lane_pc;
123     DW_OP_call_ref %__active_lane_pc;
124   ];
125   g;
126 $lex_end:
```

The DWARF procedure `%__active_lane_pc` is used to update the lane pc elements that are active, with the current program location.

Artificial variables %__lex_1_save_exec and %__lex_1_1_save_exec are created for the execution masks saved on entry to a region. Using the `DBG_VALUE` pseudo instruction, location list entries will be created that describe where the artificial variables are allocated at any given program location. The compiler may allocate them to registers or spill them to memory.

The DWARF procedures for each region use the values of the saved execution mask artificial variables to only update the lanes that are active on entry to the region. All other lanes retain the value of the enclosing region where they were last active. If they were not active on entry to the subprogram, then will have the undefined location description.

Other structured control flow regions can be handled similarly. For example, loops would set the divergent program location for the region at the end of the loop. Any lanes active will be in the loop, and any lanes not active must have exited the loop.

An `IF/THEN/ELSEIF/ELSEIF/...` region can be treated as a nest of `IF/THEN/ELSE` regions.

The DWARF procedures can use the active lane artificial variable described in DW_AT_LLVM_active_lane rather than the actual `EXEC` mask in order to support whole or quad wavefront mode.

DW_AT_LLVM_active_lane

The `DW_AT_LLVM_active_lane` attribute on a subprogram debugger information entry is used to specify the lanes that are conceptually active for a SIMT thread.

The execution mask may be modified to implement whole or quad wavefront mode operations. For example, all lanes may need to temporarily be made active to execute a whole wavefront operation. Such regions would save the `EXEC` mask, update it to enable the necessary lanes, perform the operations, and then restore the `EXEC` mask from the saved value. While executing the whole

wavefront region, the conceptual execution mask is the saved value, not the `EXEC` value.

This is handled by defining an artificial variable for the active lane mask. The active lane mask artificial variable would be the actual `EXEC` mask for normal regions, and the saved execution mask for regions where the mask is temporarily updated. The location list expression created for this artificial variable is used to define the value of the `DW_AT_LLVM_active_lane` attribute.

DW_AT_LLVM_augmentation

For AMDGPU, the `DW_AT_LLVM_augmentation` attribute of a compilation unit debugger information entry has the following value for the augmentation string:

```
[amdgpu:v0.0]
```

The "vX.Y" specifies the major X and minor Y version number of the AMDGPU extensions used in the DWARF of the compilation unit. The version number conforms to [SEMVER].

Call Frame Information

DWARF Call Frame Information (CFI) describes how a consumer can virtually *unwind* call frames in a running process or core dump. See DWARF Version 5 section 6.4 and A.6.4 Call Frame Information.

For AMDGPU, the Common Information Entry (CIE) fields have the following values:

1. `augmentation` string contains the following null-terminated UTF-8 string:

   ```
   [amd:v0.0]
   ```

   The `vX.Y` specifies the major X and minor Y version number of the AMDGPU extensions used in this CIE or to the FDEs that use it. The version number conforms to [SEMVER].
2. `address_size` for the `Global` address space is defined in Address Space Identifier.
3. `segment_selector_size` is 0 as AMDGPU does not use a segment selector.
4. `code_alignment_factor` is 4 bytes.
5. `data_alignment_factor` is 4 bytes.
6. `return_address_register` is `PC_32` for 32-bit processes and `PC_64` for 64-bit processes defined in Register Identifier.
7. `initial_instructions` Since a subprogram X with fewer registers can be called from subprogram Y that has more allocated, X will not change any of the extra registers as it

cannot access them. Therefore, the default rule for all columns is `same value`.

For AMDGPU the register number follows the numbering defined in Register Identifier.

For AMDGPU the instructions are variable size. A consumer can subtract 1 from the return address to get the address of a byte within the call site instructions. See DWARF Version 5 section 6.4.4.

Accelerated Access

See DWARF Version 5 section 6.1.

Lookup By Name Section Header

See DWARF Version 5 section 6.1.1.4.1 and A.6.1.1 Lookup By Name.

For AMDGPU the lookup by name section header table:

`augmentation_string_size` (uword)

Set to the length of the `augmentation_string` value which is always a multiple of 4.

`augmentation_string` (sequence of UTF-8 characters)

Contains the following UTF-8 string null padded to a multiple of 4 bytes:

```
[amdgpu:v0.0]
```

The "vX.Y" specifies the major X and minor Y version number of the AMDGPU extensions used in the DWARF of this index. The version number conforms to [SEMVER].

> **Note**
>
> This is different to the DWARF Version 5 definition that requires the first 4 characters to be the vendor ID. But this is consistent with the other augmentation strings and does allow multiple vendor contributions. However, backwards compatibility may be more desirable.

Lookup By Address Section Header

See DWARF Version 5 section 6.1.2.

For AMDGPU the lookup by address section header table:

`address_size` (ubyte)

Match the address size for the `Global` address space defined in Address Space Identifier.

`segment_selector_size` (ubyte)

AMDGPU does not use a segment selector so this is 0. The entries in the `.debug_aranges` do not have a segment selector.

Line Number Information

See DWARF Version 5 section 6.2 and A.6.2 Line Number Information.

AMDGPU does not use the `isa` state machine registers and always sets it to 0. The instruction set must be obtained from the ELF file header `e_flags` field in the `EF_AMDGPU_MACH` bit position (see ELF Header). See DWARF Version 5 section 6.2.2.

For AMDGPU the line number program header fields have the following values (see DWARF Version 5 section 6.2.4):

`address_size` (ubyte)

Matches the address size for the `Global` address space defined in Address Space Identifier.

`segment_selector_size` (ubyte)

AMDGPU does not use a segment selector so this is 0.

`minimum_instruction_length` (ubyte)

For GFX9-GFX11 this is 4.

`maximum_operations_per_instruction` (ubyte)

For GFX9-GFX11 this is 1.

Source text for online-compiled programs (for example, those compiled by the OpenCL language runtime) may be embedded into the DWARF Version 5 line table. See DWARF Version 5 section 6.2.4.1 which is updated by *DWARF Extensions For Heterogeneous Debugging* section DW_LNCT_LLVM_source.

The Clang option used to control source embedding in AMDGPU is defined in AMDGPU Clang Debug Options.

AMDGPU Clang Debug Options

Debug Flag	Description
-g[no-]embed-source	Enable/disable embedding source text in DWARF debug sections. Useful for environments where source cannot be written to disk, such as when performing online compilation.

For example:

`-gembed-source`
: Enable the embedded source.

`-gno-embed-source`
: Disable the embedded source.

32-Bit and 64-Bit DWARF Formats

See DWARF Version 5 section 7.4 and A.7.4 32-Bit and 64-Bit DWARF Formats.

For AMDGPU:

- For the `amdgcn` target architecture only the 64-bit process address space is supported.
- The producer can generate either 32-bit or 64-bit DWARF format. LLVM generates the 32-bit DWARF format.

Unit Headers

For AMDGPU the following values apply for each of the unit headers described in DWARF Version 5 sections 7.5.1.1, 7.5.1.2, and 7.5.1.3:

`address_size` (ubyte)
: Matches the address size for the `Global` address space defined in Address Space Identifier.

Code Conventions

This section provides code conventions used for each supported target triple OS (see Target Triples).

AMDHSA

This section provides code conventions used when the target triple OS is `amdhsa` (see Target Triples).

Code Object Metadata

The code object metadata specifies extensible metadata associated with the code objects executed on HSA [HSA] compatible runtimes (see AMDGPU Operating Systems). The encoding and semantics of this metadata depends on the code object version; see Code Object V2 Metadata, Code Object V3 Metadata, Code Object V4 Metadata and Code Object V5 Metadata.

Code object metadata is specified in a note record (see Note Records) and is required when the target triple OS is `amdhsa` (see Target Triples). It must contain the minimum information necessary to support the HSA compatible runtime kernel queries. For example, the segment sizes needed in a dispatch packet. In addition, a high-level language runtime may require other information to be included. For example, the AMD OpenCL runtime records kernel argument information.

Code Object V2 Metadata

Warning

Code object V2 is not the default code object version emitted by this version of LLVM.

Code object V2 metadata is specified by the `NT_AMD_HSA_METADATA` note record (see Code Object V2 Note Records).

The metadata is specified as a YAML formatted string (see [YAML] and YAML I/O).

The metadata is represented as a single YAML document comprised of the mapping defined in table AMDHSA Code Object V2 Metadata Map and referenced tables.

For boolean values, the string values of `false` and `true` are used for false and true respectively.

Additional information can be added to the mappings. To avoid conflicts, any non-AMD key names should be prefixed by "*vendor-name*.".

AMDHSA Code Object V2 Metadata Map

String Key	Value Type	Required?	Description
"Version"	sequence of 2 integers	Required	• The first integer is the major version. Currently 1. • The second integer is the minor version. Currently 0.
"Printf"	sequence		Each string is encoded information about a printf

	of strings		function call. The encoded information is organized as fields separated by colon (':'):
			`ID:N:S[0]:S[1]:...:S[N-1]:FormatString`
			where:
			`ID`
			A 32-bit integer as a unique id for each printf function call
			`N`
			A 32-bit integer equal to the number of arguments of printf function call minus 1
			`S[i]` (where i = 0, 1, ... , N-1)
			32-bit integers for the size in bytes of the i-th FormatString argument of the printf function call
			FormatString
			The format string passed to the printf function call.

"Kernels"	sequence of mapping	Required	Sequence of the mappings for each kernel in the code object. See AMDHSA Code Object V2 Kernel Metadata Map for the definition of the mapping.

AMDHSA Code Object V2 Kernel Metadata Map

String Key	Value Type	Required?	Description
"Name"	string	Required	Source name of the kernel.
"SymbolName"	string	Required	Name of the kernel descriptor ELF symbol.
"Language"	string		Source language of the kernel. Values include: • "OpenCL C" • "OpenCL C++" • "HCC" • "OpenMP"

“LanguageVersion”	sequence of 2 integers	• The first integer is the major version. • The second integer is the minor version.
“Attrs”	mapping	Mapping of kernel attributes. See AMDHSA Code Object V2 Kernel Attribute Metadata Map for the mapping definition.
“Args”	sequence of mapping	Sequence of mappings of the kernel arguments. See AMDHSA Code Object V2 Kernel Argument Metadata Map for the definition of the mapping.
“CodeProps”	mapping	Mapping of properties related to the kernel code. See AMDHSA Code Object V2 Kernel Code Properties Metadata Map for the mapping definition.

AMDHSA Code Object V2 Kernel Attribute Metadata Map

String Key	Value Type	Required?	Description
“ReqdWorkGroupSize”	sequence of 3 integers		If not 0, 0, 0 then all values must be >=1 and the dispatch work-group size X, Y, Z must correspond to the specified values. Defaults to 0, 0, 0. Corresponds to the OpenCL `reqd_work_group_size` attribute.
“WorkGroupSizeHint”	sequence of 3 integers		The dispatch work-group size X, Y, Z is likely to be the specified values. Corresponds to the OpenCL `work_group_size_hint` attribute.
“VecTypeHint”	string		The name of a scalar or vector type. Corresponds to the OpenCL `vec_type_hint` attribute.
“RuntimeHandle”	string		The external symbol name

			associated with a kernel. OpenCL runtime allocates a global buffer for the symbol and saves the kernel's address to it, which is used for device side enqueueing. Only available for device side enqueued kernels.

AMDHSA Code Object V2 Kernel Argument Metadata Map

String Key	Value Type	Required?	Description
"Name"	string		Kernel argument name.
"TypeName"	string		Kernel argument type name.
"Size"	integer	Required	Kernel argument size in bytes.
"Align"	integer	Required	Kernel argument alignment in bytes. Must be a power of two.
"ValueKind"	string	Required	Kernel argument kind that specifies how to set up the corresponding argument. Values include: "ByValue" The argument is copied directly into the kernarg. "GlobalBuffer" A global address space pointer to the buffer data is passed in the kernarg. "DynamicSharedPointer" A group address space pointer to dynamically allocated LDS is passed in the kernarg. "Sampler" A global address space pointer to a S# is passed in the kernarg. "Image" A global address space pointer to a T# is passed in the kernarg.

“Pipe”
: A global address space pointer to an OpenCL pipe is passed in the kernarg.

“Queue”
: A global address space pointer to an OpenCL device enqueue queue is passed in the kernarg.

“HiddenGlobalOffsetX”
: The OpenCL grid dispatch global offset for the X dimension is passed in the kernarg.

“HiddenGlobalOffsetY”
: The OpenCL grid dispatch global offset for the Y dimension is passed in the kernarg.

“HiddenGlobalOffsetZ”
: The OpenCL grid dispatch global offset for the Z dimension is passed in the kernarg.

“HiddenNone”
: An argument that is not used by the kernel. Space needs to be left for it, but it does not need to be set up.

“HiddenPrintfBuffer”
: A global address space pointer to the runtime printf buffer is passed in kernarg. Mutually exclusive with “HiddenHostcallBuffer”.

“HiddenHostcallBuffer”
: A global address space pointer to the runtime hostcall buffer is passed in kernarg. Mutually exclusive with “HiddenPrintfBuffer”.

		"HiddenDefaultQueue" A global address space pointer to the OpenCL device enqueue queue that should be used by the kernel by default is passed in the kernarg. "HiddenCompletionAction" A global address space pointer to help link enqueued kernels into the ancestor tree for determining when the parent kernel has finished. "HiddenMultiGridSyncArg" A global address space pointer for multi-grid synchronization is passed in the kernarg.
"ValueType"	string	Unused and deprecated. This should no longer be emitted, but is accepted for compatibility.
"PointeeAlign"	integer	Alignment in bytes of pointee type for pointer type kernel argument. Must be a power of 2. Only present if "ValueKind" is "DynamicSharedPointer".
"AddrSpaceQual"	string	Kernel argument address space qualifier. Only present if "ValueKind" is "GlobalBuffer" or "DynamicSharedPointer". Values are: • "Private" • "Global" • "Constant" • "Local" • "Generic" • "Region"
"AccQual"	string	Kernel argument access qualifier. Only present if "ValueKind" is "Image" or "Pipe". Values are:

		"ReadOnly" • "WriteOnly" • "ReadWrite"
"ActualAccQual"	string	The actual memory accesses performed by the kernel on the kernel argument. Only present if "ValueKind" is "GlobalBuffer", "Image", or "Pipe". This may be more restrictive than indicated by "AccQual" to reflect what the kernel actual does. If not present then the runtime must assume what is implied by "AccQual" and "IsConst". Values are: • "ReadOnly" • "WriteOnly" • "ReadWrite"
"IsConst"	boolean	Indicates if the kernel argument is const qualified. Only present if "ValueKind" is "GlobalBuffer".
"IsRestrict"	boolean	Indicates if the kernel argument is restrict qualified. Only present if "ValueKind" is "GlobalBuffer".
"IsVolatile"	boolean	Indicates if the kernel argument is volatile qualified. Only present if "ValueKind" is "GlobalBuffer".
"IsPipe"	boolean	Indicates if the kernel argument is pipe qualified. Only present if "ValueKind" is "Pipe".

AMDHSA Code Object V2 Kernel Code Properties Metadata Map

String Key	Value Type	Required?	Description
"KernargSegmentSize"	integer	Required	The size in bytes of the kernarg segment that holds the values of the arguments to the kernel.

“GroupSegmentFixedSize”	integer	Required	The amount of group segment memory required by a work-group in bytes. This does not include any dynamically allocated group segment memory that may be added when the kernel is dispatched.
“PrivateSegmentFixedSize”	integer	Required	The amount of fixed private address space memory required for a work-item in bytes. If the kernel uses a dynamic call stack then additional space must be added to this value for the call stack.
“KernargSegmentAlign”	integer	Required	The maximum byte alignment of arguments in the kernarg segment. Must be a power of 2.
“WavefrontSize”	integer	Required	Wavefront size. Must be a power of 2.
“NumSGPRs”	integer	Required	Number of scalar registers used by a wavefront for GFX6-GFX11. This includes the special SGPRs for VCC, Flat Scratch (GFX7-GFX10) and XNACK (for GFX8-GFX10). It does not include the 16 SGPR added if a trap handler is enabled. It is not

			rounded up to the allocation granularity.
“NumVGPRs”	integer	Required	Number of vector registers used by each work-item for GFX6-GFX11
“MaxFlatWorkGroupSize”	integer	Required	Maximum flat work-group size supported by the kernel in work-items. Must be >=1 and consistent with ReqdWorkGroupSize if not 0, 0, 0.
“NumSpilledSGPRs”	integer		Number of stores from a scalar register to a register allocator created spill location.
“NumSpilledVGPRs”	integer		Number of stores from a vector register to a register allocator created spill location.

Code Object V3 Metadata

Warning

Code object V3 is not the default code object version emitted by this version of LLVM.

Code object V3 and above metadata is specified by the `NT_AMDGPU_METADATA` note record (see Code Object V3 and Above Note Records).

The metadata is represented as Message Pack formatted binary data (see [MsgPack]). The top level is a Message Pack map that includes the keys defined in table AMDHSA Code Object V3 Metadata Map and referenced tables.

Additional information can be added to the maps. To avoid conflicts, any key names should be prefixed by “*vendor-name*.” where `vendor-name` can be the name of the vendor and specific vendor tool that generates the information. The prefix is abbreviated to simply “.” when it appears within a map that has been added by the same *vendor-name*.

AMDHSA Code Object V3 Metadata Map

String Key	Value Type	Required?	Description
“amdhsa.version”	sequence of 2 integers	Required	• The first integer is the major version. Currently 1. • The second integer is the minor version. Currently 0.

“amdhsa.printf”	sequence of strings		Each string is encoded information about a printf function call. The encoded information is organized as fields separated by colon (‘:’):
			`ID:N:S[0]:S[1]:...:S[N-1]:FormatString`
			where:
			`ID`
			A 32-bit integer as a unique id for each printf function call
			`N`
			A 32-bit integer equal to the number of arguments of printf function call minus 1
			`S[i]` (where i = 0, 1, … , N-1)
			32-bit integers for the size in bytes of the i-th FormatString argument of the printf function call
			FormatString
			The format string passed to the printf function call.

“amdhsa.kernels”	sequence of map	Required	Sequence of the maps for each kernel in the code object. See AMDHSA Code Object V3 Kernel Metadata Map for the definition of the keys included in that map.

AMDHSA Code Object V3 Kernel Metadata Map

String Key	Value Type	Required?	Description
".name"	string	Required	Source name of the kernel.
".symbol"	string	Required	Name of the kernel descriptor ELF symbol.
".language"	string		Source language of the kernel. Values include: • "OpenCL C" • "OpenCL C++" • "HCC" • "HIP" • "OpenMP" • "Assembler"
".language_version"	sequence of 2 integers		• The first integer is the major version. • The second integer is the minor version.
".args"	sequence of map		Sequence of maps of the kernel arguments. See AMDHSA Code Object V3 Kernel Argument Metadata Map for the definition of the keys included in that map.
".reqd_workgroup_size"	sequence of 3 integers		If not 0, 0, 0 then all values must be >=1 and the dispatch work-group size X, Y, Z must correspond to the specified values. Defaults to 0, 0, 0. Corresponds to the OpenCL `reqd_work_group_size` attribute.
".workgroup_size_hint"	sequence of 3 integers		The dispatch work-group size X, Y, Z is likely to be the specified values. Corresponds to the OpenCL

			`work_group_size_hint` attribute.
".vec_type_hint"	string		The name of a scalar or vector type. Corresponds to the OpenCL `vec_type_hint` attribute.
".device_enqueue_symbol"	string		The external symbol name associated with a kernel. OpenCL runtime allocates a global buffer for the symbol and saves the kernel's address to it, which is used for device side enqueueing. Only available for device side enqueued kernels.
".kernarg_segment_size"	integer	Required	The size in bytes of the kernarg segment that holds the values of the arguments to the kernel.
".group_segment_fixed_size"	integer	Required	The amount of group segment memory required by a work-group in bytes. This does not include any dynamically allocated group segment memory that may be added when the kernel is dispatched.
".private_segment_fixed_size"	integer	Required	The amount of fixed private address space memory required for a work-item in bytes. If the kernel uses a dynamic call stack then additional space must be added to this value for the call stack.
".kernarg_segment_align"	integer	Required	The maximum byte alignment of arguments in the kernarg segment. Must be a power of 2.
".wavefront_size"	integer	Required	Wavefront size. Must be a power of 2.

".sgpr_count"	integer	Required	Number of scalar registers required by a wavefront for GFX6-GFX9. A register is required if it is used explicitly, or if a higher numbered register is used explicitly. This includes the special SGPRs for VCC, Flat Scratch (GFX7-GFX9) and XNACK (for GFX8-GFX9). It does not include the 16 SGPR added if a trap handler is enabled. It is not rounded up to the allocation granularity.
".vgpr_count"	integer	Required	Number of vector registers required by each work-item for GFX6-GFX9. A register is required if it is used explicitly, or if a higher numbered register is used explicitly.
".agpr_count"	integer	Required	Number of accumulator registers required by each work-item for GFX90A, GFX908.
".max_flat_workgroup_size"	integer	Required	Maximum flat work-group size supported by the kernel in work-items. Must be >=1 and consistent with ReqdWorkGroupSize if not 0, 0, 0.
".sgpr_spill_count"	integer		Number of stores from a scalar register to a register allocator created spill location.
".vgpr_spill_count"	integer		Number of stores from a vector register to a register allocator created spill location.
".kind"	string		The kind of the kernel with the following values:

“normal”

Regular kernels.

“init”

These kernels must be invoked after loading the containing code object and must complete before any normal and fini kernels in the same code object are invoked.

“fini”

These kernels must be invoked before unloading the containing code object and after all init and normal kernels in the same code object have been invoked and completed.

If omitted, “normal” is assumed.

AMDHSA Code Object V3 Kernel Argument Metadata Map

String Key	Value Type	Required?	Description
“.name”	string		Kernel argument name.
“.type_name”	string		Kernel argument type name.
“.size”	integer	Required	Kernel argument size in bytes.
“.offset”	integer	Required	Kernel argument offset in bytes. The offset must be a multiple of the alignment required by the argument.
“.value_kind”	string	Required	Kernel argument kind that specifies how to set up the corresponding argument. Values include: “by_value” The argument is copied directly into the kernarg.

“global_buffer”
: A global address space pointer to the buffer data is passed in the kernarg.

“dynamic_shared_pointer”
: A group address space pointer to dynamically allocated LDS is passed in the kernarg.

“sampler”
: A global address space pointer to a S# is passed in the kernarg.

“image”
: A global address space pointer to a T# is passed in the kernarg.

“pipe”
: A global address space pointer to an OpenCL pipe is passed in the kernarg.

“queue”
: A global address space pointer to an OpenCL device enqueue queue is passed in the kernarg.

“hidden_global_offset_x”
: The OpenCL grid dispatch global offset for the X dimension is passed in the kernarg.

“hidden_global_offset_y”
: The OpenCL grid dispatch global offset for the Y dimension is passed in the kernarg.

“hidden_global_offset_z”
: The OpenCL grid dispatch global offset for the Z dimension is passed in the kernarg.

		"hidden_none" An argument that is not used by the kernel. Space needs to be left for it, but it does not need to be set up. "hidden_printf_buffer" A global address space pointer to the runtime printf buffer is passed in kernarg. Mutually exclusive with "hidden_hostcall_buffer" before Code Object V5. "hidden_hostcall_buffer" A global address space pointer to the runtime hostcall buffer is passed in kernarg. Mutually exclusive with "hidden_printf_buffer" before Code Object V5. "hidden_default_queue" A global address space pointer to the OpenCL device enqueue queue that should be used by the kernel by default is passed in the kernarg. "hidden_completion_action" A global address space pointer to help link enqueued kernels into the ancestor tree for determining when the parent kernel has finished. "hidden_multigrid_sync_arg" A global address space pointer for multi-grid synchronization is passed in the kernarg.
".value_type"	string	Unused and deprecated. This should no longer be emitted, but is accepted for compatibility.

".pointee_align"	integer	Alignment in bytes of pointee type for pointer type kernel argument. Must be a power of 2. Only present if ".value_kind" is "dynamic_shared_pointer".
".address_space"	string	Kernel argument address space qualifier. Only present if ".value_kind" is "global_buffer" or "dynamic_shared_pointer". Values are: • "private" • "global" • "constant" • "local" • "generic" • "region"
".access"	string	Kernel argument access qualifier. Only present if ".value_kind" is "image" or "pipe". Values are: • "read_only" • "write_only" • "read_write"
".actual_access"	string	The actual memory accesses performed by the kernel on the kernel argument. Only present if ".value_kind" is "global_buffer", "image", or "pipe". This may be more restrictive than indicated by ".access" to reflect what the kernel actual does. If not present then the runtime must assume what is implied by ".access" and ".is_const" . Values are: • "read_only" • "write_only" • "read_write"
".is_const"	boolean	Indicates if the kernel argument is const

		qualified. Only present if ".value_kind" is "global_buffer".
".is_restrict"	boolean	Indicates if the kernel argument is restrict qualified. Only present if ".value_kind" is "global_buffer".
".is_volatile"	boolean	Indicates if the kernel argument is volatile qualified. Only present if ".value_kind" is "global_buffer".
".is_pipe"	boolean	Indicates if the kernel argument is pipe qualified. Only present if ".value_kind" is "pipe".

Code Object V4 Metadata

Code object V4 metadata is the same as Code Object V3 Metadata with the changes and additions defined in table AMDHSA Code Object V4 Metadata Map Changes.

AMDHSA Code Object V4 Metadata Map Changes

String Key	Value Type	Required?	Description
"amdhsa.version"	sequence of 2 integers	Required	• The first integer is the major version. Currently 1. • The second integer is the minor version. Currently 1.
"amdhsa.target"	string	Required	The target name of the code using the syntax: `<target-triple> [ "-" <target-id> ]` A canonical target ID must be used. See Target Triples and Target ID.

Code Object V5 Metadata

Warning

Code object V5 is not the default code object version emitted by this version of LLVM.

Code object V5 metadata is the same as Code Object V4 Metadata with the changes defined in

table AMDHSA Code Object V5 Metadata Map Changes, table AMDHSA Code Object V5 Kernel Metadata Map Additions and table AMDHSA Code Object V5 Kernel Argument Metadata Map Additions and Changes.

AMDHSA Code Object V5 Metadata Map Changes

String Key	Value Type	Required?	Description
“amdhsa.version”	sequence of 2 integers	Required	• The first integer is the major version. Currently 1. • The second integer is the minor version. Currently 2.

AMDHSA Code Object V5 Kernel Metadata Map Additions

String Key	Value Type	Required?	Description
“.uses_dynamic_stack”	boolean		Indicates if the generated machine code is using a dynamically sized stack.
“.workgroup_processor_mode”	boolean		(GFX10+) Controls ENABLE_WGP_MODE in Code Object V3 Kernel Descriptor.

AMDHSA Code Object V5 Kernel Attribute Metadata Map

String Key	Value Type	Required?	Description
“.uniform_work_group_size”	integer		Indicates if the kernel requires that each dimension of global size is a multiple of corresponding dimension of work-group size. Value of 1 implies true and value of 0 implies false. Metadata is only emitted when value is 1.

AMDHSA Code Object V5 Kernel Argument Metadata Map Additions and Changes

String Key	Value Type	Required?	Description
“.value_kind”	string	Required	Kernel argument kind that specifies how to set up the corresponding argument. Values include: the

same as code object V3 metadata (see AMDHSA Code Object V3 Kernel Argument Metadata Map) with the following additions:

"hidden_block_count_x"

The grid dispatch work-group count for the X dimension is passed in the kernarg. Some languages, such as OpenCL, support a last work-group in each dimension being partial. This count only includes the non-partial work-group count. This is not the same as the value in the AQL dispatch packet, which has the grid size in work-items.

"hidden_block_count_y"

The grid dispatch work-group count for the Y dimension is passed in the kernarg. Some languages, such as OpenCL, support a last work-group in each dimension being partial. This count only includes the non-partial work-group count. This is not the same as the value in the AQL dispatch packet, which has the grid size in work-items. If the grid dimensionality is 1, then must be 1.

"hidden_block_count_z"

The grid dispatch work-group count for the Z dimension is passed in the kernarg. Some languages, such as OpenCL, support a last work-group in each dimension being partial. This count only includes the non-partial work-group count. This is not the same as the value in the AQL dispatch packet, which has the grid size in work-items. If the grid dimensionality is 1 or 2, then must be 1.

"hidden_group_size_x"

The grid dispatch work-group size for the X dimension is passed in the kernarg. This size

only applies to the non-partial work-groups. This is the same value as the AQL dispatch packet work-group size.

“hidden_group_size_y”

The grid dispatch work-group size for the Y dimension is passed in the kernarg. This size only applies to the non-partial work-groups. This is the same value as the AQL dispatch packet work-group size. If the grid dimensionality is 1, then must be 1.

“hidden_group_size_z”

The grid dispatch work-group size for the Z dimension is passed in the kernarg. This size only applies to the non-partial work-groups. This is the same value as the AQL dispatch packet work-group size. If the grid dimensionality is 1 or 2, then must be 1.

“hidden_remainder_x”

The grid dispatch work group size of the partial work group of the X dimension, if it exists. Must be zero if a partial work group does not exist in the X dimension.

“hidden_remainder_y”

The grid dispatch work group size of the partial work group of the Y dimension, if it exists. Must be zero if a partial work group does not exist in the Y dimension.

“hidden_remainder_z”

The grid dispatch work group size of the partial work group of the Z dimension, if it exists. Must be zero if a partial work group does not exist in the Z dimension.

“hidden_grid_dims”

The grid dispatch dimensionality. This is the

same value as the AQL dispatch packet dimensionality. Must be a value between 1 and 3.

"hidden_heap_v1"

A global address space pointer to an initialized memory buffer that conforms to the requirements of the malloc/free device library V1 version implementation.

"hidden_private_base"

The high 32 bits of the flat addressing private aperture base. Only used by GFX8 to allow conversion between private segment and flat addresses. See Flat Scratch.

"hidden_shared_base"

The high 32 bits of the flat addressing shared aperture base. Only used by GFX8 to allow conversion between shared segment and flat addresses. See Flat Scratch.

"hidden_queue_ptr"

A global memory address space pointer to the ROCm runtime `struct amd_queue_t` structure for the HSA queue of the associated dispatch AQL packet. It is only required for pre-GFX9 devices for the trap handler ABI (see Trap Handler ABI).

Kernel Dispatch

The HSA architected queuing language (AQL) defines a user space memory interface that can be used to control the dispatch of kernels, in an agent independent way. An agent can have zero or more AQL queues created for it using an HSA compatible runtime (see AMDGPU Operating Systems), in which AQL packets (all of which are 64 bytes) can be placed. See the *HSA Platform System Architecture Specification* [HSA] for the AQL queue mechanics and packet layouts.

The packet processor of a kernel agent is responsible for detecting and dispatching HSA kernels

from the AQL queues associated with it. For AMD GPUs the packet processor is implemented by the hardware command processor (CP), asynchronous dispatch controller (ADC) and shader processor input controller (SPI).

An HSA compatible runtime can be used to allocate an AQL queue object. It uses the kernel mode driver to initialize and register the AQL queue with CP.

To dispatch a kernel the following actions are performed. This can occur in the CPU host program, or from an HSA kernel executing on a GPU.

1. A pointer to an AQL queue for the kernel agent on which the kernel is to be executed is obtained.
2. A pointer to the kernel descriptor (see Kernel Descriptor) of the kernel to execute is obtained. It must be for a kernel that is contained in a code object that was loaded by an HSA compatible runtime on the kernel agent with which the AQL queue is associated.
3. Space is allocated for the kernel arguments using the HSA compatible runtime allocator for a memory region with the kernarg property for the kernel agent that will execute the kernel. It must be at least 16-byte aligned.
4. Kernel argument values are assigned to the kernel argument memory allocation. The layout is defined in the *HSA Programmer's Language Reference* [HSA]. For AMDGPU the kernel execution directly accesses the kernel argument memory in the same way constant memory is accessed. (Note that the HSA specification allows an implementation to copy the kernel argument contents to another location that is accessed by the kernel.)
5. An AQL kernel dispatch packet is created on the AQL queue. The HSA compatible runtime api uses 64-bit atomic operations to reserve space in the AQL queue for the packet. The packet must be set up, and the final write must use an atomic store release to set the packet kind to ensure the packet contents are visible to the kernel agent. AQL defines a doorbell signal mechanism to notify the kernel agent that the AQL queue has been updated. These rules, and the layout of the AQL queue and kernel dispatch packet is defined in the *HSA System Architecture Specification* [HSA].
6. A kernel dispatch packet includes information about the actual dispatch, such as grid and work-group size, together with information from the code object about the kernel, such as segment sizes. The HSA compatible runtime queries on the kernel symbol can be used to obtain the code object values which are recorded in the Code Object Metadata.
7. CP executes micro-code and is responsible for detecting and setting up the GPU to execute the wavefronts of a kernel dispatch.
8. CP ensures that when the a wavefront starts executing the kernel machine code, the scalar general purpose registers (SGPR) and vector general purpose registers (VGPR) are set up as required by the machine code. The required setup is defined in the Kernel Descriptor. The

initial register state is defined in Initial Kernel Execution State.

9. The prolog of the kernel machine code (see Kernel Prolog) sets up the machine state as necessary before continuing executing the machine code that corresponds to the kernel.
10. When the kernel dispatch has completed execution, CP signals the completion signal specified in the kernel dispatch packet if not 0.

Memory Spaces

The memory space properties are:

AMDHSA Memory Spaces

Memory Space Name	HSA Segment Name	Hardware Name	Address Size	NULL Value
Private	private	scratch	32	0x00000000
Local	group	LDS	32	0xFFFFFFFF
Global	global	global	64	0x0000000000000000
Constant	constant	*same as global*	64	0x0000000000000000
Generic	flat	flat	64	0x0000000000000000
Region	N/A	GDS	32	*not implemented for AMDHSA*

The global and constant memory spaces both use global virtual addresses, which are the same virtual address space used by the CPU. However, some virtual addresses may only be accessible to the CPU, some only accessible by the GPU, and some by both.

Using the constant memory space indicates that the data will not change during the execution of the kernel. This allows scalar read instructions to be used. The vector and scalar L1 caches are invalidated of volatile data before each kernel dispatch execution to allow constant memory to change values between kernel dispatches.

The local memory space uses the hardware Local Data Store (LDS) which is automatically allocated when the hardware creates work-groups of wavefronts, and freed when all the wavefronts of a work-group have terminated. The data store (DS) instructions can be used to access it.

The private memory space uses the hardware scratch memory support. If the kernel uses scratch, then the hardware allocates memory that is accessed using wavefront lane dword (4 byte) interleaving. The mapping used from private address to physical address is:

```
wavefront-scratch-base + (private-address * wavefront-size * 4) + (wavefront-lane-
```

```
id * 4)
```

There are different ways that the wavefront scratch base address is determined by a wavefront (see Initial Kernel Execution State). This memory can be accessed in an interleaved manner using buffer instruction with the scratch buffer descriptor and per wavefront scratch offset, by the scratch instructions, or by flat instructions. If each lane of a wavefront accesses the same private address, the interleaving results in adjacent dwords being accessed and hence requires fewer cache lines to be fetched. Multi-dword access is not supported except by flat and scratch instructions in GFX9-GFX11.

The generic address space uses the hardware flat address support available in GFX7-GFX11. This uses two fixed ranges of virtual addresses (the private and local apertures), that are outside the range of addressible global memory, to map from a flat address to a private or local address.

FLAT instructions can take a flat address and access global, private (scratch) and group (LDS) memory depending on if the address is within one of the aperture ranges. Flat access to scratch requires hardware aperture setup and setup in the kernel prologue (see Flat Scratch). Flat access to LDS requires hardware aperture setup and M0 (GFX7-GFX8) register setup (see M0).

To convert between a segment address and a flat address the base address of the apertures address can be used. For GFX7-GFX8 these are available in the HSA AQL Queue the address of which can be obtained with Queue Ptr SGPR (see Initial Kernel Execution State). For GFX9-GFX11 the aperture base addresses are directly available as inline constant registers `SRC_SHARED_BASE/LIMIT` and `SRC_PRIVATE_BASE/LIMIT`. In 64 bit address mode the aperture sizes are 2^32 bytes and the base is aligned to 2^32 which makes it easier to convert from flat to segment or segment to flat.

Image and Samplers

Image and sample handles created by an HSA compatible runtime (see AMDGPU Operating Systems) are 64-bit addresses of a hardware 32-byte V# and 48 byte S# object respectively. In order to support the HSA `query_sampler` operations two extra dwords are used to store the HSA BRIG enumeration values for the queries that are not trivially deducible from the S# representation.

HSA Signals

HSA signal handles created by an HSA compatible runtime (see AMDGPU Operating Systems) are 64-bit addresses of a structure allocated in memory accessible from both the CPU and GPU. The structure is defined by the runtime and subject to change between releases. For example, see [AMD-ROCm-github].

HSA AQL Queue

The HSA AQL queue structure is defined by an HSA compatible runtime (see AMDGPU Operating Systems) and subject to change between releases. For example, see [AMD-ROCm-github]. For some processors it contains fields needed to implement certain language features such as the flat address aperture bases. It also contains fields used by CP such as managing the allocation of scratch memory.

Kernel Descriptor

A kernel descriptor consists of the information needed by CP to initiate the execution of a kernel, including the entry point address of the machine code that implements the kernel.

Code Object V3 Kernel Descriptor

CP microcode requires the Kernel descriptor to be allocated on 64-byte alignment.

The fields used by CP for code objects before V3 also match those specified in Code Object V3 Kernel Descriptor.

Code Object V3 Kernel Descriptor

Bits	Size	Field Name	Description
31:0	4 bytes	GROUP_SEGMENT_FIXED_SIZE	The amount of fixed local address space memory required for a work-group in bytes. This does not include any dynamically allocated local address space memory that may be added when the kernel is dispatched.
63:32	4 bytes	PRIVATE_SEGMENT_FIXED_SIZE	The amount of fixed private address space memory required for a work-item in bytes. When this cannot be predicted, code object v4 and older sets this value to be higher than the minimum requirement.
95:64	4 bytes	KERNARG_SIZE	The size of the kernarg memory pointed to by the AQL dispatch packet. The kernarg memory is used to pass arguments to the kernel. • If the kernarg pointer in the dispatch

			packet is NULL then there are no kernel arguments. • If the kernarg pointer in the dispatch packet is not NULL and this value is 0 then the kernarg memory size is unspecified. • If the kernarg pointer in the dispatch packet is not NULL and this value is not 0 then the value specifies the kernarg memory size in bytes. It is recommended to provide a value as it may be used by CP to optimize making the kernarg memory visible to the kernel code.
127:96	4 bytes		Reserved, must be 0.
191:128	8 bytes	KERNEL_CODE_ENTRY_BYTE_OFFSET	Byte offset (possibly negative) from base address of kernel descriptor to kernel's entry point instruction which must be 256 byte aligned.
351:272	20 bytes		Reserved, must be 0.
383:352	4 bytes	COMPUTE_PGM_RSRC3	GFX6-GFX9 Reserved, must be 0. GFX90A, GFX940 Compute Shader (CS) program settings used by CP to set up `COMPUTE_PGM_RSRC3` configuration register. See compute_pgm_rsrc3 for GFX90A, GFX940. GFX10-GFX11 Compute Shader (CS) program settings used by CP to set up `COMPUTE_PGM_RSRC3` configuration register. See compute_pgm_rsrc3 for GFX10-GFX11.

415:384	4 bytes	COMPUTE_PGM_RSRC1	Compute Shader (CS) program settings used by CP to set up `COMPUTE_PGM_RSRC1` configuration register. See compute_pgm_rsrc1 for GFX6-GFX11.
447:416	4 bytes	COMPUTE_PGM_RSRC2	Compute Shader (CS) program settings used by CP to set up `COMPUTE_PGM_RSRC2` configuration register. See compute_pgm_rsrc2 for GFX6-GFX11.
458:448	7 bits	*See separate bits below.*	Enable the setup of the SGPR user data registers (see Initial Kernel Execution State). The total number of SGPR user data registers requested must not exceed 16 and match value in `compute_pgm_rsrc2.user_sgpr.user_sgpr_count`. Any requests beyond 16 will be ignored.
>448	1 bit	ENABLE_SGPR_PRIVATE_SEGMENT _BUFFER	If the *Target Properties* column of AMDGPU Processors specifies *Architected flat scratch* then not supported and must be 0,
>449	1 bit	ENABLE_SGPR_DISPATCH_PTR	
>450	1 bit	ENABLE_SGPR_QUEUE_PTR	
>451	1 bit	ENABLE_SGPR_KERNARG_SEGMENT_PTR	
>452	1 bit	ENABLE_SGPR_DISPATCH_ID	
>453	1 bit	ENABLE_SGPR_FLAT_SCRATCH_INIT	If the *Target Properties* column of AMDGPU Processors specifies *Architected flat scratch* then not supported and must be 0,
>454	1 bit	ENABLE_SGPR_PRIVATE_SEGMENT _SIZE	
457:455	3 bits		Reserved, must be 0.
458	1 bit	ENABLE_WAVEFRONT_SIZE32	GFX6-GFX9 Reserved, must be 0. GFX10-GFX11 • If 0 execute in wavefront size 64 mode. • If 1 execute in native wavefront size 32 mode.

459	1 bit	USES_DYNAMIC_STACK	Indicates if the generated machine code is using a dynamically sized stack. This is only set in code object v5 and later.
463:460	1 bit		Reserved, must be 0.
464	1 bit	RESERVED_464	Deprecated, must be 0.
467:465	3 bits		Reserved, must be 0.
468	1 bit	RESERVED_468	Deprecated, must be 0.
469:471	3 bits		Reserved, must be 0.
511:472	5 bytes		Reserved, must be 0.
512	Total size 64 bytes.		

compute_pgm_rsrc1 for GFX6-GFX11

Bits	Size	Field Name	Description
5:0	6 bits	GRANULATED_WORKITEM_VGPR_COUNT	Number of vector register blocks used by each work-item; granularity is device specific: GFX6-GFX9 • vgprs_used 0..256 • max(0, ceil(vgprs_used / 4) - 1) GFX90A, GFX940 • vgprs_used 0..512 • vgprs_used = align(arch_vgprs, 4) ◦ acc_vgprs • max(0, ceil(vgprs_used / 8) - 1) GFX10-GFX11 (wavefront size 64) • max_vgpr 1..256 • max(0, ceil(vgprs_used /

			4) - 1) GFX10-GFX11 (wavefront size 32) • max_vgpr 1..256 • max(0, ceil(vgprs_used / 8) - 1) Where vgprs_used is defined as the highest VGPR number explicitly referenced plus one. Used by CP to set up `COMPUTE_PGM_RSRC1.VGPRS`. The Assembler calculates this automatically for the selected processor from values provided to the *`.amdhsa_kernel`* directive by the *`.amdhsa_next_free_vgpr`* nested directive (see AMDHSA Kernel Assembler Directives).
9:6	4 bits	GRANULATED_WAVEFRONT_SGPR_COUNT	Number of scalar register blocks used by a wavefront; granularity is device specific: GFX6-GFX8 • sgprs_used 0..112 • max(0, ceil(sgprs_used / 8) - 1) GFX9 • sgprs_used 0..112 • 2 * max(0, ceil(sgprs_used / 16) - 1) GFX10-GFX11 Reserved, must be 0. (128 SGPRs always allocated.) Where sgprs_used is defined as the

			highest SGPR number explicitly referenced plus one, plus a target specific number of additional special SGPRs for VCC, FLAT_SCRATCH (GFX7+) and XNACK_MASK (GFX8+), and any additional target specific limitations. It does not include the 16 SGPRs added if a trap handler is enabled. The target specific limitations and special SGPR layout are defined in the hardware documentation, which can be found in the Processors table. Used by CP to set up `COMPUTE_PGM_RSRC1.SGPRS`. The Assembler calculates this automatically for the selected processor from values provided to the *`.amdhsa_kernel`* directive by the *`.amdhsa_next_free_sgpr`* and *`.amdhsa_reserve_*`* nested directives (see AMDHSA Kernel Assembler Directives).
11:10	2 bits	PRIORITY	Must be 0. Start executing wavefront at the specified priority. CP is responsible for filling in `COMPUTE_PGM_RSRC1.PRIORITY`.
13:12	2 bits	FLOAT_ROUND_MODE_32	Wavefront starts execution with specified rounding mode for single (32 bit) floating point precision floating point operations.

			Floating point rounding mode values are defined in Floating Point Rounding Mode Enumeration Values. Used by CP to set up `COMPUTE_PGM_RSRC1.FLOAT_MODE`.
15:14	2 bits	FLOAT_ROUND_MODE_16_64	Wavefront starts execution with specified rounding denorm mode for half/double (16 and 64-bit) floating point precision floating point operations. Floating point rounding mode values are defined in Floating Point Rounding Mode Enumeration Values. Used by CP to set up `COMPUTE_PGM_RSRC1.FLOAT_MODE`.
17:16	2 bits	FLOAT_DENORM_MODE_32	Wavefront starts execution with specified denorm mode for single (32 bit) floating point precision floating point operations. Floating point denorm mode values are defined in Floating Point Denorm Mode Enumeration Values. Used by CP to set up `COMPUTE_PGM_RSRC1.FLOAT_MODE`.
19:18	2 bits	FLOAT_DENORM_MODE_16_64	Wavefront starts execution with specified denorm mode for half/double (16 and 64-bit) floating point precision floating point operations. Floating point denorm mode values are defined in Floating Point

			Denorm Mode Enumeration Values. Used by CP to set up COMPUTE_PGM_RSRC1.FLOAT_MODE.
20	1 bit	PRIV	Must be 0. Start executing wavefront in privilege trap handler mode. CP is responsible for filling in COMPUTE_PGM_RSRC1.PRIV.
21	1 bit	ENABLE_DX10_CLAMP	Wavefront starts execution with DX10 clamp mode enabled. Used by the vector ALU to force DX10 style treatment of NaN's (when set, clamp NaN to zero, otherwise pass NaN through). Used by CP to set up COMPUTE_PGM_RSRC1.DX10_CLAMP.
22	1 bit	DEBUG_MODE	Must be 0. Start executing wavefront in single step mode. CP is responsible for filling in COMPUTE_PGM_RSRC1.DEBUG_MODE.
23	1 bit	ENABLE_IEEE_MODE	Wavefront starts execution with IEEE mode enabled. Floating point opcodes that support exception flag gathering will quiet and propagate signaling-NaN inputs per IEEE 754-2008. Min_dx10 and max_dx10 become IEEE 754-2008 compliant due to signaling-NaN propagation and quieting. Used by CP to set up COMPUTE_PGM_RSRC1.IEEE_MODE.

24	1 bit	BULKY	Must be 0. Only one work-group allowed to execute on a compute unit. CP is responsible for filling in `COMPUTE_PGM_RSRC1.BULKY`.
25	1 bit	CDBG_USER	Must be 0. Flag that can be used to control debugging code. CP is responsible for filling in `COMPUTE_PGM_RSRC1.CDBG_USER`.
26	1 bit	FP16_OVFL	GFX6-GFX8 Reserved, must be 0. GFX9-GFX11 Wavefront starts execution with specified fp16 overflow mode. • If 0, fp16 overflow generates +/-INF values. • If 1, fp16 overflow that is the result of an +/-INF input value or divide by 0 produces a +/-INF, otherwise clamps computed overflow to +/-MAX_FP16 as appropriate. Used by CP to set up `COMPUTE_PGM_RSRC1.FP16_OVFL`.
28:27	2 bits		Reserved, must be 0.
29	1 bit	WGP_MODE	GFX6-GFX9 Reserved, must be 0.

			GFX10-GFX11 • If 0 execute work-groups in CU wavefront execution mode. • If 1 execute work-groups on in WGP wavefront execution mode. See Memory Model. Used by CP to set up `COMPUTE_PGM_RSRC1.WGP_MODE`.
30	1 bit	MEM_ORDERED	GFX6-GFX9 Reserved, must be 0. GFX10-GFX11 Controls the behavior of the s_waitcnt's vmcnt and vscnt counters. • If 0 vmcnt reports completion of load and atomic with return out of order with sample instructions, and the vscnt reports the completion of store and atomic without return in order. • If 1 vmcnt reports completion of load, atomic with return and sample instructions in order, and the vscnt reports the completion of store and atomic without return in order. Used by CP to set up

			`COMPUTE_PGM_RSRC1.MEM_ORDERED`.
31	1 bit	FWD_PROGRESS	GFX6-GFX9 Reserved, must be 0. GFX10-GFX11 • If 0 execute SIMD wavefronts using oldest first policy. • If 1 execute SIMD wavefronts to ensure wavefronts will make some forward progress. Used by CP to set up `COMPUTE_PGM_RSRC1.FWD_PROGRESS`.
32	Total size 4 bytes		

compute_pgm_rsrc2 for GFX6-GFX11

Bits	Size	Field Name	Description
0	1 bit	ENABLE_PRIVATE_SEGMENT	• Enable the setup of the private segment. • If the *Target Properties* column of AMDGPU Processors does not specify *Architected flat scratch* then enable the setup of the SGPR wavefront scratch offset system register (see Initial Kernel Execution State). • If the *Target Properties* column of AMDGPU Processors specifies *Architected flat scratch* then enable the setup of the FLAT_SCRATCH register pair (see Initial Kernel

			Execution State). Used by CP to set up `COMPUTE_PGM_RSRC2.SCRATCH_EN`.
5:1	5 bits	USER_SGPR_COUNT	The total number of SGPR user data registers requested. This number must be greater than or equal to the number of user data registers enabled. Used by CP to set up `COMPUTE_PGM_RSRC2.USER_SGPR`.
6	1 bit	ENABLE_TRAP_HANDLER	Must be 0. This bit represents `COMPUTE_PGM_RSRC2.TRAP_PRESENT`, which is set by the CP if the runtime has installed a trap handler.
7	1 bit	ENABLE_SGPR_WORKGROUP_ID_X	Enable the setup of the system SGPR register for the work-group id in the X dimension (see Initial Kernel Execution State). Used by CP to set up `COMPUTE_PGM_RSRC2.TGID_X_EN`.
8	1 bit	ENABLE_SGPR_WORKGROUP_ID_Y	Enable the setup of the system SGPR register for the work-group id in the Y dimension (see Initial Kernel Execution State). Used by CP to set up `COMPUTE_PGM_RSRC2.TGID_Y_EN`.
9	1 bit	ENABLE_SGPR_WORKGROUP_ID_Z	Enable the setup of the system SGPR register for the work-group id in the Z dimension (see Initial Kernel Execution State).

			Used by CP to set up `COMPUTE_PGM_RSRC2.TGID_Z_EN`.
10	1 bit	ENABLE_SGPR_WORKGROUP_INFO	Enable the setup of the system SGPR register for work-group information (see Initial Kernel Execution State). Used by CP to set up `COMPUTE_PGM_RSRC2.TGID_SIZE_EN`.
12:11	2 bits	ENABLE_VGPR_WORKITEM_ID	Enable the setup of the VGPR system registers used for the work-item ID. System VGPR Work-Item ID Enumeration Values defines the values. Used by CP to set up `COMPUTE_PGM_RSRC2.TIDIG_CMP_CNT`.
13	1 bit	ENABLE_EXCEPTION_ADDRESS_WATCH	Must be 0. Wavefront starts execution with address watch exceptions enabled which are generated when L1 has witnessed a thread access an *address of interest*. CP is responsible for filling in the address watch bit in `COMPUTE_PGM_RSRC2.EXCP_EN_MSB` according to what the runtime requests.
14	1 bit	ENABLE_EXCEPTION_MEMORY	Must be 0. Wavefront starts execution with memory violation exceptions exceptions enabled which are generated when a memory violation has occurred for this wavefront from L1 or LDS (write-

			to-read-only-memory, mis-aligned atomic, LDS address out of range, illegal address, etc.). CP sets the memory violation bit in `COMPUTE_PGM_RSRC2.EXCP_EN_MSB` according to what the runtime requests.
23:15	9 bits	GRANULATED_LDS_SIZE	Must be 0. CP uses the rounded value from the dispatch packet, not this value, as the dispatch may contain dynamically allocated group segment memory. CP writes directly to `COMPUTE_PGM_RSRC2.LDS_SIZE`. Amount of group segment (LDS) to allocate for each work-group. Granularity is device specific: GFX6 roundup(lds-size / (64 * 4)) GFX7-GFX11 roundup(lds-size / (128 * 4))
24	1 bit	ENABLE_EXCEPTION_IEEE_754_FP _INVALID_OPERATION	Wavefront starts execution with specified exceptions enabled. Used by CP to set up `COMPUTE_PGM_RSRC2.EXCP_EN` (set from bits 0..6). IEEE 754 FP Invalid Operation
25	1 bit	ENABLE_EXCEPTION_FP_DENORMAL _SOURCE	FP Denormal one or more input operands is a denormal number
26	1 bit	ENABLE_EXCEPTION_IEEE_754_FP _DIVISION_BY_ZERO	IEEE 754 FP Division by Zero

27	1 bit	ENABLE_EXCEPTION_IEEE_754_FP_OVERFLOW	IEEE 754 FP FP Overflow
28	1 bit	ENABLE_EXCEPTION_IEEE_754_FP_UNDERFLOW	IEEE 754 FP Underflow
29	1 bit	ENABLE_EXCEPTION_IEEE_754_FP_INEXACT	IEEE 754 FP Inexact
30	1 bit	ENABLE_EXCEPTION_INT_DIVIDE_BY_ZERO	Integer Division by Zero (rcp_iflag_f32 instruction only)
31	1 bit		Reserved, must be 0.
32	Total size 4 bytes.		

compute_pgm_rsrc3 for GFX90A, GFX940

Bits	Size	Field Name	Description
5:0	6 bits	ACCUM_OFFSET	Offset of a first AccVGPR in the unified register file. Granularity 4. Value 0-63. 0 - accum-offset = 4, 1 - accum-offset = 8, ..., 63 - accum-offset = 256.
6:15	10 bits		Reserved, must be 0.
16	1 bit	TG_SPLIT	• If 0 the waves of a work-group are launched in the same CU. • If 1 the waves of a work-group can be launched in different CUs. The waves cannot use S_BARRIER or LDS.
17:31	15 bits		Reserved, must be 0.
32	Total size 4 bytes.		

compute_pgm_rsrc3 for GFX10-GFX11

Bits	Size	Field Name	Description
3:0	4 bits	SHARED_VGPR_COUNT	Number of shared VGPR blocks when executing in subvector mode. For wavefront size 64 the value is 0-15, representing 0-120 VGPRs (granularity of 8), such that (compute_pgm_rsrc1.vgprs +1)*4 +

			shared_vgpr_count*8 does not exceed 256. For wavefront size 32 shared_vgpr_count must be 0.
9:4	6 bits	INST_PREF_SIZE	GFX10 Reserved, must be 0. GFX11 Number of instruction bytes to prefetch, starting at the kernel's entry point instruction, before wavefront starts execution. The value is 0..63 with a granularity of 128 bytes.
10	1 bit	TRAP_ON_START	GFX10 Reserved, must be 0. GFX11 Must be 0. If 1, wavefront starts execution by trapping into the trap handler. CP is responsible for filling in the trap on start bit in `COMPUTE_PGM_RSRC3.TRAP_ON_START` according to what the runtime requests.
11	1 bit	TRAP_ON_END	GFX10 Reserved, must be 0. GFX11 Must be 0. If 1, wavefront execution terminates by trapping into the trap handler. CP is responsible for filling in the trap on end bit in `COMPUTE_PGM_RSRC3.TRAP_ON_END` according to what the runtime requests.
30:12	19 bits		Reserved, must be 0.
31	1	IMAGE_OP	GFX10

bit	Reserved, must be 0.
GFX11	If 1, the kernel execution contains image instructions. If executed as part of a graphics pipeline, image read instructions will stall waiting for any necessary `WAIT_SYNC` fence to be performed in order to indicate that earlier pipeline stages have completed writing to the image. Not used for compute kernels that are not part of a graphics pipeline and must be 0.

32	Total size 4 bytes.

Floating Point Rounding Mode Enumeration Values

Enumeration Name	Value	Description
FLOAT_ROUND_MODE_NEAR_EVEN	0	Round Ties To Even
FLOAT_ROUND_MODE_PLUS_INFINITY	1	Round Toward +infinity
FLOAT_ROUND_MODE_MINUS_INFINITY	2	Round Toward -infinity
FLOAT_ROUND_MODE_ZERO	3	Round Toward 0

Floating Point Denorm Mode Enumeration Values

Enumeration Name	Value	Description
FLOAT_DENORM_MODE_FLUSH_SRC_DST	0	Flush Source and Destination Denorms
FLOAT_DENORM_MODE_FLUSH_DST	1	Flush Output Denorms
FLOAT_DENORM_MODE_FLUSH_SRC	2	Flush Source Denorms
FLOAT_DENORM_MODE_FLUSH_NONE	3	No Flush

Denormal flushing is sign respecting. i.e. the behavior expected by `"denormal-fp-math"="preserve-sign"`. The behavior is undefined with `"denormal-fp-math"="positive-zero"`

System VGPR Work-Item ID Enumeration Values

Enumeration Name	Value	Description
SYSTEM_VGPR_WORKITEM_ID_X	0	Set work-item X dimension ID.
SYSTEM_VGPR_WORKITEM_ID_X_Y	1	Set work-item X and Y

		dimensions ID.
SYSTEM_VGPR_WORKITEM_ID_X_Y_Z	2	Set work-item X, Y and Z dimensions ID.
SYSTEM_VGPR_WORKITEM_ID_UNDEFINED	3	Undefined.

Initial Kernel Execution State

This section defines the register state that will be set up by the packet processor prior to the start of execution of every wavefront. This is limited by the constraints of the hardware controllers of CP/ADC/SPI.

The order of the SGPR registers is defined, but the compiler can specify which ones are actually setup in the kernel descriptor using the `enable_sgpr_*` bit fields (see Kernel Descriptor). The register numbers used for enabled registers are dense starting at SGPR0: the first enabled register is SGPR0, the next enabled register is SGPR1 etc.; disabled registers do not have an SGPR number.

The initial SGPRs comprise up to 16 User SRGPs that are set by CP and apply to all wavefronts of the grid. It is possible to specify more than 16 User SGPRs using the `enable_sgpr_*` bit fields, in which case only the first 16 are actually initialized. These are then immediately followed by the System SGPRs that are set up by ADC/SPI and can have different values for each wavefront of the grid dispatch.

SGPR register initial state is defined in SGPR Register Set Up Order.

SGPR Register Set Up Order

SGPR Order	Name (kernel descriptor enable field)	Number of SGPRs	Description
First	Private Segment Buffer (enable_sgpr_private_segment_buffer)	4	See Private Segment Buffer.
then	Dispatch Ptr (enable_sgpr_dispatch_ptr)	2	64-bit address of AQL dispatch packet for kernel dispatch actually executing.
then	Queue Ptr (enable_sgpr_queue_ptr)	2	64-bit address of amd_queue_t object for AQL queue on which the dispatch packet was queued.
then	Kernarg Segment Ptr (enable_sgpr_kernarg_segment_ptr)	2	64-bit address of Kernarg segment. This is directly copied from the kernarg_address in the kernel dispatch

			packet. Having CP load it once avoids loading it at the beginning of every wavefront.
then	Dispatch Id (enable_sgpr_dispatch_id)	2	64-bit Dispatch ID of the dispatch packet being executed.
then	Flat Scratch Init (enable_sgpr_flat_scratch_init)	2	See Flat Scratch.
then	Private Segment Size (enable_sgpr_private_segment_size)	1	The 32-bit byte size of a single work-item's memory allocation. This is the value from the kernel dispatch packet Private Segment Byte Size rounded up by CP to a multiple of DWORD. Having CP load it once avoids loading it at the beginning of every wavefront. This is not used for GFX7-GFX8 since it is the same value as the second SGPR of Flat Scratch Init. However, it may be needed for GFX9-GFX11 which changes the meaning of the Flat Scratch Init value.
then	Work-Group Id X (enable_sgpr_workgroup_id_X)	1	32-bit work-group id in X dimension of grid for wavefront.
then	Work-Group Id Y (enable_sgpr_workgroup_id_Y)	1	32-bit work-group id in Y dimension of grid for wavefront.
then	Work-Group Id Z (enable_sgpr_workgroup_id_Z)	1	32-bit work-group id in Z dimension of grid for wavefront.
then	Work-Group Info (enable_sgpr_workgroup_info)	1	{first_wavefront, 14'b0000, ordered_append_term[10:0], threadgroup_size_in_wavefronts[5:0]}
then	Scratch Wavefront Offset (enable_sgpr_private	1	See Flat Scratch. and Private Segment Buffer.

	_segment_wavefront_offset)

The order of the VGPR registers is defined, but the compiler can specify which ones are actually setup in the kernel descriptor using the `enable_vgpr*` bit fields (see Kernel Descriptor). The register numbers used for enabled registers are dense starting at VGPR0: the first enabled register is VGPR0, the next enabled register is VGPR1 etc.; disabled registers do not have a VGPR number.

There are different methods used for the VGPR initial state:

- Unless the *Target Properties* column of AMDGPU Processors specifies otherwise, a separate VGPR register is used per work-item ID. The VGPR register initial state for this method is defined in VGPR Register Set Up Order for Unpacked Work-Item ID Method.
- If *Target Properties* column of AMDGPU Processors specifies *Packed work-item IDs*, the initial value of VGPR0 register is used for all work-item IDs. The register layout for this method is defined in Register Layout for Packed Work-Item ID Method.

VGPR Register Set Up Order for Unpacked Work-Item ID Method

VGPR Order	Name (kernel descriptor enable field)	Number of VGPRs	Description
First	Work-Item Id X (Always initialized)	1	32-bit work-item id in X dimension of work-group for wavefront lane.
then	Work-Item Id Y (enable_vgpr_workitem_id > 0)	1	32-bit work-item id in Y dimension of work-group for wavefront lane.
then	Work-Item Id Z (enable_vgpr_workitem_id > 1)	1	32-bit work-item id in Z dimension of work-group for wavefront lane.

Register Layout for Packed Work-Item ID Method

Bits	Size	Field Name	Description
0:9	10 bits	Work-Item Id X	Work-item id in X dimension of work-group for wavefront lane. Always initialized.
10:19	10 bits	Work-Item Id Y	Work-item id in Y dimension of work-group for wavefront lane. Initialized if enable_vgpr_workitem_id > 0, otherwise set to 0.
20:29	10 bits	Work-Item Id Z	Work-item id in Z dimension of work-group for

		wavefront lane. Initialized if enable_vgpr_workitem_id > 1, otherwise set to 0.
30:31	2 bits	Reserved, set to 0.

The setting of registers is done by GPU CP/ADC/SPI hardware as follows:

1. SGPRs before the Work-Group Ids are set by CP using the 16 User Data registers.
2. Work-group Id registers X, Y, Z are set by ADC which supports any combination including none.
3. Scratch Wavefront Offset is set by SPI in a per wavefront basis which is why its value cannot be included with the flat scratch init value which is per queue (see Flat Scratch).
4. The VGPRs are set by SPI which only supports specifying either (X), (X, Y) or (X, Y, Z).
5. Flat Scratch register pair initialization is described in Flat Scratch.

The global segment can be accessed either using buffer instructions (GFX6 which has V# 64-bit address support), flat instructions (GFX7-GFX11), or global instructions (GFX9-GFX11).

If buffer operations are used, then the compiler can generate a V# with the following properties:

- base address of 0
- no swizzle
- ATC: 1 if IOMMU present (such as APU)
- ptr64: 1
- MTYPE set to support memory coherence that matches the runtime (such as CC for APU and NC for dGPU).

Kernel Prolog

The compiler performs initialization in the kernel prologue depending on the target and information about things like stack usage in the kernel and called functions. Some of this initialization requires the compiler to request certain User and System SGPRs be present in the Initial Kernel Execution State via the Kernel Descriptor.

CFI

1. The CFI return address is undefined.
2. The CFI CFA is defined using an expression which evaluates to a location description that comprises one memory location description for the `DW_ASPACE_AMDGPU_private_lane` address

space address `0`.

M0

GFX6-GFX8
: The M0 register must be initialized with a value at least the total LDS size if the kernel may access LDS via DS or flat operations. Total LDS size is available in dispatch packet. For M0, it is also possible to use maximum possible value of LDS for given target (0x7FFF for GFX6 and 0xFFFF for GFX7-GFX8).

GFX9-GFX11
: The M0 register is not used for range checking LDS accesses and so does not need to be initialized in the prolog.

Stack Pointer

If the kernel has function calls it must set up the ABI stack pointer described in Non-Kernel Functions by setting SGPR32 to the unswizzled scratch offset of the address past the last local allocation.

Frame Pointer

If the kernel needs a frame pointer for the reasons defined in `SIFrameLowering` then SGPR33 is used and is always set to `0` in the kernel prolog. If a frame pointer is not required then all uses of the frame pointer are replaced with immediate `0` offsets.

Flat Scratch

There are different methods used for initializing flat scratch:

- If the *Target Properties* column of AMDGPU Processors specifies *Does not support generic address space*:

  Flat scratch is not supported and there is no flat scratch register pair.

- If the *Target Properties* column of AMDGPU Processors specifies *Offset flat scratch*:

  If the kernel or any function it calls may use flat operations to access scratch memory, the prolog code must set up the FLAT_SCRATCH register pair (FLAT_SCRATCH_LO/FLAT_SCRATCH_HI). Initialization uses Flat Scratch Init and Scratch Wavefront Offset SGPR registers (see Initial Kernel Execution State):

1. The low word of Flat Scratch Init is the 32-bit byte offset from `SH_HIDDEN_PRIVATE_BASE_VIMID` to the base of scratch backing memory being managed by SPI for the queue executing the kernel dispatch. This is the same value used in the Scratch Segment Buffer V# base address.

   CP obtains this from the runtime. (The Scratch Segment Buffer base address is `SH_HIDDEN_PRIVATE_BASE_VIMID` plus this offset.)

   The prolog must add the value of Scratch Wavefront Offset to get the wavefront's byte scratch backing memory offset from `SH_HIDDEN_PRIVATE_BASE_VIMID`.

   The Scratch Wavefront Offset must also be used as an offset with Private segment address when using the Scratch Segment Buffer.

   Since FLAT_SCRATCH_LO is in units of 256 bytes, the offset must be right shifted by 8 before moving into FLAT_SCRATCH_HI.

   FLAT_SCRATCH_HI corresponds to SGPRn-4 on GFX7, and SGPRn-6 on GFX8 (where SGPRn is the highest numbered SGPR allocated to the wavefront). FLAT_SCRATCH_HI is multiplied by 256 (as it is in units of 256 bytes) and added to `SH_HIDDEN_PRIVATE_BASE_VIMID` to calculate the per wavefront FLAT SCRATCH BASE in flat memory instructions that access the scratch aperture.

2. The second word of Flat Scratch Init is 32-bit byte size of a single work-items scratch memory usage.

   CP obtains this from the runtime, and it is always a multiple of DWORD. CP checks that the value in the kernel dispatch packet Private Segment Byte Size is not larger and requests the runtime to increase the queue's scratch size if necessary.

   CP directly loads from the kernel dispatch packet Private Segment Byte Size field and rounds up to a multiple of DWORD. Having CP load it once avoids loading it at the beginning of every wavefront.

   The kernel prolog code must move it to FLAT_SCRATCH_LO which is SGPRn-3 on GFX7 and SGPRn-5 on GFX8. FLAT_SCRATCH_LO is used as the FLAT SCRATCH SIZE in flat memory instructions.

- If the *Target Properties* column of AMDGPU Processors specifies *Absolute flat scratch*:

  If the kernel or any function it calls may use flat operations to access scratch memory, the prolog code must set up the FLAT_SCRATCH register pair (FLAT_SCRATCH_LO/FLAT_SCRATCH_HI which are in SGPRn-4/SGPRn-3). Initialization uses

Flat Scratch Init and Scratch Wavefront Offset SGPR registers (see Initial Kernel Execution State):

The Flat Scratch Init is the 64-bit address of the base of scratch backing memory being managed by SPI for the queue executing the kernel dispatch.

CP obtains this from the runtime.

The kernel prolog must add the value of the wave's Scratch Wavefront Offset and move the result as a 64-bit value to the FLAT_SCRATCH SGPR register pair which is SGPRn-6 and SGPRn-5. It is used as the FLAT SCRATCH BASE in flat memory instructions.

The Scratch Wavefront Offset must also be used as an offset with Private segment address when using the Scratch Segment Buffer (see Private Segment Buffer).

- If the *Target Properties* column of AMDGPU Processors specifies *Architected flat scratch*:

  If ENABLE_PRIVATE_SEGMENT is enabled in compute_pgm_rsrc2 for GFX6-GFX11 then the FLAT_SCRATCH register pair will be initialized to the 64-bit address of the base of scratch backing memory being managed by SPI for the queue executing the kernel dispatch plus the value of the wave's Scratch Wavefront Offset for use as the flat scratch base in flat memory instructions.

Private Segment Buffer

If the *Target Properties* column of AMDGPU Processors specifies *Architected flat scratch* then a Private Segment Buffer is not supported. Instead the flat SCRATCH instructions are used.

Otherwise, Private Segment Buffer SGPR register is used to initialize 4 SGPRs that are used as a V# to access scratch. CP uses the value provided by the runtime. It is used, together with Scratch Wavefront Offset as an offset, to access the private memory space using a segment address. See Initial Kernel Execution State.

The scratch V# is a four-aligned SGPR and always selected for the kernel as follows:

- If it is known during instruction selection that there is stack usage, SGPR0-3 is reserved for use as the scratch V#. Stack usage is assumed if optimizations are disabled (`-O0`), if stack objects already exist (for locals, etc.), or if there are any function calls.
- Otherwise, four high numbered SGPRs beginning at a four-aligned SGPR index are reserved for the tentative scratch V#. These will be used if it is determined that spilling is needed.

- If no use is made of the tentative scratch V#, then it is unreserved, and the register count is determined ignoring it.
- If use is made of the tentative scratch V#, then its register numbers are shifted to the first four-aligned SGPR index after the highest one allocated by the register allocator, and all uses are updated. The register count includes them in the shifted location.
- In either case, if the processor has the SGPR allocation bug, the tentative allocation is not shifted or unreserved in order to ensure the register count is higher to workaround the bug.

> **Note**
>
> This approach of using a tentative scratch V# and shifting the register numbers if used avoids having to perform register allocation a second time if the tentative V# is eliminated. This is more efficient and avoids the problem that the second register allocation may perform spilling which will fail as there is no longer a scratch V#.

When the kernel prolog code is being emitted it is known whether the scratch V# described above is actually used. If it is, the prolog code must set it up by copying the Private Segment Buffer to the scratch V# registers and then adding the Private Segment Wavefront Offset to the queue base address in the V#. The result is a V# with a base address pointing to the beginning of the wavefront scratch backing memory.

The Private Segment Buffer is always requested, but the Private Segment Wavefront Offset is only requested if it is used (see Initial Kernel Execution State).

Memory Model

This section describes the mapping of the LLVM memory model onto AMDGPU machine code (see Memory Model for Concurrent Operations).

The AMDGPU backend supports the memory synchronization scopes specified in Memory Scopes.

The code sequences used to implement the memory model specify the order of instructions that a single thread must execute. The `s_waitcnt` and cache management instructions such as `buffer_wbinvl1_vol` are defined with respect to other memory instructions executed by the same thread. This allows them to be moved earlier or later which can allow them to be combined with other instances of the same instruction, or hoisted/sunk out of loops to improve performance. Only the instructions related to the memory model are given; additional `s_waitcnt` instructions are required to ensure registers are defined before being used. These may be able to be combined with the memory model `s_waitcnt` instructions as described above.

The AMDGPU backend supports the following memory models:

HSA Memory Model [HSA]

The HSA memory model uses a single happens-before relation for all address spaces (see Address Spaces).

OpenCL Memory Model [OpenCL]

The OpenCL memory model which has separate happens-before relations for the global and local address spaces. Only a fence specifying both global and local address space, and seq_cst instructions join the relationships. Since the LLVM `memfence` instruction does not allow an address space to be specified the OpenCL fence has to conservatively assume both local and global address space was specified. However, optimizations can often be done to eliminate the additional `s_waitcnt` instructions when there are no intervening memory instructions which access the corresponding address space. The code sequences in the table indicate what can be omitted for the OpenCL memory. The target triple environment is used to determine if the source language is OpenCL (see OpenCL).

`ds/flat_load/store/atomic` instructions to local memory are termed LDS operations.

`buffer/global/flat_load/store/atomic` instructions to global memory are termed vector memory operations.

Private address space uses `buffer_load/store` using the scratch V# (GFX6-GFX8), or `scratch_load/store` (GFX9-GFX11). Since only a single thread is accessing the memory, atomic memory orderings are not meaningful, and all accesses are treated as non-atomic.

Constant address space uses `buffer/global_load` instructions (or equivalent scalar memory instructions). Since the constant address space contents do not change during the execution of a kernel dispatch it is not legal to perform stores, and atomic memory orderings are not meaningful, and all accesses are treated as non-atomic.

A memory synchronization scope wider than work-group is not meaningful for the group (LDS) address space and is treated as work-group.

The memory model does not support the region address space which is treated as non-atomic.

Acquire memory ordering is not meaningful on store atomic instructions and is treated as non-atomic.

Release memory ordering is not meaningful on load atomic instructions and is treated a non-

atomic.

Acquire-release memory ordering is not meaningful on load or store atomic instructions and is treated as acquire and release respectively.

The memory order also adds the single thread optimization constraints defined in table AMDHSA Memory Model Single Thread Optimization Constraints.

AMDHSA Memory Model Single Thread Optimization Constraints

LLVM Memory Ordering	Optimization Constraints
unordered	*none*
monotonic	*none*
acquire	• If a load atomic/atomicrmw then no following load/load atomic/store/store atomic/atomicrmw/fence instruction can be moved before the acquire. • If a fence then same as load atomic, plus no preceding associated fence-paired-atomic can be moved after the fence.
release	• If a store atomic/atomicrmw then no preceding load/load atomic/store/store atomic/atomicrmw/fence instruction can be moved after the release. • If a fence then same as store atomic, plus no following associated fence-paired-atomic can be moved before the fence.
acq_rel	Same constraints as both acquire and release.
seq_cst	• If a load atomic then same constraints as acquire, plus no preceding sequentially consistent load atomic/store atomic/atomicrmw/fence instruction can be moved after the seq_cst. • If a store atomic then the same constraints as release, plus no following sequentially consistent load atomic/store atomic/atomicrmw/fence instruction can be moved before the seq_cst. • If an atomicrmw/fence then same constraints as acq_rel.

The code sequences used to implement the memory model are defined in the following sections:

Memory Model GFX6-GFX9

For GFX6-GFX9:

- Each agent has multiple shader arrays (SA).
- Each SA has multiple compute units (CU).
- Each CU has multiple SIMDs that execute wavefronts.
- The wavefronts for a single work-group are executed in the same CU but may be executed by different SIMDs.
- Each CU has a single LDS memory shared by the wavefronts of the work-groups executing on it.
- All LDS operations of a CU are performed as wavefront wide operations in a global order and involve no caching. Completion is reported to a wavefront in execution order.
- The LDS memory has multiple request queues shared by the SIMDs of a CU. Therefore, the LDS operations performed by different wavefronts of a work-group can be reordered relative to each other, which can result in reordering the visibility of vector memory operations with respect to LDS operations of other wavefronts in the same work-group. A `s_waitcnt lgkmcnt(0)` is required to ensure synchronization between LDS operations and vector memory operations between wavefronts of a work-group, but not between operations performed by the same wavefront.
- The vector memory operations are performed as wavefront wide operations and completion is reported to a wavefront in execution order. The exception is that for GFX7-GFX9 `flat_load/store/atomic` instructions can report out of vector memory order if they access LDS memory, and out of LDS operation order if they access global memory.
- The vector memory operations access a single vector L1 cache shared by all SIMDs a CU. Therefore, no special action is required for coherence between the lanes of a single wavefront, or for coherence between wavefronts in the same work-group. A `buffer_wbinvl1_vol` is required for coherence between wavefronts executing in different work-groups as they may be executing on different CUs.
- The scalar memory operations access a scalar L1 cache shared by all wavefronts on a group of CUs. The scalar and vector L1 caches are not coherent. However, scalar operations are used in a restricted way so do not impact the memory model. See Memory Spaces.
- The vector and scalar memory operations use an L2 cache shared by all CUs on the same agent.

- The L2 cache has independent channels to service disjoint ranges of virtual addresses.
- Each CU has a separate request queue per channel. Therefore, the vector and scalar memory operations performed by wavefronts executing in different work-groups (which may be executing on different CUs) of an agent can be reordered relative to each other. A `s_waitcnt vmcnt(0)` is required to ensure synchronization between vector memory operations of different CUs. It ensures a previous vector memory operation has completed before executing a subsequent vector memory or LDS operation and so can be used to meet the requirements of acquire and release.
- The L2 cache can be kept coherent with other agents on some targets, or ranges of virtual addresses can be set up to bypass it to ensure system coherence.

Scalar memory operations are only used to access memory that is proven to not change during the execution of the kernel dispatch. This includes constant address space and global address space for program scope `const` variables. Therefore, the kernel machine code does not have to maintain the scalar cache to ensure it is coherent with the vector caches. The scalar and vector caches are invalidated between kernel dispatches by CP since constant address space data may change between kernel dispatch executions. See Memory Spaces.

The one exception is if scalar writes are used to spill SGPR registers. In this case the AMDGPU backend ensures the memory location used to spill is never accessed by vector memory operations at the same time. If scalar writes are used then a `s_dcache_wb` is inserted before the `s_endpgm` and before a function return since the locations may be used for vector memory instructions by a future wavefront that uses the same scratch area, or a function call that creates a frame at the same address, respectively. There is no need for a `s_dcache_inv` as all scalar writes are write-before-read in the same thread.

For kernarg backing memory:

- CP invalidates the L1 cache at the start of each kernel dispatch.
- On dGPU the kernarg backing memory is allocated in host memory accessed as MTYPE UC (uncached) to avoid needing to invalidate the L2 cache. This also causes it to be treated as non-volatile and so is not invalidated by `*_vol`.
- On APU the kernarg backing memory it is accessed as MTYPE CC (cache coherent) and so the L2 cache will be coherent with the CPU and other agents.

Scratch backing memory (which is used for the private address space) is accessed with MTYPE NC_NV (non-coherent non-volatile). Since the private address space is only accessed by a single thread, and is always write-before-read, there is never a need to invalidate these entries from the L1 cache. Hence all cache invalidates are done as `*_vol` to only invalidate the volatile cache lines.

The code sequences used to implement the memory model for GFX6-GFX9 are defined in table AMDHSA Memory Model Code Sequences GFX6-GFX9.

AMDHSA Memory Model Code Sequences GFX6-GFX9

LLVM Instr	LLVM Memory Ordering	LLVM Memory Sync Scope	AMDGPU Address Space	AMDGPU Machine Code GFX6-GFX9
Non-Atomic				
load	*none*	*none*	• global • generic • private • constant	• !volatile & !nontemporal 1. buffer/global/flat_load • !volatile & nontemporal 1. buffer/global/flat_load glc=1 slc=1 • volatile 1. buffer/global/flat_load glc=1 2. s_waitcnt vmcnt(0) ◦ Must happen before any following volatile global/generic load/store. ◦ Ensures that volatile operations to different addresses will not be reordered by hardware.
load	*none*	*none*	• local	1. ds_load
store	*none*	*none*	• global • generic • private	• !volatile & !nontemporal 1. buffer/global/flat_store

			• constant	!volatile & nontemporal 1. buffer/global/flat_store glc=1 slc=1 • volatile 1. buffer/global/flat_store 2. s_waitcnt vmcnt(0) ◦ Must happen before any following volatile global/generic load/store. ◦ Ensures that volatile operations to different addresses will not be reordered by hardware.
store	*none*	*none*	• local	1. ds_store
Unordered Atomic				
load atomic	unordered	*any*	*any*	*Same as non-atomic*.
store atomic	unordered	*any*	*any*	*Same as non-atomic*.
atomicrmw	unordered	*any*	*any*	*Same as monotonic atomic*.
Monotonic Atomic				
load atomic	monotonic	• singlethread • wavefront • workgroup	• global • local • generic	1. buffer/global/ds/flat_load
load atomic	monotonic	• agent • system	• global • generic	1. buffer/global/flat_load glc=1

store atomic	monotonic	• singlethread • wavefront • workgroup • agent • system	• global • generic	1. buffer/global/flat_store	
store atomic	monotonic	• singlethread • wavefront • workgroup	• local	1. ds_store	
atomicrmw	monotonic	• singlethread • wavefront • workgroup • agent • system	• global • generic	1. buffer/global/flat_atomic	
atomicrmw	monotonic	• singlethread • wavefront • workgroup	• local	1. ds_atomic	
Acquire Atomic					
load atomic	acquire	• singlethread • wavefront	• global • local • generic	1. buffer/global/ds/flat_load	
load atomic	acquire	• workgroup	• global	1. buffer/global_load	
load atomic	acquire	• workgroup	• local • generic	1. ds/flat_load 2. s_waitcnt lgkmcnt(0) • If OpenCL, omit. • Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures any following global	

				data read is no older than a local load atomic value being acquired.
load atomic	acquire	• agent • system	• global	1. buffer/global_load glc=1 2. s_waitcnt vmcnt(0) • Must happen before following buffer_wbinvl1_vol. • Ensures the load has completed before invalidating the cache. 3. buffer_wbinvl1_vol • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale global data.
load atomic	acquire	• agent • system	• generic	1. flat_load glc=1 2. s_waitcnt vmcnt(0) & lgkmcnt(0) • If OpenCL omit lgkmcnt(0). • Must happen before following buffer_wbinvl1_vol. • Ensures the flat_load has completed before invalidating the

				cache. 3. buffer_wbinvl1_vol • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale global data.
atomicrmw	acquire	• singlethread • wavefront	• global • local • generic	1. buffer/global/ds/flat_atomic
atomicrmw	acquire	• workgroup	• global	1. buffer/global_atomic
atomicrmw	acquire	• workgroup	• local • generic	1. ds/flat_atomic 2. s_waitcnt lgkmcnt(0) • If OpenCL, omit. • Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures any following global data read is no older than a local atomicrmw value being acquired.
atomicrmw	acquire	• agent • system	• global	1. buffer/global_atomic 2. s_waitcnt vmcnt(0)

				• Must happen before following buffer_wbinvl1_vol. • Ensures the atomicrmw has completed before invalidating the cache. 3. buffer_wbinvl1_vol • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale global data.
atomicrmw	acquire	• agent • system	• generic	1. flat_atomic 2. s_waitcnt vmcnt(0) & lgkmcnt(0) • If OpenCL, omit lgkmcnt(0). • Must happen before following buffer_wbinvl1_vol. • Ensures the atomicrmw has completed before invalidating the cache. 3. buffer_wbinvl1_vol • Must happen before any following global/generic

				load/load atomic/atomicrmw. • Ensures that following loads will not see stale global data.
fence	acquire	• singlethread • wavefront	*none*	*none*
fence	acquire	• workgroup	*none*	1. s_waitcnt lgkmcnt(0) • If OpenCL and address space is not generic, omit. • However, since LLVM currently has no address space on the fence need to conservatively always generate. If fence had an address space then set to address space of OpenCL fence flag, or to generic if both local and global flags are specified. • Must happen after any preceding local/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-

				paired-atomic). • Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures any following global data read is no older than the value read by the fence-paired-atomic.
fence	acquire	• agent • system	*none*	1. s_waitcnt lgkmcnt(0) & vmcnt(0) • If OpenCL and address space is not generic, omit lgkmcnt(0). • However, since LLVM currently has no address space on the fence need to conservatively always generate (see comment for previous fence). • Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0)

must happen after any preceding global/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic).

- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic).
- Must happen before the following buffer_wbinvl1_vol.
- Ensures that the fence-paired atomic has completed before invalidating the cache. Therefore any following locations read must be no older than the value read by the fence-paired-atomic.

				2. buffer_wbinvl1_vol • Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures that following loads will not see stale global data.
Release Atomic				
store atomic	release	• singlethread • wavefront	• global • local • generic	1. buffer/global/ds/flat_store
store atomic	release	• workgroup	• global • generic	1. s_waitcnt lgkmcnt(0) • If OpenCL, omit. • Must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw. • Must happen before the following store. • Ensures that all memory operations to local have completed before performing the store that is being released. 2. buffer/global/flat_store

store atomic	release	• workgroup	• local	1. ds_store
store atomic	release	• agent • system	• global • generic	1. s_waitcnt lgkmcnt(0) & vmcnt(0) • If OpenCL and address space is not generic, omit lgkmcnt(0). • Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw. • s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw. • Must happen before the following store. • Ensures that all memory operations to memory have completed before

				performing the store that is being released. 2. buffer/global/flat_store
atomicrmw	release	• singlethread • wavefront	• global • local • generic	1. buffer/global/ds/flat_atomic
atomicrmw	release	• workgroup	• global • generic	1. s_waitcnt lgkmcnt(0) • If OpenCL, omit. • Must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw. • Must happen before the following atomicrmw. • Ensures that all memory operations to local have completed before performing the atomicrmw that is being released. 2. buffer/global/flat_atomic
atomicrmw	release	• workgroup	• local	1. ds_atomic
atomicrmw	release	• agent • system	• global • generic	1. s_waitcnt lgkmcnt(0) & vmcnt(0) • If OpenCL, omit lgkmcnt(0). • Could be split into separate s_waitcnt

				vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw. • s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw. • Must happen before the following atomicrmw. • Ensures that all memory operations to global and local have completed before performing the atomicrmw that is being released. 2. buffer/global/flat_atomic
fence	release	• singlethread • wavefront	*none*	*none*
fence	release	• workgroup	*none*	1. s_waitcnt lgkmcnt(0)

- If OpenCL and address space is not generic, omit.
- However, since LLVM currently has no address space on the fence need to conservatively always generate. If fence had an address space then set to address space of OpenCL fence flag, or to generic if both local and global flags are specified.
- Must happen after any preceding local/generic load/load atomic/store/store atomic/atomicrmw.
- Must happen before any following store atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic).
- Ensures that all memory operations to local have completed before

				performing the following fence-paired-atomic.
fence	release	• agent • system	*none*	1. s_waitcnt lgkmcnt(0) & vmcnt(0) • If OpenCL and address space is not generic, omit lgkmcnt(0). • If OpenCL and address space is local, omit vmcnt(0). • However, since LLVM currently has no address space on the fence need to conservatively always generate. If fence had an address space then set to address space of OpenCL fence flag, or to generic if both local and global flags are specified. • Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after

				any preceding global/generic load/store/load atomic/store atomic/atomicrmw. • s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw. • Must happen before any following store atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic). • Ensures that all memory operations have completed before performing the following fence-paired-atomic.
Acquire-Release Atomic				
atomicrmw	acq_rel	• singlethread • wavefront	• global • local • generic	1. buffer/global/ds/flat_atomic
atomicrmw	acq_rel	• workgroup	• global	1. s_waitcnt lgkmcnt(0) • If OpenCL, omit. • Must happen after

				any preceding local/generic load/store/load atomic/store atomic/atomicrmw. • Must happen before the following atomicrmw. • Ensures that all memory operations to local have completed before performing the atomicrmw that is being released. 2. buffer/global_atomic
atomicrmw	acq_rel	• workgroup	• local	1. ds_atomic 2. s_waitcnt lgkmcnt(0) • If OpenCL, omit. • Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures any following global data read is no older than the local load atomic value being acquired.
atomicrmw	acq_rel	• workgroup	• generic	1. s_waitcnt lgkmcnt(0) • If OpenCL, omit. • Must happen after

				any preceding local/generic load/store/load atomic/store atomic/atomicrmw. • Must happen before the following atomicrmw. • Ensures that all memory operations to local have completed before performing the atomicrmw that is being released. 2. flat_atomic 3. s_waitcnt lgkmcnt(0) • If OpenCL, omit. • Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures any following global data read is no older than a local load atomic value being acquired.
atomicrmw	acq_rel	• agent • system	• global	1. s_waitcnt lgkmcnt(0) & vmcnt(0) • If OpenCL, omit lgkmcnt(0). • Could be split into separate s_waitcnt

vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.

- s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before the following atomicrmw.
- Ensures that all memory operations to global have completed before performing the atomicrmw that is being released.

2. buffer/global_atomic
3. s_waitcnt vmcnt(0)

    - Must happen before following buffer_wbinvl1_vol.

				• Ensures the atomicrmw has completed before invalidating the cache. 4. buffer_wbinvl1_vol • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale global data.
atomicrmw	acq_rel	• agent • system	• generic	1. s_waitcnt lgkmcnt(0) & vmcnt(0) • If OpenCL, omit lgkmcnt(0). • Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw. • s_waitcnt

lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before the following atomicrmw.
- Ensures that all memory operations to global have completed before performing the atomicrmw that is being released.

2. flat_atomic
3. s_waitcnt vmcnt(0) & lgkmcnt(0)

    - If OpenCL, omit lgkmcnt(0).
    - Must happen before following buffer_wbinvl1_vol.
    - Ensures the atomicrmw has completed before invalidating the cache.

4. buffer_wbinvl1_vol

    - Must happen before any following global/generic load/load atomic/atomicrmw.

				Ensures that following loads will not see stale global data.
fence	acq_rel	• singlethread • wavefront	*none*	*none*
fence	acq_rel	• workgroup	*none*	1. s_waitcnt lgkmcnt(0) • If OpenCL and address space is not generic, omit. • However, since LLVM currently has no address space on the fence need to conservatively always generate (see comment for previous fence). • Must happen after any preceding local/generic load/load atomic/store/store atomic/atomicrmw. • Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures that all memory operations to local have completed before performing any following global

memory operations.
- Ensures that the preceding local/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the acquire-fence-paired-atomic) has completed before following global memory operations. This satisfies the requirements of acquire.
- Ensures that all previous memory operations have completed before a following local/generic store atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the release-fence-paired-atomic). This satisfies the requirements of release.

fence	acq_rel	• agent • system	*none*	1. s_waitcnt lgkmcnt(0) & vmcnt(0) • If OpenCL and address space is not generic, omit lgkmcnt(0). • However, since LLVM currently has no address space on the fence need to conservatively always generate (see comment for previous fence). • Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw. • s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.

Must happen before the following buffer_wbinvl1_vol.

- Ensures that the preceding global/local/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the acquire-fence-paired-atomic) has completed before invalidating the cache. This satisfies the requirements of acquire.
- Ensures that all previous memory operations have completed before a following global/local/generic store atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the release-fence-paired-atomic). This satisfies the

				requirements of release. 2. buffer_wbinvl1_vol • Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures that following loads will not see stale global data. This satisfies the requirements of acquire.
Sequential Consistent Atomic				
load atomic	seq_cst	• singlethread • wavefront	• global • local • generic	*Same as corresponding load atomic acquire, except must generate all instructions even for OpenCL.*
load atomic	seq_cst	• workgroup	• global • generic	1. s_waitcnt lgkmcnt(0) • Must happen after preceding local/generic load atomic/store atomic/atomicrmw with memory ordering of seq_cst and with equal or wider sync scope. (Note that seq_cst fences have their own s_waitcnt lgkmcnt(0) and so do not need to be

considered.)

- Ensures any preceding sequential consistent local memory instructions have completed before executing this sequentially consistent instruction. This prevents reordering a seq_cst store followed by a seq_cst load. (Note that seq_cst is stronger than acquire/release as the reordering of load acquire followed by a store release is prevented by the s_waitcnt of the release, but there is nothing preventing a store release followed by load acquire from completing out of order. The s_waitcnt could be placed after seq_store or before the seq_load. We choose the load to make the s_waitcnt be as late as possible so that

				the store may have already completed.) 2. *Following instructions same as corresponding load atomic acquire, except must generate all instructions even for OpenCL.*
load atomic	seq_cst	• workgroup	• local	*Same as corresponding load atomic acquire, except must generate all instructions even for OpenCL.*
load atomic	seq_cst	• agent • system	• global • generic	1. s_waitcnt lgkmcnt(0) & vmcnt(0) • Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt lgkmcnt(0) must happen after preceding global/generic load atomic/store atomic/atomicrmw with memory ordering of seq_cst and with equal or wider sync scope. (Note that seq_cst fences have their own s_waitcnt lgkmcnt(0) and so do not need to be

considered.)

- s_waitcnt vmcnt(0) must happen after preceding global/generic load atomic/store atomic/atomicrmw with memory ordering of seq_cst and with equal or wider sync scope. (Note that seq_cst fences have their own s_waitcnt vmcnt(0) and so do not need to be considered.)
- Ensures any preceding sequential consistent global memory instructions have completed before executing this sequentially consistent instruction. This prevents reordering a seq_cst store followed by a seq_cst load. (Note that seq_cst is stronger than acquire/release as the reordering of load acquire followed by a store

				release is prevented by the s_waitcnt of the release, but there is nothing preventing a store release followed by load acquire from completing out of order. The s_waitcnt could be placed after seq_store or before the seq_load. We choose the load to make the s_waitcnt be as late as possible so that the store may have already completed.) 2. *Following instructions same as corresponding load atomic acquire, except must generate all instructions even for OpenCL.*
store atomic	seq_cst	• singlethread • wavefront • workgroup • agent • system	• global • local • generic	*Same as corresponding store atomic release, except must generate all instructions even for OpenCL.*
atomicrmw	seq_cst	• singlethread • wavefront • workgroup • agent • system	• global • local • generic	*Same as corresponding atomicrmw acq_rel, except must generate all instructions even for OpenCL.*
fence	seq_cst	• singlethread • wavefront	*none*	*Same as corresponding fence acq_rel, except must generate all instructions*

	• workgroup • agent • system	*even for OpenCL.*

Memory Model GFX90A

For GFX90A:

- Each agent has multiple shader arrays (SA).
- Each SA has multiple compute units (CU).
- Each CU has multiple SIMDs that execute wavefronts.
- The wavefronts for a single work-group are executed in the same CU but may be executed by different SIMDs. The exception is when in tgsplit execution mode when the wavefronts may be executed by different SIMDs in different CUs.
- Each CU has a single LDS memory shared by the wavefronts of the work-groups executing on it. The exception is when in tgsplit execution mode when no LDS is allocated as wavefronts of the same work-group can be in different CUs.
- All LDS operations of a CU are performed as wavefront wide operations in a global order and involve no caching. Completion is reported to a wavefront in execution order.
- The LDS memory has multiple request queues shared by the SIMDs of a CU. Therefore, the LDS operations performed by different wavefronts of a work-group can be reordered relative to each other, which can result in reordering the visibility of vector memory operations with respect to LDS operations of other wavefronts in the same work-group. A `s_waitcnt lgkmcnt(0)` is required to ensure synchronization between LDS operations and vector memory operations between wavefronts of a work-group, but not between operations performed by the same wavefront.
- The vector memory operations are performed as wavefront wide operations and completion is reported to a wavefront in execution order. The exception is that `flat_load/store/atomic` instructions can report out of vector memory order if they access LDS memory, and out of LDS operation order if they access global memory.
- The vector memory operations access a single vector L1 cache shared by all SIMDs a CU. Therefore:
  - No special action is required for coherence between the lanes of a single wavefront.
  - No special action is required for coherence between wavefronts in the same work-group since they execute on the same CU. The exception is when in tgsplit execution mode as wavefronts of the same work-group can be in different CUs and so a `buffer_wbinvl1_vol` is required as described in the following item.
  - A `buffer_wbinvl1_vol` is required for coherence between wavefronts executing in

different work-groups as they may be executing on different CUs.

- The scalar memory operations access a scalar L1 cache shared by all wavefronts on a group of CUs. The scalar and vector L1 caches are not coherent. However, scalar operations are used in a restricted way so do not impact the memory model. See Memory Spaces.
- The vector and scalar memory operations use an L2 cache shared by all CUs on the same agent.
  - The L2 cache has independent channels to service disjoint ranges of virtual addresses.
  - Each CU has a separate request queue per channel. Therefore, the vector and scalar memory operations performed by wavefronts executing in different work-groups (which may be executing on different CUs), or the same work-group if executing in tgsplit mode, of an agent can be reordered relative to each other. A `s_waitcnt vmcnt(0)` is required to ensure synchronization between vector memory operations of different CUs. It ensures a previous vector memory operation has completed before executing a subsequent vector memory or LDS operation and so can be used to meet the requirements of acquire and release.
  - The L2 cache of one agent can be kept coherent with other agents by: using the MTYPE RW (read-write) or MTYPE CC (cache-coherent) with the PTE C-bit for memory local to the L2; and using the MTYPE NC (non-coherent) with the PTE C-bit set or MTYPE UC (uncached) for memory not local to the L2.
    - Any local memory cache lines will be automatically invalidated by writes from CUs associated with other L2 caches, or writes from the CPU, due to the cache probe caused by coherent requests. Coherent requests are caused by GPU accesses to pages with the PTE C-bit set, by CPU accesses over XGMI, and by PCIe requests that are configured to be coherent requests.
    - XGMI accesses from the CPU to local memory may be cached on the CPU. Subsequent access from the GPU will automatically invalidate or writeback the CPU cache due to the L2 probe filter and and the PTE C-bit being set.
    - Since all work-groups on the same agent share the same L2, no L2 invalidation or writeback is required for coherence.
    - To ensure coherence of local and remote memory writes of work-groups in different agents a `buffer_wbl2` is required. It will writeback dirty L2 cache lines of MTYPE RW (used for local coarse grain memory) and MTYPE NC ()used for remote coarse grain memory). Note that MTYPE CC (used for local fine grain memory) causes write through to DRAM, and MTYPE UC (used for remote fine grain memory) bypasses the L2, so both will never result in dirty L2 cache lines.
    - To ensure coherence of local and remote memory reads of work-groups in different agents a `buffer_invl2` is required. It will invalidate L2 cache lines with MTYPE NC (used for remote coarse grain memory). Note that MTYPE CC (used for

local fine grain memory) and MTYPE RW (used for local coarse memory) cause local reads to be invalidated by remote writes with with the PTE C-bit so these cache lines are not invalidated. Note that MTYPE UC (used for remote fine grain memory) bypasses the L2, so will never result in L2 cache lines that need to be invalidated.

- PCIe access from the GPU to the CPU memory is kept coherent by using the MTYPE UC (uncached) which bypasses the L2.

Scalar memory operations are only used to access memory that is proven to not change during the execution of the kernel dispatch. This includes constant address space and global address space for program scope `const` variables. Therefore, the kernel machine code does not have to maintain the scalar cache to ensure it is coherent with the vector caches. The scalar and vector caches are invalidated between kernel dispatches by CP since constant address space data may change between kernel dispatch executions. See Memory Spaces.

The one exception is if scalar writes are used to spill SGPR registers. In this case the AMDGPU backend ensures the memory location used to spill is never accessed by vector memory operations at the same time. If scalar writes are used then a `s_dcache_wb` is inserted before the `s_endpgm` and before a function return since the locations may be used for vector memory instructions by a future wavefront that uses the same scratch area, or a function call that creates a frame at the same address, respectively. There is no need for a `s_dcache_inv` as all scalar writes are write-before-read in the same thread.

For kernarg backing memory:

- CP invalidates the L1 cache at the start of each kernel dispatch.
- On dGPU over XGMI or PCIe the kernarg backing memory is allocated in host memory accessed as MTYPE UC (uncached) to avoid needing to invalidate the L2 cache. This also causes it to be treated as non-volatile and so is not invalidated by `*_vol`.
- On APU the kernarg backing memory is accessed as MTYPE CC (cache coherent) and so the L2 cache will be coherent with the CPU and other agents.

Scratch backing memory (which is used for the private address space) is accessed with MTYPE NC_NV (non-coherent non-volatile). Since the private address space is only accessed by a single thread, and is always write-before-read, there is never a need to invalidate these entries from the L1 cache. Hence all cache invalidates are done as `*_vol` to only invalidate the volatile cache lines.

The code sequences used to implement the memory model for GFX90A are defined in table AMDHSA Memory Model Code Sequences GFX90A.

AMDHSA Memory Model Code Sequences GFX90A

LLVM Instr	LLVM Memory Ordering	LLVM Memory Sync Scope	AMDGPU Address Space	AMDGPU Machine Code GFX90A
Non-Atomic				
load	*none*	*none*	• global • generic • private • constant	• !volatile & !nontemporal 1. buffer/global/flat_load • !volatile & nontemporal 1. buffer/global/flat_load glc=1 slc=1 • volatile 1. buffer/global/flat_load glc=1 2. s_waitcnt vmcnt(0) ◦ Must happen before any following volatile global/generic load/store. ◦ Ensures that volatile operations to different addresses will not be reordered by hardware.
load	*none*	*none*	• local	1. ds_load
store	*none*	*none*	• global • generic • private • constant	• !volatile & !nontemporal 1. buffer/global/flat_store • !volatile & nontemporal 1. buffer/global/flat_store glc=1 slc=1 • volatile

				1. buffer/global/flat_store 2. s_waitcnt vmcnt(0) ◦ Must happen before any following volatile global/generic load/store. ◦ Ensures that volatile operations to different addresses will not be reordered by hardware.
store	*none*	*none*	• local	1. ds_store
Unordered Atomic				
load atomic	unordered	*any*	*any*	*Same as non-atomic*.
store atomic	unordered	*any*	*any*	*Same as non-atomic*.
atomicrmw	unordered	*any*	*any*	*Same as monotonic atomic*.
Monotonic Atomic				
load atomic	monotonic	• singlethread • wavefront	• global • generic	1. buffer/global/flat_load
load atomic	monotonic	• workgroup	• global • generic	1. buffer/global/flat_load glc=1 • If not TgSplit execution mode, omit glc=1.
load atomic	monotonic	• singlethread • wavefront	• local	*If TgSplit execution mode, local address space cannot be used.*

		workgroup		1. ds_load
load atomic	monotonic	• agent	• global • generic	1. buffer/global/flat_load glc=1
load atomic	monotonic	• system	• global • generic	1. buffer/global/flat_load glc=1
store atomic	monotonic	• singlethread • wavefront • workgroup • agent	• global • generic	1. buffer/global/flat_store
store atomic	monotonic	• system	• global • generic	1. buffer/global/flat_store
store atomic	monotonic	• singlethread • wavefront • workgroup	• local	*If TgSplit execution mode, local address space cannot be used.* 1. ds_store
atomicrmw	monotonic	• singlethread • wavefront • workgroup • agent	• global • generic	1. buffer/global/flat_atomic
atomicrmw	monotonic	• system	• global • generic	1. buffer/global/flat_atomic
atomicrmw	monotonic	• singlethread • wavefront • workgroup	• local	*If TgSplit execution mode, local address space cannot be used.* 1. ds_atomic
Acquire Atomic				
load atomic	acquire	• singlethread • wavefront	• global • local • generic	1. buffer/global/ds/flat_load
load	acquire	• workgroup	• global	1. buffer/global_load glc=1

atomic				• If not TgSplit execution mode, omit glc=1. 2. s_waitcnt vmcnt(0) • If not TgSplit execution mode, omit. • Must happen before the following buffer_wbinvl1_vol. 3. buffer_wbinvl1_vol • If not TgSplit execution mode, omit. • Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures that following loads will not see stale data.
load atomic	acquire	• workgroup	• local	*If TgSplit execution mode, local address space cannot be used.* 1. ds_load 2. s_waitcnt lgkmcnt(0) • If OpenCL, omit. • Must happen before any following global/generic load/load

				atomic/store/store atomic/atomicrmw. • Ensures any following global data read is no older than the local load atomic value being acquired.
load atomic	acquire	• workgroup	• generic	1. flat_load glc=1 • If not TgSplit execution mode, omit glc=1. 2. s_waitcnt lgkm/vmcnt(0) • Use lgkmcnt(0) if not TgSplit execution mode and vmcnt(0) if TgSplit execution mode. • If OpenCL, omit lgkmcnt(0). • Must happen before the following buffer_wbinvl1_vol and any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures any following global data read is no older than a local load atomic value being acquired.

				3. buffer_wbinvl1_vol • If not TgSplit execution mode, omit. • Ensures that following loads will not see stale data.
load atomic	acquire	• agent	• global	1. buffer/global_load glc=1 2. s_waitcnt vmcnt(0) • Must happen before following buffer_wbinvl1_vol. • Ensures the load has completed before invalidating the cache. 3. buffer_wbinvl1_vol • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale global data.
load atomic	acquire	• system	• global	1. buffer/global/flat_load glc=1 2. s_waitcnt vmcnt(0) • Must happen before following buffer_invl2 and buffer_wbinvl1_vol. • Ensures the load

				has completed before invalidating the cache. 3. buffer_invl2; buffer_wbinvl1_vol • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale L1 global data, nor see stale L2 MTYPE NC global data. MTYPE RW and CC memory will never be stale in L2 due to the memory probes.
load atomic	acquire	• agent	• generic	1. flat_load glc=1 2. s_waitcnt vmcnt(0) & lgkmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL omit lgkmcnt(0). • Must happen before following buffer_wbinvl1_vol. • Ensures the flat_load has completed before invalidating the cache.

				3. buffer_wbinvl1_vol • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale global data.
load atomic	acquire	• system	• generic	1. flat_load glc=1 2. s_waitcnt vmcnt(0) & lgkmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL omit lgkmcnt(0). • Must happen before following buffer_invl2 and buffer_wbinvl1_vol. • Ensures the flat_load has completed before invalidating the caches. 3. buffer_invl2; buffer_wbinvl1_vol • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will

				not see stale L1 global data, nor see stale L2 MTYPE NC global data. MTYPE RW and CC memory will never be stale in L2 due to the memory probes.
atomicrmw	acquire	• singlethread • wavefront	• global • generic	1. buffer/global/flat_atomic
atomicrmw	acquire	• singlethread • wavefront	• local	*If TgSplit execution mode, local address space cannot be used.* 1. ds_atomic
atomicrmw	acquire	• workgroup	• global	1. buffer/global_atomic 2. s_waitcnt vmcnt(0) • If not TgSplit execution mode, omit. • Must happen before the following buffer_wbinvl1_vol. • Ensures the atomicrmw has completed before invalidating the cache. 3. buffer_wbinvl1_vol • If not TgSplit execution mode, omit. • Must happen before any following global/generic

				load/load atomic/atomicrmw. • Ensures that following loads will not see stale global data.	
atomicrmw	acquire	• workgroup	• local	*If TgSplit execution mode, local address space cannot be used.* 1. ds_atomic 2. s_waitcnt lgkmcnt(0) • If OpenCL, omit. • Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures any following global data read is no older than the local atomicrmw value being acquired.	
atomicrmw	acquire	• workgroup	• generic	1. flat_atomic 2. s_waitcnt lgkm/vmcnt(0) • Use lgkmcnt(0) if not TgSplit execution mode and vmcnt(0) if TgSplit execution mode. • If OpenCL, omit lgkmcnt(0). • Must happen before the following	

				buffer_wbinvl1_vol and any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures any following global data read is no older than a local atomicrmw value being acquired. 3. buffer_wbinvl1_vol • If not TgSplit execution mode, omit. • Ensures that following loads will not see stale data.
atomicrmw	acquire	• agent	• global	1. buffer/global_atomic 2. s_waitcnt vmcnt(0) • Must happen before following buffer_wbinvl1_vol. • Ensures the atomicrmw has completed before invalidating the cache. 3. buffer_wbinvl1_vol • Must happen before any following global/generic load/load

				atomic/atomicrmw. • Ensures that following loads will not see stale global data.
atomicrmw	acquire	• system	• global	1. buffer/global_atomic 2. s_waitcnt vmcnt(0) • Must happen before following buffer_invl2 and buffer_wbinvl1_vol. • Ensures the atomicrmw has completed before invalidating the caches. 3. buffer_invl2; buffer_wbinvl1_vol • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale L1 global data, nor see stale L2 MTYPE NC global data. MTYPE RW and CC memory will never be stale in L2 due to the memory probes.
atomicrmw	acquire	• agent	• generic	1. flat_atomic 2. s_waitcnt vmcnt(0) & lgkmcnt(0)

				• If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL, omit lgkmcnt(0). • Must happen before following buffer_wbinvl1_vol. • Ensures the atomicrmw has completed before invalidating the cache. 3. buffer_wbinvl1_vol • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale global data.
atomicrmw	acquire	• system	• generic	1. flat_atomic 2. s_waitcnt vmcnt(0) & lgkmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL, omit lgkmcnt(0). • Must happen before following buffer_invl2 and buffer_wbinvl1_vol. • Ensures the

				atomicrmw has completed before invalidating the caches. 3. buffer_invl2; buffer_wbinvl1_vol • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale L1 global data, nor see stale L2 MTYPE NC global data. MTYPE RW and CC memory will never be stale in L2 due to the memory probes.
fence	acquire	• singlethread • wavefront	*none*	*none*
fence	acquire	• workgroup	*none*	1. s_waitcnt lgkm/vmcnt(0) • Use lgkmcnt(0) if not TgSplit execution mode and vmcnt(0) if TgSplit execution mode. • If OpenCL and address space is not generic, omit lgkmcnt(0). • If OpenCL and address space is

local, omit vmcnt(0).

- However, since LLVM currently has no address space on the fence need to conservatively always generate. If fence had an address space then set to address space of OpenCL fence flag, or to generic if both local and global flags are specified.
- s_waitcnt vmcnt(0) must happen after any preceding global/generic load atomic/ atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic).
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this

				is termed the fence-paired-atomic). • Must happen before the following buffer_wbinvl1_vol and any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures any following global data read is no older than the value read by the fence-paired-atomic. 2. buffer_wbinvl1_vol • If not TgSplit execution mode, omit. • Ensures that following loads will not see stale data.
fence	acquire	• agent	*none*	1. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL and address space is not generic, omit lgkmcnt(0). • However, since LLVM currently has no address space on the fence need to

conservatively always generate (see comment for previous fence).
- Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.
- s_waitcnt vmcnt(0) must happen after any preceding global/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic).
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic).

				• Must happen before the following buffer_wbinvl1_vol. • Ensures that the fence-paired atomic has completed before invalidating the cache. Therefore any following locations read must be no older than the value read by the fence-paired-atomic. 2. buffer_wbinvl1_vol • Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures that following loads will not see stale global data.
fence	acquire	• system	*none*	1. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL and address space is not generic, omit lgkmcnt(0). • However, since LLVM currently has

no address space on the fence need to conservatively always generate (see comment for previous fence).
- Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.
- s_waitcnt vmcnt(0) must happen after any preceding global/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic).
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this

				is termed the fence-paired-atomic). • Must happen before the following buffer_invl2 and buffer_wbinvl1_vol. • Ensures that the fence-paired atomic has completed before invalidating the cache. Therefore any following locations read must be no older than the value read by the fence-paired-atomic. 2. buffer_invl2; buffer_wbinvl1_vol • Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures that following loads will not see stale L1 global data, nor see stale L2 MTYPE NC global data. MTYPE RW and CC memory will never be stale in L2 due to the memory probes.
Release Atomic				
store	release	• singlethread	• global	1. buffer/global/flat_store

atomic		• wavefront	• generic	
store atomic	release	• singlethread • wavefront	• local	*If TgSplit execution mode, local address space cannot be used.* 1. ds_store
store atomic	release	• workgroup	• global • generic	1. s_waitcnt lgkm/vmcnt(0) • Use lgkmcnt(0) if not TgSplit execution mode and vmcnt(0) if TgSplit execution mode. • If OpenCL, omit lgkmcnt(0). • s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/ load atomic/store atomic/ atomicrmw. • s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw. • Must happen before the following store. • Ensures that all memory operations have completed before performing the store that is being released.

				2. buffer/global/flat_store
store atomic	release	• workgroup	• local	*If TgSplit execution mode, local address space cannot be used.* 1. ds_store
store atomic	release	• agent	• global • generic	1. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL and address space is not generic, omit lgkmcnt(0). • Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw. • s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store

				atomic/atomicrmw. • Must happen before the following store. • Ensures that all memory operations to memory have completed before performing the store that is being released. 2. buffer/global/flat_store
store atomic	release	• system	• global • generic	1. buffer_wbl2 • Must happen before following s_waitcnt. • Performs L2 writeback to ensure previous global/generic store/atomicrmw are visible at system scope. 2. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL and address space is not generic, omit lgkmcnt(0). • Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow

				them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw. • s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw. • Must happen before the following store. • Ensures that all memory operations to memory and the L2 writeback have completed before performing the store that is being released. 3. buffer/global/flat_store	
atomicrmw	release	• singlethread • wavefront	• global • generic	1. buffer/global/flat_atomic	
atomicrmw	release	• singlethread • wavefront	• local	*If TgSplit execution mode, local address space cannot be used.* 1. ds_atomic	

atomicrmw	release	• workgroup	• global • generic	1. s_waitcnt lgkm/vmcnt(0) • Use lgkmcnt(0) if not TgSplit execution mode and vmcnt(0) if TgSplit execution mode. • If OpenCL, omit lgkmcnt(0). • s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/ load atomic/store atomic/ atomicrmw. • s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw. • Must happen before the following atomicrmw. • Ensures that all memory operations have completed before performing the atomicrmw that is being released. 2. buffer/global/flat_atomic
atomicrmw	release	• workgroup	• local	*If TgSplit execution mode, local address space cannot be used.*

				1. ds_atomic
atomicrmw	release	• agent	• global • generic	1. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL, omit lgkmcnt(0). • Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw. • s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw. • Must happen before the following atomicrmw. • Ensures that all memory operations

				to global and local have completed before performing the atomicrmw that is being released. 2. buffer/global/flat_atomic
atomicrmw	release	• system	• global • generic	1. buffer_wbl2 • Must happen before following s_waitcnt. • Performs L2 writeback to ensure previous global/generic store/atomicrmw are visible at system scope. 2. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL, omit lgkmcnt(0). • Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after any preceding

				global/generic load/store/load atomic/store atomic/atomicrmw. • s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw. • Must happen before the following atomicrmw. • Ensures that all memory operations to memory and the L2 writeback have completed before performing the store that is being released. 3. buffer/global/flat_atomic
fence	release	• singlethread • wavefront	*none*	*none*
fence	release	• workgroup	*none*	1. s_waitcnt lgkm/vmcnt(0) • Use lgkmcnt(0) if not TgSplit execution mode and vmcnt(0) if TgSplit execution mode. • If OpenCL and address space is not generic, omit

lgkmcnt(0).

- If OpenCL and address space is local, omit vmcnt(0).
- However, since LLVM currently has no address space on the fence need to conservatively always generate. If fence had an address space then set to address space of OpenCL fence flag, or to generic if both local and global flags are specified.
- s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/ load atomic/store atomic/ atomicrmw.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/load atomic/store/store atomic/atomicrmw.
- Must happen before any following store atomic/atomicrmw with an equal or wider sync scope

				and memory ordering stronger than unordered (this is termed the fence-paired-atomic). • Ensures that all memory operations have completed before performing the following fence-paired-atomic.
fence	release	• agent	*none*	1. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL and address space is not generic, omit lgkmcnt(0). • If OpenCL and address space is local, omit vmcnt(0). • However, since LLVM currently has no address space on the fence need to conservatively always generate. If fence had an address space then set to address space of OpenCL fence flag, or to generic if both local and global flags are specified.

Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.

- s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before any following store atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic).
- Ensures that all memory operations have completed before performing

				the following fence-paired-atomic.
fence	release	• system	*none*	1. buffer_wbl2 • If OpenCL and address space is local, omit. • Must happen before following s_waitcnt. • Performs L2 writeback to ensure previous global/generic store/atomicrmw are visible at system scope. 2. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL and address space is not generic, omit lgkmcnt(0). • If OpenCL and address space is local, omit vmcnt(0). • However, since LLVM currently has no address space on the fence need to conservatively always generate. If fence had an address space then set to address space

of OpenCL fence flag, or to generic if both local and global flags are specified.

- Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.
- s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before any following store atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-

				paired-atomic). • Ensures that all memory operations have completed before performing the following fence-paired-atomic.	
Acquire-Release Atomic					
atomicrmw	acq_rel	• singlethread • wavefront	• global • generic	1. buffer/global/flat_atomic	
atomicrmw	acq_rel	• singlethread • wavefront	• local	*If TgSplit execution mode, local address space cannot be used.* 1. ds_atomic	
atomicrmw	acq_rel	• workgroup	• global	1. s_waitcnt lgkm/vmcnt(0) • Use lgkmcnt(0) if not TgSplit execution mode and vmcnt(0) if TgSplit execution mode. • If OpenCL, omit lgkmcnt(0). • Must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw. • s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/ load atomic/store atomic/ atomicrmw.	

- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before the following atomicrmw.
- Ensures that all memory operations have completed before performing the atomicrmw that is being released.

2. buffer/global_atomic
3. s_waitcnt vmcnt(0)

    - If not TgSplit execution mode, omit.
    - Must happen before the following buffer_wbinvl1_vol.
    - Ensures any following global data read is no older than the atomicrmw value being acquired.

4. buffer_wbinvl1_vol

    - If not TgSplit execution mode, omit.

				Ensures that following loads will not see stale data.
atomicrmw	acq_rel	• workgroup	• local	*If TgSplit execution mode, local address space cannot be used.* 1. ds_atomic 2. s_waitcnt lgkmcnt(0) • If OpenCL, omit. • Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures any following global data read is no older than the local load atomic value being acquired.
atomicrmw	acq_rel	• workgroup	• generic	1. s_waitcnt lgkm/vmcnt(0) • Use lgkmcnt(0) if not TgSplit execution mode and vmcnt(0) if TgSplit execution mode. • If OpenCL, omit lgkmcnt(0). • s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/ load atomic/store

atomic/ atomicrmw.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before the following atomicrmw.
- Ensures that all memory operations have completed before performing the atomicrmw that is being released.

2. flat_atomic
3. s_waitcnt lgkmcnt(0) & vmcnt(0)

    - If not TgSplit execution mode, omit vmcnt(0).
    - If OpenCL, omit lgkmcnt(0).
    - Must happen before the following buffer_wbinvl1_vol and any following global/generic load/load atomic/store/store atomic/atomicrmw.
    - Ensures any following global data read is no older than a local

				load atomic value being acquired. 3. buffer_wbinvl1_vol • If not TgSplit execution mode, omit. • Ensures that following loads will not see stale data.
atomicrmw	acq_rel	• agent	• global	1. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL, omit lgkmcnt(0). • Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw. • s_waitcnt lgkmcnt(0) must happen after any preceding

local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before the following atomicrmw.
- Ensures that all memory operations to global have completed before performing the atomicrmw that is being released.

2. buffer/global_atomic
3. s_waitcnt vmcnt(0)

    - Must happen before following buffer_wbinvl1_vol.
    - Ensures the atomicrmw has completed before invalidating the cache.

4. buffer_wbinvl1_vol

    - Must happen before any following global/generic load/load atomic/atomicrmw.
    - Ensures that following loads will not see stale global data.

	atomicrmw	acq_rel	system	global	1. buffer_wbl2 • Must happen before following s_waitcnt. • Performs L2 writeback to ensure previous global/generic store/atomicrmw are visible at system scope. 2. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL, omit lgkmcnt(0). • Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw. • s_waitcnt lgkmcnt(0) must happen after any preceding

local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before the following atomicrmw.
- Ensures that all memory operations to global and L2 writeback have completed before performing the atomicrmw that is being released.

3. buffer/global_atomic
4. s_waitcnt vmcnt(0)

    - Must happen before following buffer_invl2 and buffer_wbinvl1_vol.
    - Ensures the atomicrmw has completed before invalidating the caches.

5. buffer_invl2; buffer_wbinvl1_vol

    - Must happen before any following global/generic load/load atomic/atomicrmw.
    - Ensures that following loads will not see stale L1

				global data, nor see stale L2 MTYPE NC global data. MTYPE RW and CC memory will never be stale in L2 due to the memory probes.
atomicrmw	acq_rel	• agent	• generic	1. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL, omit lgkmcnt(0). • Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw. • s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store

atomic/atomicrmw.
- Must happen before the following atomicrmw.
- Ensures that all memory operations to global have completed before performing the atomicrmw that is being released.

2. flat_atomic
3. s_waitcnt vmcnt(0) & lgkmcnt(0)

    - If TgSplit execution mode, omit lgkmcnt(0).
    - If OpenCL, omit lgkmcnt(0).
    - Must happen before following buffer_wbinvl1_vol.
    - Ensures the atomicrmw has completed before invalidating the cache.

4. buffer_wbinvl1_vol

    - Must happen before any following global/generic load/load atomic/atomicrmw.
    - Ensures that following loads will not see stale global

				data.
atomicrmw	acq_rel	• system	• generic	1. buffer_wbl2 • Must happen before following s_waitcnt. • Performs L2 writeback to ensure previous global/generic store/atomicrmw are visible at system scope. 2. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL, omit lgkmcnt(0). • Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw. • s_waitcnt lgkmcnt(0) must

happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before the following atomicrmw.
- Ensures that all memory operations to global and L2 writeback have completed before performing the atomicrmw that is being released.

3. flat_atomic
4. s_waitcnt vmcnt(0) & lgkmcnt(0)

    - If TgSplit execution mode, omit lgkmcnt(0).
    - If OpenCL, omit lgkmcnt(0).
    - Must happen before following buffer_invl2 and buffer_wbinvl1_vol.
    - Ensures the atomicrmw has completed before invalidating the caches.

5. buffer_invl2; buffer_wbinvl1_vol

    - Must happen before

				any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale L1 global data, nor see stale L2 MTYPE NC global data. MTYPE RW and CC memory will never be stale in L2 due to the memory probes.
fence	acq_rel	• singlethread • wavefront	*none*	*none*
fence	acq_rel	• workgroup	*none*	1. s_waitcnt lgkm/vmcnt(0) • Use lgkmcnt(0) if not TgSplit execution mode and vmcnt(0) if TgSplit execution mode. • If OpenCL and address space is not generic, omit lgkmcnt(0). • If OpenCL and address space is local, omit vmcnt(0). • However, since LLVM currently has no address space on the fence need to conservatively always generate (see comment for

previous fence).

- s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/ load atomic/store atomic/ atomicrmw.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/load atomic/store/store atomic/atomicrmw.
- Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw.
- Ensures that all memory operations have completed before performing any following global memory operations.
- Ensures that the preceding local/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the

acquire-fence-paired-atomic) has completed before following global memory operations. This satisfies the requirements of acquire.

- Ensures that all previous memory operations have completed before a following local/generic store atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the release-fence-paired-atomic). This satisfies the requirements of release.
- Must happen before the following buffer_wbinvl1_vol.
- Ensures that the acquire-fence-paired atomic has completed before invalidating the cache. Therefore any following locations read must be no older than the

				value read by the acquire-fence-paired-atomic. 2. buffer_wbinvl1_vol • If not TgSplit execution mode, omit. • Ensures that following loads will not see stale data.
fence	acq_rel	• agent	*none*	1. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL and address space is not generic, omit lgkmcnt(0). • However, since LLVM currently has no address space on the fence need to conservatively always generate (see comment for previous fence). • Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.

- s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before the following buffer_wbinvl1_vol.
- Ensures that the preceding global/local/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the acquire-fence-paired-atomic) has completed before invalidating the cache. This satisfies the requirements of acquire.
- Ensures that all previous memory

				operations have completed before a following global/local/generic store atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the release-fence-paired-atomic). This satisfies the requirements of release. 2. buffer_wbinvl1_vol • Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures that following loads will not see stale global data. This satisfies the requirements of acquire.
fence	acq_rel	• system	*none*	1. buffer_wbl2 • If OpenCL and address space is local, omit. • Must happen before

following s_waitcnt.
- Performs L2 writeback to ensure previous global/generic store/atomicrmw are visible at system scope.

2. s_waitcnt lgkmcnt(0) & vmcnt(0)

    - If TgSplit execution mode, omit lgkmcnt(0).
    - If OpenCL and address space is not generic, omit lgkmcnt(0).
    - However, since LLVM currently has no address space on the fence need to conservatively always generate (see comment for previous fence).
    - Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.
    - s_waitcnt vmcnt(0) must happen after any preceding global/generic

load/store/load atomic/store atomic/atomicrmw.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before the following buffer_invl2 and buffer_wbinvl1_vol.
- Ensures that the preceding global/local/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the acquire-fence-paired-atomic) has completed before invalidating the cache. This satisfies the requirements of acquire.
- Ensures that all previous memory operations have completed before a following

global/local/generic store atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the release-fence-paired-atomic). This satisfies the requirements of release.

3. buffer_invl2; buffer_wbinvl1_vol
   - Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw.
   - Ensures that following loads will not see stale L1 global data, nor see stale L2 MTYPE NC global data. MTYPE RW and CC memory will never be stale in L2 due to the memory probes.

Sequential Consistent Atomic				
load atomic	seq_cst	• singlethread • wavefront	• global • local • generic	*Same as corresponding load atomic acquire, except must generate all instructions even for OpenCL.*

load atomic	seq_cst	• workgroup	• global • generic	1. s_waitcnt lgkm/vmcnt(0) • Use lgkmcnt(0) if not TgSplit execution mode and vmcnt(0) if TgSplit execution mode. • s_waitcnt lgkmcnt(0) must happen after preceding local/generic load atomic/store atomic/atomicrmw with memory ordering of seq_cst and with equal or wider sync scope. (Note that seq_cst fences have their own s_waitcnt lgkmcnt(0) and so do not need to be considered.) • s_waitcnt vmcnt(0) must happen after preceding global/generic load atomic/store atomic/atomicrmw with memory ordering of seq_cst and with equal or wider sync scope. (Note that seq_cst fences have their own s_waitcnt vmcnt(0) and so do

not need to be considered.)

- Ensures any preceding sequential consistent global/local memory instructions have completed before executing this sequentially consistent instruction. This prevents reordering a seq_cst store followed by a seq_cst load. (Note that seq_cst is stronger than acquire/release as the reordering of load acquire followed by a store release is prevented by the s_waitcnt of the release, but there is nothing preventing a store release followed by load acquire from completing out of order. The s_waitcnt could be placed after seq_store or before the seq_load. We choose the load to make the

				s_waitcnt be as late as possible so that the store may have already completed.) 2. *Following instructions same as corresponding load atomic acquire, except must generate all instructions even for OpenCL.*
load atomic	seq_cst	• workgroup	• local	*If TgSplit execution mode, local address space cannot be used.* *Same as corresponding load atomic acquire, except must generate all instructions even for OpenCL.*
load atomic	seq_cst	• agent • system	• global • generic	1. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt lgkmcnt(0) must happen after preceding global/generic load atomic/store atomic/atomicrmw with memory

ordering of seq_cst and with equal or wider sync scope. (Note that seq_cst fences have their own s_waitcnt lgkmcnt(0) and so do not need to be considered.)

- s_waitcnt vmcnt(0) must happen after preceding global/generic load atomic/store atomic/atomicrmw with memory ordering of seq_cst and with equal or wider sync scope. (Note that seq_cst fences have their own s_waitcnt vmcnt(0) and so do not need to be considered.)
- Ensures any preceding sequential consistent global memory instructions have completed before executing this sequentially consistent instruction. This prevents reordering a seq_cst store

				followed by a seq_cst load. (Note that seq_cst is stronger than acquire/release as the reordering of load acquire followed by a store release is prevented by the s_waitcnt of the release, but there is nothing preventing a store release followed by load acquire from completing out of order. The s_waitcnt could be placed after seq_store or before the seq_load. We choose the load to make the s_waitcnt be as late as possible so that the store may have already completed.) 2. *Following instructions same as corresponding load atomic acquire, except must generate all instructions even for OpenCL.*
store atomic	seq_cst	• singlethread • wavefront • workgroup • agent • system	• global • local • generic	*Same as corresponding store atomic release, except must generate all instructions even for OpenCL.*

atomicrmw	seq_cst	• singlethread • wavefront • workgroup • agent • system	• global • local • generic	*Same as corresponding atomicrmw acq_rel, except must generate all instructions even for OpenCL.*
fence	seq_cst	• singlethread • wavefront • workgroup • agent • system	*none*	*Same as corresponding fence acq_rel, except must generate all instructions even for OpenCL.*

Memory Model GFX940

For GFX940:

- Each agent has multiple shader arrays (SA).
- Each SA has multiple compute units (CU).
- Each CU has multiple SIMDs that execute wavefronts.
- The wavefronts for a single work-group are executed in the same CU but may be executed by different SIMDs. The exception is when in tgsplit execution mode when the wavefronts may be executed by different SIMDs in different CUs.
- Each CU has a single LDS memory shared by the wavefronts of the work-groups executing on it. The exception is when in tgsplit execution mode when no LDS is allocated as wavefronts of the same work-group can be in different CUs.
- All LDS operations of a CU are performed as wavefront wide operations in a global order and involve no caching. Completion is reported to a wavefront in execution order.
- The LDS memory has multiple request queues shared by the SIMDs of a CU. Therefore, the LDS operations performed by different wavefronts of a work-group can be reordered relative to each other, which can result in reordering the visibility of vector memory operations with respect to LDS operations of other wavefronts in the same work-group. A `s_waitcnt lgkmcnt(0)` is required to ensure synchronization between LDS operations and vector memory operations between wavefronts of a work-group, but not between operations performed by the same wavefront.
- The vector memory operations are performed as wavefront wide operations and completion is reported to a wavefront in execution order. The exception is that `flat_load/store/atomic` instructions can report out of vector memory order if they access LDS memory, and out of LDS operation order if they access global memory.

- The vector memory operations access a single vector L1 cache shared by all SIMDs a CU. Therefore:
  - No special action is required for coherence between the lanes of a single wavefront.
  - No special action is required for coherence between wavefronts in the same work-group since they execute on the same CU. The exception is when in tgsplit execution mode as wavefronts of the same work-group can be in different CUs and so a `buffer_inv sc0` is required which will invalidate the L1 cache.
  - A `buffer_inv sc0` is required to invalidate the L1 cache for coherence between wavefronts executing in different work-groups as they may be executing on different CUs.
  - Atomic read-modify-write instructions implicitly bypass the L1 cache. Therefore, they do not use the sc0 bit for coherence and instead use it to indicate if the instruction returns the original value being updated. They do use sc1 to indicate system or agent scope coherence.
- The scalar memory operations access a scalar L1 cache shared by all wavefronts on a group of CUs. The scalar and vector L1 caches are not coherent. However, scalar operations are used in a restricted way so do not impact the memory model. See Memory Spaces.
- The vector and scalar memory operations use an L2 cache.
  - The gfx940 can be configured as a number of smaller agents with each having a single L2 shared by all CUs on the same agent, or as fewer (possibly one) larger agents with groups of CUs on each agent each sharing separate L2 caches.
  - The L2 cache has independent channels to service disjoint ranges of virtual addresses.
  - Each CU has a separate request queue per channel for its associated L2. Therefore, the vector and scalar memory operations performed by wavefronts executing with different L1 caches and the same L2 cache can be reordered relative to each other.
  - A `s_waitcnt vmcnt(0)` is required to ensure synchronization between vector memory operations of different CUs. It ensures a previous vector memory operation has completed before executing a subsequent vector memory or LDS operation and so can be used to meet the requirements of acquire and release.
  - An L2 cache can be kept coherent with other L2 caches by using the MTYPE RW (read-write) for memory local to the L2, and MTYPE NC (non-coherent) with the PTE C-bit set for memory not local to the L2.
    - Any local memory cache lines will be automatically invalidated by writes from CUs associated with other L2 caches, or writes from the CPU, due to the cache probe caused by the PTE C-bit.
    - XGMI accesses from the CPU to local memory may be cached on the CPU. Subsequent access from the GPU will automatically invalidate or writeback the CPU cache due to the L2 probe filter.

        - To ensure coherence of local memory writes of CUs with different L1 caches in the same agent a `buffer_wbl2` is required. It does nothing if the agent is configured to have a single L2, or will writeback dirty L2 cache lines if configured to have multiple L2 caches.
        - To ensure coherence of local memory writes of CUs in different agents a `buffer_wbl2 sc1` is required. It will writeback dirty L2 cache lines.
        - To ensure coherence of local memory reads of CUs with different L1 caches in the same agent a `buffer_inv sc1` is required. It does nothing if the agent is configured to have a single L2, or will invalidate non-local L2 cache lines if configured to have multiple L2 caches.
        - To ensure coherence of local memory reads of CUs in different agents a `buffer_inv sc0 sc1` is required. It will invalidate non-local L2 cache lines if configured to have multiple L2 caches.
    - PCIe access from the GPU to the CPU can be kept coherent by using the MTYPE UC (uncached) which bypasses the L2.

Scalar memory operations are only used to access memory that is proven to not change during the execution of the kernel dispatch. This includes constant address space and global address space for program scope `const` variables. Therefore, the kernel machine code does not have to maintain the scalar cache to ensure it is coherent with the vector caches. The scalar and vector caches are invalidated between kernel dispatches by CP since constant address space data may change between kernel dispatch executions. See Memory Spaces.

The one exception is if scalar writes are used to spill SGPR registers. In this case the AMDGPU backend ensures the memory location used to spill is never accessed by vector memory operations at the same time. If scalar writes are used then a `s_dcache_wb` is inserted before the `s_endpgm` and before a function return since the locations may be used for vector memory instructions by a future wavefront that uses the same scratch area, or a function call that creates a frame at the same address, respectively. There is no need for a `s_dcache_inv` as all scalar writes are write-before-read in the same thread.

For kernarg backing memory:

- CP invalidates the L1 cache at the start of each kernel dispatch.
- On dGPU over XGMI or PCIe the kernarg backing memory is allocated in host memory accessed as MTYPE UC (uncached) to avoid needing to invalidate the L2 cache. This also causes it to be treated as non-volatile and so is not invalidated by `*_vol`.
- On APU the kernarg backing memory is accessed as MTYPE CC (cache coherent) and so the L2 cache will be coherent with the CPU and other agents.

Scratch backing memory (which is used for the private address space) is accessed with MTYPE NC_NV (non-coherent non-volatile). Since the private address space is only accessed by a single thread, and is always write-before-read, there is never a need to invalidate these entries from the L1 cache. Hence all cache invalidates are done as `*_vol` to only invalidate the volatile cache lines.

The code sequences used to implement the memory model for GFX940 are defined in table AMDHSA Memory Model Code Sequences GFX940.

AMDHSA Memory Model Code Sequences GFX940

LLVM Instr	LLVM Memory Ordering	LLVM Memory Sync Scope	AMDGPU Address Space	AMDGPU Machine Code GFX940
Non-Atomic				
load	*none*	*none*	• global • generic • private • constant	• !volatile & !nontemporal 1. buffer/global/flat_load • !volatile & nontemporal 1. buffer/global/flat_load nt=1 • volatile 1. buffer/global/flat_load sc0=1 sc1=1 2. s_waitcnt vmcnt(0) ◦ Must happen before any following volatile global/generic load/store. ◦ Ensures that volatile operations to different addresses will not be reordered by hardware.

load	*none*	*none*	• local	1. ds_load
store	*none*	*none*	• global • generic • private • constant	• !volatile & !nontemporal 1. buffer/global/flat_store • !volatile & nontemporal 1. buffer/global/flat_store nt=1 • volatile 1. buffer/global/flat_store sc0=1 sc1=1 2. s_waitcnt vmcnt(0) ◦ Must happen before any following volatile global/generic load/store. ◦ Ensures that volatile operations to different addresses will not be reordered by hardware.
store	*none*	*none*	• local	1. ds_store
Unordered Atomic				
load atomic	unordered	*any*	*any*	*Same as non-atomic*.
store atomic	unordered	*any*	*any*	*Same as non-atomic*.
atomicrmw	unordered	*any*	*any*	*Same as monotonic atomic*.
Monotonic Atomic				
load	monotonic	• singlethread	• global	1. buffer/global/flat_load

atomic		• wavefront	• generic	
load atomic	monotonic	• workgroup	• global • generic	1. buffer/global/flat_load sc0=1
load atomic	monotonic	• singlethread • wavefront • workgroup	• local	*If TgSplit execution mode, local address space cannot be used.* 1. ds_load
load atomic	monotonic	• agent	• global • generic	1. buffer/global/flat_load sc1=1
load atomic	monotonic	• system	• global • generic	1. buffer/global/flat_load sc0=1 sc1=1
store atomic	monotonic	• singlethread • wavefront	• global • generic	1. buffer/global/flat_store
store atomic	monotonic	• workgroup	• global • generic	1. buffer/global/flat_store sc0=1
store atomic	monotonic	• agent	• global • generic	1. buffer/global/flat_store sc1=1
store atomic	monotonic	• system	• global • generic	1. buffer/global/flat_store sc0=1 sc1=1
store atomic	monotonic	• singlethread • wavefront • workgroup	• local	*If TgSplit execution mode, local address space cannot be used.* 1. ds_store
atomicrmw	monotonic	• singlethread • wavefront • workgroup • agent	• global • generic	1. buffer/global/flat_atomic
atomicrmw	monotonic	• system	• global • generic	1. buffer/global/flat_atomic sc1=1

atomicrmw	monotonic	• singlethread • wavefront • workgroup	• local	*If TgSplit execution mode, local address space cannot be used.* 1. ds_atomic
Acquire Atomic				
load atomic	acquire	• singlethread • wavefront	• global • local • generic	1. buffer/global/ds/flat_load
load atomic	acquire	• workgroup	• global	1. buffer/global_load sc0=1 2. s_waitcnt vmcnt(0) • If not TgSplit execution mode, omit. • Must happen before the following buffer_inv. 3. buffer_inv sc0=1 • If not TgSplit execution mode, omit. • Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures that following loads will not see stale data.
load atomic	acquire	• workgroup	• local	*If TgSplit execution mode, local address space cannot be used.* 1. ds_load 2. s_waitcnt lgkmcnt(0)

				• If OpenCL, omit. • Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures any following global data read is no older than the local load atomic value being acquired.
load atomic	acquire	• workgroup	• generic	1. flat_load sc0=1 2. s_waitcnt lgkm/vmcnt(0) • Use lgkmcnt(0) if not TgSplit execution mode and vmcnt(0) if TgSplit execution mode. • If OpenCL, omit lgkmcnt(0). • Must happen before the following buffer_inv and any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures any following global data read is no older than a local load atomic value being acquired.

				3. buffer_inv sc0=1 • If not TgSplit execution mode, omit. • Ensures that following loads will not see stale data.
load atomic	acquire	• agent	• global	1. buffer/global_load sc1=1 2. s_waitcnt vmcnt(0) • Must happen before following buffer_inv. • Ensures the load has completed before invalidating the cache. 3. buffer_inv sc1=1 • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale global data.
load atomic	acquire	• system	• global	1. buffer/global/flat_load sc0=1 sc1=1 2. s_waitcnt vmcnt(0) • Must happen before following buffer_inv. • Ensures the load has completed before invalidating

				the cache. 3. buffer_inv sc0=1 sc1=1 • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale MTYPE NC global data. MTYPE RW and CC memory will never be stale due to the memory probes.
load atomic	acquire	• agent	• generic	1. flat_load sc1=1 2. s_waitcnt vmcnt(0) & lgkmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL omit lgkmcnt(0). • Must happen before following buffer_inv. • Ensures the flat_load has completed before invalidating the cache. 3. buffer_inv sc1=1 • Must happen before any following global/generic load/load

				atomic/atomicrmw. • Ensures that following loads will not see stale global data.
load atomic	acquire	• system	• generic	1. flat_load sc0=1 sc1=1 2. s_waitcnt vmcnt(0) & lgkmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL omit lgkmcnt(0). • Must happen before the following buffer_inv. • Ensures the flat_load has completed before invalidating the caches. 3. buffer_inv sc0=1 sc1=1 • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale MTYPE NC global data. MTYPE RW and CC memory will never be stale due to the memory probes.

atomicrmw	acquire	singlethread • wavefront	global • generic	1. buffer/global/flat_atomic
atomicrmw	acquire	• singlethread • wavefront	• local	*If TgSplit execution mode, local address space cannot be used.* 1. ds_atomic
atomicrmw	acquire	• workgroup	• global	1. buffer/global_atomic 2. s_waitcnt vmcnt(0) • If not TgSplit execution mode, omit. • Must happen before the following buffer_inv. • Ensures the atomicrmw has completed before invalidating the cache. 3. buffer_inv sc0=1 • If not TgSplit execution mode, omit. • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale global data.
atomicrmw	acquire	• workgroup	• local	*If TgSplit execution mode, local address space cannot be used.*

				1. ds_atomic 2. s_waitcnt lgkmcnt(0) • If OpenCL, omit. • Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures any following global data read is no older than the local atomicrmw value being acquired.
atomicrmw	acquire	• workgroup	• generic	1. flat_atomic 2. s_waitcnt lgkm/vmcnt(0) • Use lgkmcnt(0) if not TgSplit execution mode and vmcnt(0) if TgSplit execution mode. • If OpenCL, omit lgkmcnt(0). • Must happen before the following buffer_inv and any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures any following global data read is no

				older than a local atomicrmw value being acquired. 3. buffer_inv sc0=1 • If not TgSplit execution mode, omit. • Ensures that following loads will not see stale data.
atomicrmw	acquire	• agent	• global	1. buffer/global_atomic 2. s_waitcnt vmcnt(0) • Must happen before following buffer_inv. • Ensures the atomicrmw has completed before invalidating the cache. 3. buffer_inv sc1=1 • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale global data.
atomicrmw	acquire	• system	• global	1. buffer/global_atomic sc1=1 2. s_waitcnt vmcnt(0) • Must happen before

				following buffer_inv. • Ensures the atomicrmw has completed before invalidating the caches. 3. buffer_inv sc0=1 sc1=1 • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale MTYPE NC global data. MTYPE RW and CC memory will never be stale due to the memory probes.
atomicrmw	acquire	• agent	• generic	1. flat_atomic 2. s_waitcnt vmcnt(0) & lgkmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL, omit lgkmcnt(0). • Must happen before following buffer_inv. • Ensures the atomicrmw has completed before invalidating the cache.

				3. buffer_inv sc1=1 • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale global data.
atomicrmw	acquire	• system	• generic	1. flat_atomic sc1=1 2. s_waitcnt vmcnt(0) & lgkmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL, omit lgkmcnt(0). • Must happen before following buffer_inv. • Ensures the atomicrmw has completed before invalidating the caches. 3. buffer_inv sc0=1 sc1=1 • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale MTYPE NC global data.

				MTYPE RW and CC memory will never be stale due to the memory probes.
fence	acquire	• singlethread • wavefront	*none*	*none*
fence	acquire	• workgroup	*none*	1. s_waitcnt lgkm/vmcnt(0) • Use lgkmcnt(0) if not TgSplit execution mode and vmcnt(0) if TgSplit execution mode. • If OpenCL and address space is not generic, omit lgkmcnt(0). • If OpenCL and address space is local, omit vmcnt(0). • However, since LLVM currently has no address space on the fence need to conservatively always generate. If fence had an address space then set to address space of OpenCL fence flag, or to generic if both local and global flags are specified. • s_waitcnt vmcnt(0) must happen after any preceding

global/generic load atomic/ atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic).
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic).
- Must happen before the following buffer_inv and any following global/generic load/load atomic/store/store atomic/atomicrmw.
- Ensures any following global data read is no older than the value read by the fence-paired-atomic.

3. buffer_inv sc0=1

				• If not TgSplit execution mode, omit. • Ensures that following loads will not see stale data.
fence	acquire	• agent	*none*	1. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL and address space is not generic, omit lgkmcnt(0). • However, since LLVM currently has no address space on the fence need to conservatively always generate (see comment for previous fence). • Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after any preceding global/generic load atomic/atomicrmw

with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic).
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic).
- Must happen before the following buffer_inv.
- Ensures that the fence-paired atomic has completed before invalidating the cache. Therefore any following locations read must be no older than the value read by the fence-paired-atomic.

2. buffer_inv sc1=1

   - Must happen before any following

				global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures that following loads will not see stale global data.
fence	acquire	• system	*none*	1. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL and address space is not generic, omit lgkmcnt(0). • However, since LLVM currently has no address space on the fence need to conservatively always generate (see comment for previous fence). • Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after any preceding

global/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic).
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic).
- Must happen before the following buffer_inv.
- Ensures that the fence-paired atomic has completed before invalidating the cache. Therefore any following locations read must be no older than the value read by the fence-paired-atomic.

2. buffer_inv sc0=1 sc1=1

				• Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures that following loads will not see stale global data.	
Release Atomic					
store atomic	release	• singlethread • wavefront	• global • generic	1. buffer/global/flat_store	
store atomic	release	• singlethread • wavefront	• local	*If TgSplit execution mode, local address space cannot be used.* 1. ds_store	
store atomic	release	• workgroup	• global • generic	1. s_waitcnt lgkm/vmcnt(0) • Use lgkmcnt(0) if not TgSplit execution mode and vmcnt(0) if TgSplit execution mode. • If OpenCL, omit lgkmcnt(0). • s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/ load atomic/store atomic/ atomicrmw. • s_waitcnt lgkmcnt(0) must happen after any	

				preceding local/generic load/store/load atomic/store atomic/atomicrmw. • Must happen before the following store. • Ensures that all memory operations have completed before performing the store that is being released. 2. buffer/global/flat_store sc0=1
store atomic	release	• workgroup	• local	*If TgSplit execution mode, local address space cannot be used.* 1. ds_store
store atomic	release	• agent	• global • generic	1. buffer_wbl2 sc1=1 • Must happen before following s_waitcnt. • Performs L2 writeback to ensure previous global/generic store/atomicrmw are visible at agent scope. 2. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL and address space is not

generic, omit lgkmcnt(0).
- Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.
- s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before the following store.
- Ensures that all memory operations to memory have completed before performing the store that is being released.

3. buffer/global/flat_store sc1=1

store atomic	release	system	global • generic	1. buffer_wbl2 sc0=1 sc1=1 • Must happen before following s_waitcnt. • Performs L2 writeback to ensure previous global/generic store/atomicrmw are visible at system scope. 2. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL and address space is not generic, omit lgkmcnt(0). • Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw. • s_waitcnt lgkmcnt(0) must

				happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw. • Must happen before the following store. • Ensures that all memory operations to memory and the L2 writeback have completed before performing the store that is being released. 3. buffer/global/flat_store sc0=1 sc1=1
atomicrmw	release	• singlethread • wavefront	• global • generic	1. buffer/global/flat_atomic
atomicrmw	release	• singlethread • wavefront	• local	*If TgSplit execution mode, local address space cannot be used.* 1. ds_atomic
atomicrmw	release	• workgroup	• global • generic	1. s_waitcnt lgkm/vmcnt(0) • Use lgkmcnt(0) if not TgSplit execution mode and vmcnt(0) if TgSplit execution mode. • If OpenCL, omit lgkmcnt(0). • s_waitcnt vmcnt(0) must happen after any preceding

				global/generic load/store/ load atomic/store atomic/ atomicrmw. • s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw. • Must happen before the following atomicrmw. • Ensures that all memory operations have completed before performing the atomicrmw that is being released. 2. buffer/global/flat_atomic sc0=1	
atomicrmw	release	• workgroup	• local	*If TgSplit execution mode, local address space cannot be used.* 1. ds_atomic	
atomicrmw	release	• agent	• global • generic	1. buffer_wbl2 sc1=1 • Must happen before following s_waitcnt. • Performs L2 writeback to ensure previous global/generic store/atomicrmw	

are visible at agent scope.

2. s_waitcnt lgkmcnt(0) & vmcnt(0)

    - If TgSplit execution mode, omit lgkmcnt(0).
    - If OpenCL, omit lgkmcnt(0).
    - Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.
    - s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw.
    - s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
    - Must happen before the following atomicrmw.
    - Ensures that all memory operations

				to global and local have completed before performing the atomicrmw that is being released. 3. buffer/global/flat_atomic sc1=1
atomicrmw	release	• system	• global • generic	1. buffer_wbl2 sc0=1 sc1=1 • Must happen before following s_waitcnt. • Performs L2 writeback to ensure previous global/generic store/atomicrmw are visible at system scope. 2. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL, omit lgkmcnt(0). • Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after

				any preceding global/generic load/store/load atomic/store atomic/atomicrmw. • s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw. • Must happen before the following atomicrmw. • Ensures that all memory operations to memory and the L2 writeback have completed before performing the store that is being released. 3. buffer/global/flat_atomic sc0=1 sc1=1
fence	release	• singlethread • wavefront	*none*	*none*
fence	release	• workgroup	*none*	1. s_waitcnt lgkm/vmcnt(0) • Use lgkmcnt(0) if not TgSplit execution mode and vmcnt(0) if TgSplit execution mode. • If OpenCL and

address space is not generic, omit lgkmcnt(0).
- If OpenCL and address space is local, omit vmcnt(0).
- However, since LLVM currently has no address space on the fence need to conservatively always generate. If fence had an address space then set to address space of OpenCL fence flag, or to generic if both local and global flags are specified.
- s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/ load atomic/store atomic/ atomicrmw.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/load atomic/store/store atomic/atomicrmw.
- Must happen before any following store atomic/atomicrmw

				with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic). • Ensures that all memory operations have completed before performing the following fence-paired-atomic.
fence	release	• agent	*none*	1. buffer_wbl2 sc1=1 • If OpenCL and address space is local, omit. • Must happen before following s_waitcnt. • Performs L2 writeback to ensure previous global/generic store/atomicrmw are visible at agent scope. 2. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL and address space is not generic, omit lgkmcnt(0). • If OpenCL and

address space is local, omit vmcnt(0).
- However, since LLVM currently has no address space on the fence need to conservatively always generate. If fence had an address space then set to address space of OpenCL fence flag, or to generic if both local and global flags are specified.
- Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.
- s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load

				atomic/store atomic/atomicrmw. • Must happen before any following store atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic). • Ensures that all memory operations have completed before performing the following fence-paired-atomic.
fence	release	• system	*none*	1. buffer_wbl2 sc0=1 sc1=1 • Must happen before following s_waitcnt. • Performs L2 writeback to ensure previous global/generic store/atomicrmw are visible at system scope. 2. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL and address space is not generic, omit

lgkmcnt(0).

- If OpenCL and address space is local, omit vmcnt(0).
- However, since LLVM currently has no address space on the fence need to conservatively always generate. If fence had an address space then set to address space of OpenCL fence flag, or to generic if both local and global flags are specified.
- Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.
- s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw.
- s_waitcnt lgkmcnt(0) must happen after any preceding

				local/generic load/store/load atomic/store atomic/atomicrmw. • Must happen before any following store atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic). • Ensures that all memory operations have completed before performing the following fence-paired-atomic.	
Acquire-Release Atomic					
atomicrmw	acq_rel	• singlethread • wavefront	• global • generic	1. buffer/global/flat_atomic	
atomicrmw	acq_rel	• singlethread • wavefront	• local	*If TgSplit execution mode, local address space cannot be used.* 1. ds_atomic	
atomicrmw	acq_rel	• workgroup	• global	1. s_waitcnt lgkm/vmcnt(0) • Use lgkmcnt(0) if not TgSplit execution mode and vmcnt(0) if TgSplit execution mode. • If OpenCL, omit lgkmcnt(0).	

- Must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/ load atomic/store atomic/ atomicrmw.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before the following atomicrmw.
- Ensures that all memory operations have completed before performing the atomicrmw that is being released.

2. buffer/global_atomic
3. s_waitcnt vmcnt(0)

    - If not TgSplit execution mode, omit.
    - Must happen before the following

				buffer_inv. • Ensures any following global data read is no older than the atomicrmw value being acquired. 4. buffer_inv sc0=1 • If not TgSplit execution mode, omit. • Ensures that following loads will not see stale data.
atomicrmw	acq_rel	• workgroup	• local	*If TgSplit execution mode, local address space cannot be used.* 1. ds_atomic 2. s_waitcnt lgkmcnt(0) • If OpenCL, omit. • Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures any following global data read is no older than the local load atomic value being acquired.
atomicrmw	acq_rel	• workgroup	• generic	1. s_waitcnt lgkm/vmcnt(0) • Use lgkmcnt(0) if

not TgSplit execution mode and vmcnt(0) if TgSplit execution mode.
- If OpenCL, omit lgkmcnt(0).
- s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/ load atomic/store atomic/ atomicrmw.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before the following atomicrmw.
- Ensures that all memory operations have completed before performing the atomicrmw that is being released.

2. flat_atomic
3. s_waitcnt lgkmcnt(0) & vmcnt(0)

    - If not TgSplit execution mode, omit vmcnt(0).
    - If OpenCL, omit lgkmcnt(0).

				• Must happen before the following buffer_inv and any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures any following global data read is no older than a local load atomic value being acquired. 3. buffer_inv sc0=1 • If not TgSplit execution mode, omit. • Ensures that following loads will not see stale data.
atomicrmw	acq_rel	• agent	• global	1. buffer_wbl2 sc1=1 • Must happen before following s_waitcnt. • Performs L2 writeback to ensure previous global/generic store/atomicrmw are visible at agent scope. 2. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit

lgkmcnt(0).
- If OpenCL, omit lgkmcnt(0).
- Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.
- s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before the following atomicrmw.
- Ensures that all memory operations to global have completed before performing the atomicrmw that is being released.

				3. buffer/global_atomic 4. s_waitcnt vmcnt(0) • Must happen before following buffer_inv. • Ensures the atomicrmw has completed before invalidating the cache. 5. buffer_inv sc1=1 • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale global data.
atomicrmw	acq_rel	• system	• global	1. buffer_wbl2 sc0=1 sc1=1 • Must happen before following s_waitcnt. • Performs L2 writeback to ensure previous global/generic store/atomicrmw are visible at system scope. 2. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0).

- If OpenCL, omit lgkmcnt(0).
- Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.
- s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before the following atomicrmw.
- Ensures that all memory operations to global and L2 writeback have completed before performing the atomicrmw that is being released.

				3. buffer/global_atomic sc1=1 4. s_waitcnt vmcnt(0) • Must happen before following buffer_inv. • Ensures the atomicrmw has completed before invalidating the caches. 5. buffer_inv sc0=1 sc1=1 • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale MTYPE NC global data. MTYPE RW and CC memory will never be stale due to the memory probes.
atomicrmw	acq_rel	• agent	• generic	1. buffer_wbl2 sc1=1 • Must happen before following s_waitcnt. • Performs L2 writeback to ensure previous global/generic store/atomicrmw are visible at agent scope. 2. s_waitcnt lgkmcnt(0) & vmcnt(0)

- If TgSplit execution mode, omit lgkmcnt(0).
- If OpenCL, omit lgkmcnt(0).
- Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.
- s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before the following atomicrmw.
- Ensures that all memory operations to global have completed before performing the

				atomicrmw that is being released. 3. flat_atomic 4. s_waitcnt vmcnt(0) & lgkmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • If OpenCL, omit lgkmcnt(0). • Must happen before following buffer_inv. • Ensures the atomicrmw has completed before invalidating the cache. 5. buffer_inv sc1=1 • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale global data.
atomicrmw	acq_rel	• system	• generic	1. buffer_wbl2 sc0=1 sc1=1 • Must happen before following s_waitcnt. • Performs L2 writeback to ensure previous global/generic store/atomicrmw

are visible at system scope.

2. s_waitcnt lgkmcnt(0) & vmcnt(0)

    - If TgSplit execution mode, omit lgkmcnt(0).
    - If OpenCL, omit lgkmcnt(0).
    - Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.
    - s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw.
    - s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
    - Must happen before the following atomicrmw.
    - Ensures that all memory operations

to global and L2 writeback have completed before performing the atomicrmw that is being released.

3. flat_atomic sc1=1
4. s_waitcnt vmcnt(0) & lgkmcnt(0)

    - If TgSplit execution mode, omit lgkmcnt(0).
    - If OpenCL, omit lgkmcnt(0).
    - Must happen before following buffer_inv.
    - Ensures the atomicrmw has completed before invalidating the caches.

5. buffer_inv sc0=1 sc1=1

    - Must happen before any following global/generic load/load atomic/atomicrmw.
    - Ensures that following loads will not see stale MTYPE NC global data. MTYPE RW and CC memory will never be stale due to the memory probes.

fence	acq_rel	singlethread • wavefront	*none*	*none*
fence	acq_rel	• workgroup	*none*	1. s_waitcnt lgkm/vmcnt(0) • Use lgkmcnt(0) if not TgSplit execution mode and vmcnt(0) if TgSplit execution mode. • If OpenCL and address space is not generic, omit lgkmcnt(0). • If OpenCL and address space is local, omit vmcnt(0). • However, since LLVM currently has no address space on the fence need to conservatively always generate (see comment for previous fence). • s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/ load atomic/store atomic/ atomicrmw. • s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/load

atomic/store/store atomic/atomicrmw.
- Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw.
- Ensures that all memory operations have completed before performing any following global memory operations.
- Ensures that the preceding local/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the acquire-fence-paired-atomic) has completed before following global memory operations. This satisfies the requirements of acquire.
- Ensures that all previous memory operations have completed before a following local/generic store

				atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the release-fence-paired-atomic). This satisfies the requirements of release. • Must happen before the following buffer_inv. • Ensures that the acquire-fence-paired atomic has completed before invalidating the cache. Therefore any following locations read must be no older than the value read by the acquire-fence-paired-atomic. 3. buffer_inv sc0=1 • If not TgSplit execution mode, omit. • Ensures that following loads will not see stale data.
fence	acq_rel	• agent	*none*	1. buffer_wbl2 sc1=1

- If OpenCL and address space is local, omit.
- Must happen before following s_waitcnt.
- Performs L2 writeback to ensure previous global/generic store/atomicrmw are visible at agent scope.

2. s_waitcnt lgkmcnt(0) & vmcnt(0)

   - If TgSplit execution mode, omit lgkmcnt(0).
   - If OpenCL and address space is not generic, omit lgkmcnt(0).
   - However, since LLVM currently has no address space on the fence need to conservatively always generate (see comment for previous fence).
   - Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.

- s_waitcnt vmcnt(0) must happen after any preceding global/generic load/store/load atomic/store atomic/atomicrmw.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before the following buffer_inv.
- Ensures that the preceding global/local/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the acquire-fence-paired-atomic) has completed before invalidating the cache. This satisfies the requirements of acquire.
- Ensures that all previous memory

				operations have completed before a following global/local/generic store atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the release-fence-paired-atomic). This satisfies the requirements of release. 3. buffer_inv sc1=1 • Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures that following loads will not see stale global data. This satisfies the requirements of acquire.
fence	acq_rel	• system	*none*	1. buffer_wbl2 sc0=1 sc1=1 • If OpenCL and address space is local, omit. • Must happen before

following s_waitcnt.
- Performs L2 writeback to ensure previous global/generic store/atomicrmw are visible at system scope.

1. s_waitcnt lgkmcnt(0) & vmcnt(0)

   - If TgSplit execution mode, omit lgkmcnt(0).
   - If OpenCL and address space is not generic, omit lgkmcnt(0).
   - However, since LLVM currently has no address space on the fence need to conservatively always generate (see comment for previous fence).
   - Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.
   - s_waitcnt vmcnt(0) must happen after any preceding global/generic

load/store/load atomic/store atomic/atomicrmw.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before the following buffer_inv.
- Ensures that the preceding global/local/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the acquire-fence-paired-atomic) has completed before invalidating the cache. This satisfies the requirements of acquire.
- Ensures that all previous memory operations have completed before a following global/local/generic

				store atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the release-fence-paired-atomic). This satisfies the requirements of release. 2. buffer_inv sc0=1 sc1=1 • Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures that following loads will not see stale MTYPE NC global data. MTYPE RW and CC memory will never be stale due to the memory probes.
Sequential Consistent Atomic				
load atomic	seq_cst	• singlethread • wavefront	• global • local • generic	*Same as corresponding load atomic acquire, except must generate all instructions even for OpenCL.*
load atomic	seq_cst	• workgroup	• global • generic	1. s_waitcnt lgkm/vmcnt(0) • Use lgkmcnt(0) if

not TgSplit execution mode and vmcnt(0) if TgSplit execution mode.
- s_waitcnt lgkmcnt(0) must happen after preceding local/generic load atomic/store atomic/atomicrmw with memory ordering of seq_cst and with equal or wider sync scope. (Note that seq_cst fences have their own s_waitcnt lgkmcnt(0) and so do not need to be considered.)
- s_waitcnt vmcnt(0) must happen after preceding global/generic load atomic/store atomic/atomicrmw with memory ordering of seq_cst and with equal or wider sync scope. (Note that seq_cst fences have their own s_waitcnt vmcnt(0) and so do not need to be considered.)
- Ensures any

preceding sequential consistent global/local memory instructions have completed before executing this sequentially consistent instruction. This prevents reordering a seq_cst store followed by a seq_cst load. (Note that seq_cst is stronger than acquire/release as the reordering of load acquire followed by a store release is prevented by the s_waitcnt of the release, but there is nothing preventing a store release followed by load acquire from completing out of order. The s_waitcnt could be placed after seq_store or before the seq_load. We choose the load to make the s_waitcnt be as late as possible so that the store may have

				already completed.) 2. *Following instructions same as corresponding load atomic acquire, except must generate all instructions even for OpenCL.*
load atomic	seq_cst	• workgroup	• local	*If TgSplit execution mode, local address space cannot be used.* *Same as corresponding load atomic acquire, except must generate all instructions even for OpenCL.*
load atomic	seq_cst	• agent • system	• global • generic	1. s_waitcnt lgkmcnt(0) & vmcnt(0) • If TgSplit execution mode, omit lgkmcnt(0). • Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt lgkmcnt(0) must happen after preceding global/generic load atomic/store atomic/atomicrmw with memory ordering of seq_cst and with equal or wider sync scope.

(Note that seq_cst fences have their own s_waitcnt lgkmcnt(0) and so do not need to be considered.)
- s_waitcnt vmcnt(0) must happen after preceding global/generic load atomic/store atomic/atomicrmw with memory ordering of seq_cst and with equal or wider sync scope. (Note that seq_cst fences have their own s_waitcnt vmcnt(0) and so do not need to be considered.)
- Ensures any preceding sequential consistent global memory instructions have completed before executing this sequentially consistent instruction. This prevents reordering a seq_cst store followed by a seq_cst load. (Note that seq_cst is

				stronger than acquire/release as the reordering of load acquire followed by a store release is prevented by the s_waitcnt of the release, but there is nothing preventing a store release followed by load acquire from completing out of order. The s_waitcnt could be placed after seq_store or before the seq_load. We choose the load to make the s_waitcnt be as late as possible so that the store may have already completed.) 2. *Following instructions same as corresponding load atomic acquire, except must generate all instructions even for OpenCL.*
store atomic	seq_cst	• singlethread • wavefront • workgroup • agent • system	• global • local • generic	*Same as corresponding store atomic release, except must generate all instructions even for OpenCL.*
atomicrmw	seq_cst	• singlethread • wavefront • workgroup	• global • local • generic	*Same as corresponding atomicrmw acq_rel, except must generate all instructions even for OpenCL.*

		• agent • system		
fence	seq_cst	• singlethread • wavefront • workgroup • agent • system	*none*	*Same as corresponding fence acq_rel, except must generate all instructions even for OpenCL.*

Memory Model GFX10-GFX11

For GFX10-GFX11:

- Each agent has multiple shader arrays (SA).
- Each SA has multiple work-group processors (WGP).
- Each WGP has multiple compute units (CU).
- Each CU has multiple SIMDs that execute wavefronts.
- The wavefronts for a single work-group are executed in the same WGP. In CU wavefront execution mode the wavefronts may be executed by different SIMDs in the same CU. In WGP wavefront execution mode the wavefronts may be executed by different SIMDs in different CUs in the same WGP.
- Each WGP has a single LDS memory shared by the wavefronts of the work-groups executing on it.
- All LDS operations of a WGP are performed as wavefront wide operations in a global order and involve no caching. Completion is reported to a wavefront in execution order.
- The LDS memory has multiple request queues shared by the SIMDs of a WGP. Therefore, the LDS operations performed by different wavefronts of a work-group can be reordered relative to each other, which can result in reordering the visibility of vector memory operations with respect to LDS operations of other wavefronts in the same work-group. A `s_waitcnt lgkmcnt(0)` is required to ensure synchronization between LDS operations and vector memory operations between wavefronts of a work-group, but not between operations performed by the same wavefront.
- The vector memory operations are performed as wavefront wide operations. Completion of load/store/sample operations are reported to a wavefront in execution order of other load/store/sample operations performed by that wavefront.
- The vector memory operations access a vector L0 cache. There is a single L0 cache per CU. Each SIMD of a CU accesses the same L0 cache. Therefore, no special action is required for coherence between the lanes of a single wavefront. However, a `buffer_gl0_inv` is required for

coherence between wavefronts executing in the same work-group as they may be executing on SIMDs of different CUs that access different L0s. A `buffer_gl0_inv` is also required for coherence between wavefronts executing in different work-groups as they may be executing on different WGPs.

- The scalar memory operations access a scalar L0 cache shared by all wavefronts on a WGP. The scalar and vector L0 caches are not coherent. However, scalar operations are used in a restricted way so do not impact the memory model. See Memory Spaces.
- The vector and scalar memory L0 caches use an L1 cache shared by all WGPs on the same SA. Therefore, no special action is required for coherence between the wavefronts of a single work-group. However, a `buffer_gl1_inv` is required for coherence between wavefronts executing in different work-groups as they may be executing on different SAs that access different L1s.
- The L1 caches have independent quadrants to service disjoint ranges of virtual addresses.
- Each L0 cache has a separate request queue per L1 quadrant. Therefore, the vector and scalar memory operations performed by different wavefronts, whether executing in the same or different work-groups (which may be executing on different CUs accessing different L0s), can be reordered relative to each other. A `s_waitcnt vmcnt(0) & vscnt(0)` is required to ensure synchronization between vector memory operations of different wavefronts. It ensures a previous vector memory operation has completed before executing a subsequent vector memory or LDS operation and so can be used to meet the requirements of acquire, release and sequential consistency.
- The L1 caches use an L2 cache shared by all SAs on the same agent.
- The L2 cache has independent channels to service disjoint ranges of virtual addresses.
- Each L1 quadrant of a single SA accesses a different L2 channel. Each L1 quadrant has a separate request queue per L2 channel. Therefore, the vector and scalar memory operations performed by wavefronts executing in different work-groups (which may be executing on different SAs) of an agent can be reordered relative to each other. A `s_waitcnt vmcnt(0) & vscnt(0)` is required to ensure synchronization between vector memory operations of different SAs. It ensures a previous vector memory operation has completed before executing a subsequent vector memory and so can be used to meet the requirements of acquire, release and sequential consistency.
- The L2 cache can be kept coherent with other agents on some targets, or ranges of virtual addresses can be set up to bypass it to ensure system coherence.
- On GFX10.3 and GFX11 a memory attached last level (MALL) cache exists for GPU memory. The MALL cache is fully coherent with GPU memory and has no impact on system coherence. All agents (GPU and CPU) access GPU memory through the MALL cache.

Scalar memory operations are only used to access memory that is proven to not change during the

execution of the kernel dispatch. This includes constant address space and global address space for program scope `const` variables. Therefore, the kernel machine code does not have to maintain the scalar cache to ensure it is coherent with the vector caches. The scalar and vector caches are invalidated between kernel dispatches by CP since constant address space data may change between kernel dispatch executions. See Memory Spaces.

The one exception is if scalar writes are used to spill SGPR registers. In this case the AMDGPU backend ensures the memory location used to spill is never accessed by vector memory operations at the same time. If scalar writes are used then a `s_dcache_wb` is inserted before the `s_endpgm` and before a function return since the locations may be used for vector memory instructions by a future wavefront that uses the same scratch area, or a function call that creates a frame at the same address, respectively. There is no need for a `s_dcache_inv` as all scalar writes are write-before-read in the same thread.

For kernarg backing memory:

- CP invalidates the L0 and L1 caches at the start of each kernel dispatch.
- On dGPU the kernarg backing memory is accessed as MTYPE UC (uncached) to avoid needing to invalidate the L2 cache.
- On APU the kernarg backing memory is accessed as MTYPE CC (cache coherent) and so the L2 cache will be coherent with the CPU and other agents.

Scratch backing memory (which is used for the private address space) is accessed with MTYPE NC (non-coherent). Since the private address space is only accessed by a single thread, and is always write-before-read, there is never a need to invalidate these entries from the L0 or L1 caches.

Wavefronts are executed in native mode with in-order reporting of loads and sample instructions. In this mode vmcnt reports completion of load, atomic with return and sample instructions in order, and the vscnt reports the completion of store and atomic without return in order. See `MEM_ORDERED` field in compute_pgm_rsrc1 for GFX6-GFX11.

Wavefronts can be executed in WGP or CU wavefront execution mode:

- In WGP wavefront execution mode the wavefronts of a work-group are executed on the SIMDs of both CUs of the WGP. Therefore, explicit management of the per CU L0 caches is required for work-group synchronization. Also accesses to L1 at work-group scope need to be explicitly ordered as the accesses from different CUs are not ordered.
- In CU wavefront execution mode the wavefronts of a work-group are executed on the SIMDs of a single CU of the WGP. Therefore, all global memory access by the work-group access the same L0 which in turn ensures L1 accesses are ordered and so do not require explicit management of the caches for work-group synchronization.

See `WGP_MODE` field in compute_pgm_rsrc1 for GFX6-GFX11 and Target Features.

The code sequences used to implement the memory model for GFX10-GFX11 are defined in table AMDHSA Memory Model Code Sequences GFX10-GFX11.

AMDHSA Memory Model Code Sequences GFX10-GFX11

LLVM Instr	LLVM Memory Ordering	LLVM Memory Sync Scope	AMDGPU Address Space	AMDGPU Machine Code GFX10-GFX11
Non-Atomic				
load	*none*	*none*	• global • generic • private • constant	• !volatile & !nontemporal 1. buffer/global/flat_load • !volatile & nontemporal 1. buffer/global/flat_load slc=1 dlc=1 ◦ If GFX10, omit dlc=1. • volatile 1. buffer/global/flat_load glc=1 dlc=1 2. s_waitcnt vmcnt(0) ◦ Must happen before any following volatile global/generic load/store. ◦ Ensures that volatile operations to different addresses will not be reordered by hardware.

load	*none*	*none*	• local	1. ds_load
store	*none*	*none*	• global • generic • private • constant	• !volatile & !nontemporal 1. buffer/global/flat_store • !volatile & nontemporal 1. buffer/global/flat_store glc=1 slc=1 dlc=1 ◦ If GFX10, omit dlc=1. • volatile 1. buffer/global/flat_store dlc=1 ◦ If GFX10, omit dlc=1. 2. s_waitcnt vscnt(0) ◦ Must happen before any following volatile global/generic load/store. ◦ Ensures that volatile operations to different addresses will not be reordered by hardware.
store	*none*	*none*	• local	1. ds_store
Unordered Atomic				

load atomic	unordered	*any*	*any*	*Same as non-atomic*.
store atomic	unordered	*any*	*any*	*Same as non-atomic*.
atomicrmw	unordered	*any*	*any*	*Same as monotonic atomic*.
Monotonic Atomic				
load atomic	monotonic	• singlethread • wavefront	• global • generic	1. buffer/global/flat_load
load atomic	monotonic	• workgroup	• global • generic	1. buffer/global/flat_load glc=1 • If CU wavefront execution mode, omit glc=1.
load atomic	monotonic	• singlethread • wavefront • workgroup	• local	1. ds_load
load atomic	monotonic	• agent • system	• global • generic	1. buffer/global/flat_load glc=1 dlc=1 • If GFX11, omit dlc=1.
store atomic	monotonic	• singlethread • wavefront • workgroup • agent • system	• global • generic	1. buffer/global/flat_store
store atomic	monotonic	• singlethread • wavefront • workgroup	• local	1. ds_store
atomicrmw	monotonic	• singlethread • wavefront • workgroup • agent	• global • generic	1. buffer/global/flat_atomic

		• system		
atomicrmw	monotonic	• singlethread • wavefront • workgroup	• local	1. ds_atomic
Acquire Atomic				
load atomic	acquire	• singlethread • wavefront	• global • local • generic	1. buffer/global/ds/flat_load

load atomic	acquire	• workgroup	• global	1. buffer/global_load glc=1
				  • If CU wavefront execution mode, omit glc=1.
				2. s_waitcnt vmcnt(0)
				  • If CU wavefront execution mode, omit.
				  • Must happen before the following buffer_gl0_inv and before any following global/generic load/load atomic/store/store atomic/atomicrmw.
				3. buffer_gl0_inv
				  • If CU wavefront execution mode, omit.
				  • Ensures that following loads will not see stale data.

load atomic	acquire	• workgroup	• local	1. ds_load 2. s_waitcnt lgkmcnt(0) • If OpenCL, omit. • Must happen before the following buffer_gl0_inv and before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures any following global data read is no older than the local load atomic value being acquired. 3. buffer_gl0_inv • If CU wavefront execution mode, omit. • If OpenCL, omit. • Ensures that following loads will not see stale data.
load atomic	acquire	• workgroup	• generic	1. flat_load glc=1 • If CU wavefront execution mode, omit glc=1. 2. s_waitcnt lgkmcnt(0) & vmcnt(0) • If CU wavefront execution mode, omit vmcnt(0).

				• If OpenCL, omit lgkmcnt(0). • Must happen before the following buffer_gl0_inv and any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures any following global data read is no older than a local load atomic value being acquired. 3. buffer_gl0_inv • If CU wavefront execution mode, omit. • Ensures that following loads will not see stale data.
load atomic	acquire	• agent • system	• global	1. buffer/global_load glc=1 dlc=1 • If GFX11, omit dlc=1. 2. s_waitcnt vmcnt(0) • Must happen before following buffer_gl*_inv. • Ensures the load has completed before invalidating the caches.

				3. buffer_gl0_inv; buffer_gl1_inv • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale global data.
load atomic	acquire	• agent • system	• generic	1. flat_load glc=1 dlc=1 • If GFX11, omit dlc=1. 2. s_waitcnt vmcnt(0) & lgkmcnt(0) • If OpenCL omit lgkmcnt(0). • Must happen before following buffer_gl*_invl. • Ensures the flat_load has completed before invalidating the caches. 3. buffer_gl0_inv; buffer_gl1_inv • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale global

				data.
atomicrmw	acquire	• singlethread • wavefront	• global • local • generic	1. buffer/global/ds/flat_atomic
atomicrmw	acquire	• workgroup	• global	1. buffer/global_atomic 2. s_waitcnt vm/vscnt(0) • If CU wavefront execution mode, omit. • Use vmcnt(0) if atomic with return and vscnt(0) if atomic with no-return. • Must happen before the following buffer_gl0_inv and before any following global/generic load/load atomic/store/store atomic/atomicrmw. 3. buffer_gl0_inv • If CU wavefront execution mode, omit. • Ensures that following loads will not see stale data.
atomicrmw	acquire	• workgroup	• local	1. ds_atomic 2. s_waitcnt lgkmcnt(0) • If OpenCL, omit. • Must happen before

				the following buffer_gl0_inv. • Ensures any following global data read is no older than the local atomicrmw value being acquired. 3. buffer_gl0_inv • If OpenCL omit. • Ensures that following loads will not see stale data.
atomicrmw	acquire	• workgroup	• generic	1. flat_atomic 2. s_waitcnt lgkmcnt(0) & vm/vscnt(0) • If CU wavefront execution mode, omit vm/vscnt(0). • If OpenCL, omit lgkmcnt(0). • Use vmcnt(0) if atomic with return and vscnt(0) if atomic with no-return. • Must happen before the following buffer_gl0_inv. • Ensures any following global data read is no older than a local atomicrmw value being acquired.

				3. buffer_gl0_inv • If CU wavefront execution mode, omit. • Ensures that following loads will not see stale data.
atomicrmw	acquire	• agent • system	• global	1. buffer/global_atomic 2. s_waitcnt vm/vscnt(0) • Use vmcnt(0) if atomic with return and vscnt(0) if atomic with no-return. • Must happen before following buffer_gl*_inv. • Ensures the atomicrmw has completed before invalidating the caches. 3. buffer_gl0_inv; buffer_gl1_inv • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale global data.
atomicrmw	acquire	• agent	• generic	1. flat_atomic

		system		2. s_waitcnt vm/vscnt(0) & lgkmcnt(0) • If OpenCL, omit lgkmcnt(0). • Use vmcnt(0) if atomic with return and vscnt(0) if atomic with no-return. • Must happen before following buffer_gl*_inv. • Ensures the atomicrmw has completed before invalidating the caches. 3. buffer_gl0_inv; buffer_gl1_inv • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale global data.
fence	acquire	• singlethread • wavefront	*none*	*none*
fence	acquire	• workgroup	*none*	1. s_waitcnt lgkmcnt(0) & vmcnt(0) & vscnt(0) • If CU wavefront execution mode, omit vmcnt(0) and

vscnt(0).

- If OpenCL and address space is not generic, omit lgkmcnt(0).
- If OpenCL and address space is local, omit vmcnt(0) and vscnt(0).
- However, since LLVM currently has no address space on the fence need to conservatively always generate. If fence had an address space then set to address space of OpenCL fence flag, or to generic if both local and global flags are specified.
- Could be split into separate s_waitcnt vmcnt(0), s_waitcnt vscnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.
- s_waitcnt vmcnt(0) must happen after any preceding global/generic load atomic/

atomicrmw-with-return-value with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic).

- s_waitcnt vscnt(0) must happen after any preceding global/generic atomicrmw-no-return-value with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic).
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic).
- Must happen before the following buffer_gl0_inv.
- Ensures that the fence-paired atomic

				has completed before invalidating the cache. Therefore any following locations read must be no older than the value read by the fence-paired-atomic. 3. buffer_gl0_inv • If CU wavefront execution mode, omit. • Ensures that following loads will not see stale data.
fence	acquire	• agent • system	*none*	1. s_waitcnt lgkmcnt(0) & vmcnt(0) & vscnt(0) • If OpenCL and address space is not generic, omit lgkmcnt(0). • If OpenCL and address space is local, omit vmcnt(0) and vscnt(0). • However, since LLVM currently has no address space on the fence need to conservatively always generate (see comment for previous fence). • Could be split into

separate s_waitcnt vmcnt(0), s_waitcnt vscnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.

- s_waitcnt vmcnt(0) must happen after any preceding global/generic load atomic/ atomicrmw-with-return-value with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic).
- s_waitcnt vscnt(0) must happen after any preceding global/generic atomicrmw-no-return-value with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic).
- s_waitcnt lgkmcnt(0) must happen after any preceding

				local/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic). • Must happen before the following buffer_gl*_inv. • Ensures that the fence-paired atomic has completed before invalidating the caches. Therefore any following locations read must be no older than the value read by the fence-paired-atomic. 2. buffer_gl0_inv; buffer_gl1_inv • Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures that following loads will not see stale global data.
Release Atomic				
store	release	• singlethread •	• global	1. buffer/global/ds/flat_store

atomic		• wavefront	• local • generic	
store atomic	release	• workgroup	• global • generic	1. s_waitcnt lgkmcnt(0) & vmcnt(0) & vscnt(0) • If CU wavefront execution mode, omit vmcnt(0) and vscnt(0). • If OpenCL, omit lgkmcnt(0). • Could be split into separate s_waitcnt vmcnt(0), s_waitcnt vscnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after any preceding global/generic load/load atomic/ atomicrmw-with-return-value. • s_waitcnt vscnt(0) must happen after any preceding global/generic store/store atomic/ atomicrmw-no-return-value. • s_waitcnt lgkmcnt(0) must

				happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw. • Must happen before the following store. • Ensures that all memory operations have completed before performing the store that is being released. 2. buffer/global/flat_store
store atomic	release	• workgroup	• local	1. s_waitcnt vmcnt(0) & vscnt(0) • If CU wavefront execution mode, omit. • If OpenCL, omit. • Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt vscnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after any preceding global/generic load/load atomic/ atomicrmw-with-return-value. • s_waitcnt vscnt(0)

				must happen after any preceding global/generic store/store atomic/ atomicrmw-no-return-value. • Must happen before the following store. • Ensures that all global memory operations have completed before performing the store that is being released. 2. ds_store
store atomic	release	• agent • system	• global • generic	1. s_waitcnt lgkmcnt(0) & vmcnt(0) & vscnt(0) • If OpenCL and address space is not generic, omit lgkmcnt(0). • Could be split into separate s_waitcnt vmcnt(0), s_waitcnt vscnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after any preceding global/generic

				load/load atomic/ atomicrmw-with-return-value. • s_waitcnt vscnt(0) must happen after any preceding global/generic store/store atomic/ atomicrmw-no-return-value. • s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw. • Must happen before the following store. • Ensures that all memory operations have completed before performing the store that is being released. 2. buffer/global/flat_store
atomicrmw	release	• singlethread • wavefront	• global • local • generic	1. buffer/global/ds/flat_atomic
atomicrmw	release	• workgroup	• global • generic	1. s_waitcnt lgkmcnt(0) & vmcnt(0) & vscnt(0) • If CU wavefront execution mode, omit vmcnt(0) and

vscnt(0).
- If OpenCL, omit lgkmcnt(0).
- Could be split into separate s_waitcnt vmcnt(0), s_waitcnt vscnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.
- s_waitcnt vmcnt(0) must happen after any preceding global/generic load/load atomic/ atomicrmw-with-return-value.
- s_waitcnt vscnt(0) must happen after any preceding global/generic store/store atomic/ atomicrmw-no-return-value.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before the following atomicrmw.

				• Ensures that all memory operations have completed before performing the atomicrmw that is being released. 2. buffer/global/flat_atomic
atomicrmw	release	• workgroup	• local	1. s_waitcnt vmcnt(0) & vscnt(0) • If CU wavefront execution mode, omit. • If OpenCL, omit. • Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt vscnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after any preceding global/generic load/load atomic/ atomicrmw-with-return-value. • s_waitcnt vscnt(0) must happen after any preceding global/generic store/store atomic/ atomicrmw-no-return-value. • Must happen before the following store.

				• Ensures that all global memory operations have completed before performing the store that is being released. 2. ds_atomic
atomicrmw	release	• agent • system	• global • generic	1. s_waitcnt lgkmcnt(0) & vmcnt(0) & vscnt(0) • If OpenCL, omit lgkmcnt(0). • Could be split into separate s_waitcnt vmcnt(0), s_waitcnt vscnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after any preceding global/generic load/load atomic/ atomicrmw-with-return-value. • s_waitcnt vscnt(0) must happen after any preceding global/generic store/store atomic/ atomicrmw-no-return-value.

				• s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw. • Must happen before the following atomicrmw. • Ensures that all memory operations to global and local have completed before performing the atomicrmw that is being released. 2. buffer/global/flat_atomic
fence	release	• singlethread • wavefront	*none*	*none*
fence	release	• workgroup	*none*	1. s_waitcnt lgkmcnt(0) & vmcnt(0) & vscnt(0) • If CU wavefront execution mode, omit vmcnt(0) and vscnt(0). • If OpenCL and address space is not generic, omit lgkmcnt(0). • If OpenCL and address space is local, omit vmcnt(0) and vscnt(0).

- However, since LLVM currently has no address space on the fence need to conservatively always generate. If fence had an address space then set to address space of OpenCL fence flag, or to generic if both local and global flags are specified.
- Could be split into separate s_waitcnt vmcnt(0), s_waitcnt vscnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.
- s_waitcnt vmcnt(0) must happen after any preceding global/generic load/load atomic/ atomicrmw-with-return-value.
- s_waitcnt vscnt(0) must happen after any preceding global/generic store/store atomic/ atomicrmw-no-return-value.

				• s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/ atomicrmw. • Must happen before any following store atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic). • Ensures that all memory operations have completed before performing the following fence-paired-atomic.
fence	release	• agent • system	*none*	1. s_waitcnt lgkmcnt(0) & vmcnt(0) & vscnt(0) • If OpenCL and address space is not generic, omit lgkmcnt(0). • If OpenCL and address space is local, omit vmcnt(0) and vscnt(0). • However, since LLVM currently has

no address space on the fence need to conservatively always generate. If fence had an address space then set to address space of OpenCL fence flag, or to generic if both local and global flags are specified.

- Could be split into separate s_waitcnt vmcnt(0), s_waitcnt vscnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.
- s_waitcnt vmcnt(0) must happen after any preceding global/generic load/load atomic/ atomicrmw-with-return-value.
- s_waitcnt vscnt(0) must happen after any preceding global/generic store/store atomic/ atomicrmw-no-return-value.
- s_waitcnt lgkmcnt(0) must

				happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw. • Must happen before any following store atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the fence-paired-atomic). • Ensures that all memory operations have completed before performing the following fence-paired-atomic.
Acquire-Release Atomic				
atomicrmw	acq_rel	• singlethread • wavefront	• global • local • generic	1. buffer/global/ds/flat_atomic
atomicrmw	acq_rel	• workgroup	• global	1. s_waitcnt lgkmcnt(0) & vmcnt(0) & vscnt(0) • If CU wavefront execution mode, omit vmcnt(0) and vscnt(0). • If OpenCL, omit lgkmcnt(0). • Must happen after any preceding

local/generic load/store/load atomic/store atomic/atomicrmw.
- Could be split into separate s_waitcnt vmcnt(0), s_waitcnt vscnt(0), and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.
- s_waitcnt vmcnt(0) must happen after any preceding global/generic load/load atomic/ atomicrmw-with-return-value.
- s_waitcnt vscnt(0) must happen after any preceding global/generic store/store atomic/ atomicrmw-no-return-value.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before the following

				atomicrmw. • Ensures that all memory operations have completed before performing the atomicrmw that is being released. 2. buffer/global_atomic 3. s_waitcnt vm/vscnt(0) • If CU wavefront execution mode, omit. • Use vmcnt(0) if atomic with return and vscnt(0) if atomic with no-return. • Must happen before the following buffer_gl0_inv. • Ensures any following global data read is no older than the atomicrmw value being acquired. 4. buffer_gl0_inv • If CU wavefront execution mode, omit. • Ensures that following loads will not see stale data.
atomicrmw	acq_rel	• workgroup	• local	1. s_waitcnt vmcnt(0) & vscnt(0)

- If CU wavefront execution mode, omit.
- If OpenCL, omit.
- Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt vscnt(0) to allow them to be independently moved according to the following rules.
- s_waitcnt vmcnt(0) must happen after any preceding global/generic load/load atomic/ atomicrmw-with-return-value.
- s_waitcnt vscnt(0) must happen after any preceding global/generic store/store atomic/ atomicrmw-no-return-value.
- Must happen before the following store.
- Ensures that all global memory operations have completed before performing the store that is being released.

2. ds_atomic
3. s_waitcnt lgkmcnt(0)

				• If OpenCL, omit. • Must happen before the following buffer_gl0_inv. • Ensures any following global data read is no older than the local load atomic value being acquired. 4. buffer_gl0_inv • If CU wavefront execution mode, omit. • If OpenCL omit. • Ensures that following loads will not see stale data.
atomicrmw	acq_rel	• workgroup	• generic	1. s_waitcnt lgkmcnt(0) & vmcnt(0) & vscnt(0) • If CU wavefront execution mode, omit vmcnt(0) and vscnt(0). • If OpenCL, omit lgkmcnt(0). • Could be split into separate s_waitcnt vmcnt(0), s_waitcnt vscnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to

the following rules.
- s_waitcnt vmcnt(0) must happen after any preceding global/generic load/load atomic/ atomicrmw-with-return-value.
- s_waitcnt vscnt(0) must happen after any preceding global/generic store/store atomic/ atomicrmw-no-return-value.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before the following atomicrmw.
- Ensures that all memory operations have completed before performing the atomicrmw that is being released.

2. flat_atomic
3. s_waitcnt lgkmcnt(0) & vmcnt(0) & vscnt(0)

    - If CU wavefront execution mode,

				omit vmcnt(0) and vscnt(0). • If OpenCL, omit lgkmcnt(0). • Must happen before the following buffer_gl0_inv. • Ensures any following global data read is no older than the load atomic value being acquired. 3. buffer_gl0_inv • If CU wavefront execution mode, omit. • Ensures that following loads will not see stale data.
atomicrmw	acq_rel	• agent • system	• global	1. s_waitcnt lgkmcnt(0) & vmcnt(0) & vscnt(0) • If OpenCL, omit lgkmcnt(0). • Could be split into separate s_waitcnt vmcnt(0), s_waitcnt vscnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0)

must happen after any preceding global/generic load/load atomic/ atomicrmw-with-return-value.
- s_waitcnt vscnt(0) must happen after any preceding global/generic store/store atomic/ atomicrmw-no-return-value.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before the following atomicrmw.
- Ensures that all memory operations to global have completed before performing the atomicrmw that is being released.

2. buffer/global_atomic
3. s_waitcnt vm/vscnt(0)

    - Use vmcnt(0) if atomic with return and vscnt(0) if atomic with no-

				return. • Must happen before following buffer_gl*_inv. • Ensures the atomicrmw has completed before invalidating the caches. 4. buffer_gl0_inv; buffer_gl1_inv • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale global data.
atomicrmw	acq_rel	• agent • system	• generic	1. s_waitcnt lgkmcnt(0) & vmcnt(0) & vscnt(0) • If OpenCL, omit lgkmcnt(0). • Could be split into separate s_waitcnt vmcnt(0), s_waitcnt vscnt(0), and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after

any preceding global/generic load/load atomic atomicrmw-with-return-value.
- s_waitcnt vscnt(0) must happen after any preceding global/generic store/store atomic/atomicrmw-no-return-value.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before the following atomicrmw.
- Ensures that all memory operations have completed before performing the atomicrmw that is being released.

2. flat_atomic
3. s_waitcnt vm/vscnt(0) & lgkmcnt(0)

    - If OpenCL, omit lgkmcnt(0).
    - Use vmcnt(0) if atomic with return and vscnt(0) if

				atomic with no-return. • Must happen before following buffer_gl*_inv. • Ensures the atomicrmw has completed before invalidating the caches. 4. buffer_gl0_inv; buffer_gl1_inv • Must happen before any following global/generic load/load atomic/atomicrmw. • Ensures that following loads will not see stale global data.
fence	acq_rel	• singlethread • wavefront	*none*	*none*
fence	acq_rel	• workgroup	*none*	1. s_waitcnt lgkmcnt(0) & vmcnt(0) & vscnt(0) • If CU wavefront execution mode, omit vmcnt(0) and vscnt(0). • If OpenCL and address space is not generic, omit lgkmcnt(0). • If OpenCL and address space is

local, omit vmcnt(0) and vscnt(0).
- However, since LLVM currently has no address space on the fence need to conservatively always generate (see comment for previous fence).
- Could be split into separate s_waitcnt vmcnt(0), s_waitcnt vscnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.
- s_waitcnt vmcnt(0) must happen after any preceding global/generic load/load atomic/ atomicrmw-with-return-value.
- s_waitcnt vscnt(0) must happen after any preceding global/generic store/store atomic/ atomicrmw-no-return-value.
- s_waitcnt lgkmcnt(0) must happen after any preceding

local/generic load/store/load atomic/store atomic/ atomicrmw.
- Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw.
- Ensures that all memory operations have completed before performing any following global memory operations.
- Ensures that the preceding local/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the acquire-fence-paired-atomic) has completed before following global memory operations. This satisfies the requirements of acquire.
- Ensures that all previous memory operations have completed before a

following local/generic store atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the release-fence-paired-atomic). This satisfies the requirements of release.
- Must happen before the following buffer_gl0_inv.
- Ensures that the acquire-fence-paired atomic has completed before invalidating the cache. Therefore any following locations read must be no older than the value read by the acquire-fence-paired-atomic.

3. buffer_gl0_inv

    - If CU wavefront execution mode, omit.
    - Ensures that following loads will not see stale data.

fence	acq_rel	• agent • system	*none*	1. s_waitcnt lgkmcnt(0) & vmcnt(0) & vscnt(0) • If OpenCL and address space is not generic, omit lgkmcnt(0). • If OpenCL and address space is local, omit vmcnt(0) and vscnt(0). • However, since LLVM currently has no address space on the fence need to conservatively always generate (see comment for previous fence). • Could be split into separate s_waitcnt vmcnt(0), s_waitcnt vscnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt vmcnt(0) must happen after any preceding global/generic load/load atomic/ atomicrmw-with-return-value. • s_waitcnt vscnt(0) must happen after

any preceding global/generic store/store atomic/atomicrmw-no-return-value.
- s_waitcnt lgkmcnt(0) must happen after any preceding local/generic load/store/load atomic/store atomic/atomicrmw.
- Must happen before the following buffer_gl*_inv.
- Ensures that the preceding global/local/generic load atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the acquire-fence-paired-atomic) has completed before invalidating the caches. This satisfies the requirements of acquire.
- Ensures that all previous memory operations have

				completed before a following global/local/generic store atomic/atomicrmw with an equal or wider sync scope and memory ordering stronger than unordered (this is termed the release-fence-paired-atomic). This satisfies the requirements of release. 2. buffer_gl0_inv; buffer_gl1_inv • Must happen before any following global/generic load/load atomic/store/store atomic/atomicrmw. • Ensures that following loads will not see stale global data. This satisfies the requirements of acquire.
Sequential Consistent Atomic				
load atomic	seq_cst	• singlethread • wavefront	• global • local • generic	*Same as corresponding load atomic acquire, except must generate all instructions even for OpenCL.*
load atomic	seq_cst	• workgroup	• global • generic	1. s_waitcnt lgkmcnt(0) & vmcnt(0) & vscnt(0)

- If CU wavefront execution mode, omit vmcnt(0) and vscnt(0).
- Could be split into separate s_waitcnt vmcnt(0), s_waitcnt vscnt(0), and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules.
- s_waitcnt lgkmcnt(0) must happen after preceding local/generic load atomic/store atomic/atomicrmw with memory ordering of seq_cst and with equal or wider sync scope. (Note that seq_cst fences have their own s_waitcnt lgkmcnt(0) and so do not need to be considered.)
- s_waitcnt vmcnt(0) must happen after preceding global/generic load atomic/ atomicrmw-with-

return-value with memory ordering of seq_cst and with equal or wider sync scope. (Note that seq_cst fences have their own s_waitcnt vmcnt(0) and so do not need to be considered.)

- s_waitcnt vscnt(0) Must happen after preceding global/generic store atomic/ atomicrmw-no-return-value with memory ordering of seq_cst and with equal or wider sync scope. (Note that seq_cst fences have their own s_waitcnt vscnt(0) and so do not need to be considered.)
- Ensures any preceding sequential consistent global/local memory instructions have completed before executing this sequentially consistent instruction. This

				prevents reordering a seq_cst store followed by a seq_cst load. (Note that seq_cst is stronger than acquire/release as the reordering of load acquire followed by a store release is prevented by the s_waitcnt of the release, but there is nothing preventing a store release followed by load acquire from completing out of order. The s_waitcnt could be placed after seq_store or before the seq_load. We choose the load to make the s_waitcnt be as late as possible so that the store may have already completed.) 2. *Following instructions same as corresponding load atomic acquire, except must generate all instructions even for OpenCL.*
load atomic	seq_cst	• workgroup	• local	1. s_waitcnt vmcnt(0) & vscnt(0) • If CU wavefront execution mode,

omit.
- Could be split into separate s_waitcnt vmcnt(0) and s_waitcnt vscnt(0) to allow them to be independently moved according to the following rules.
- s_waitcnt vmcnt(0) Must happen after preceding global/generic load atomic/ atomicrmw-with-return-value with memory ordering of seq_cst and with equal or wider sync scope. (Note that seq_cst fences have their own s_waitcnt vmcnt(0) and so do not need to be considered.)
- s_waitcnt vscnt(0) Must happen after preceding global/generic store atomic/ atomicrmw-no-return-value with memory ordering of seq_cst and with equal or wider sync scope. (Note that seq_cst fences have their own s_waitcnt

vscnt(0) and so do not need to be considered.)

- Ensures any preceding sequential consistent global memory instructions have completed before executing this sequentially consistent instruction. This prevents reordering a seq_cst store followed by a seq_cst load. (Note that seq_cst is stronger than acquire/release as the reordering of load acquire followed by a store release is prevented by the s_waitcnt of the release, but there is nothing preventing a store release followed by load acquire from completing out of order. The s_waitcnt could be placed after seq_store or before the seq_load. We choose the load to make the

				s_waitcnt be as late as possible so that the store may have already completed.) 2. *Following instructions same as corresponding load atomic acquire, except must generate all instructions even for OpenCL.*
load atomic	seq_cst	• agent • system	• global • generic	1. s_waitcnt lgkmcnt(0) & vmcnt(0) & vscnt(0) • Could be split into separate s_waitcnt vmcnt(0), s_waitcnt vscnt(0) and s_waitcnt lgkmcnt(0) to allow them to be independently moved according to the following rules. • s_waitcnt lgkmcnt(0) must happen after preceding local load atomic/store atomic/atomicrmw with memory ordering of seq_cst and with equal or wider sync scope. (Note that seq_cst fences have their own s_waitcnt lgkmcnt(0) and so do not need to be

considered.)

- s_waitcnt vmcnt(0) must happen after preceding global/generic load atomic/ atomicrmw-with-return-value with memory ordering of seq_cst and with equal or wider sync scope. (Note that seq_cst fences have their own s_waitcnt vmcnt(0) and so do not need to be considered.)
- s_waitcnt vscnt(0) Must happen after preceding global/generic store atomic/ atomicrmw-no-return-value with memory ordering of seq_cst and with equal or wider sync scope. (Note that seq_cst fences have their own s_waitcnt vscnt(0) and so do not need to be considered.)
- Ensures any preceding sequential consistent global memory

instructions have completed before executing this sequentially consistent instruction. This prevents reordering a seq_cst store followed by a seq_cst load. (Note that seq_cst is stronger than acquire/release as the reordering of load acquire followed by a store release is prevented by the s_waitcnt of the release, but there is nothing preventing a store release followed by load acquire from completing out of order. The s_waitcnt could be placed after seq_store or before the seq_load. We choose the load to make the s_waitcnt be as late as possible so that the store may have already completed.)

2. *Following instructions same as corresponding load atomic acquire, except must generate*

				all instructions even for OpenCL.
store atomic	seq_cst	• singlethread • wavefront • workgroup • agent • system	• global • local • generic	*Same as corresponding store atomic release, except must generate all instructions even for OpenCL.*
atomicrmw	seq_cst	• singlethread • wavefront • workgroup • agent • system	• global • local • generic	*Same as corresponding atomicrmw acq_rel, except must generate all instructions even for OpenCL.*
fence	seq_cst	• singlethread • wavefront • workgroup • agent • system	*none*	*Same as corresponding fence acq_rel, except must generate all instructions even for OpenCL.*

Trap Handler ABI

For code objects generated by the AMDGPU backend for HSA [HSA] compatible runtimes (see AMDGPU Operating Systems), the runtime installs a trap handler that supports the `s_trap` instruction. For usage see:

- AMDGPU Trap Handler for AMDHSA OS Code Object V2
- AMDGPU Trap Handler for AMDHSA OS Code Object V3
- AMDGPU Trap Handler for AMDHSA OS Code Object V4 and Above

AMDGPU Trap Handler for AMDHSA OS Code Object V2

Usage	Code Sequence	Trap Handler Inputs	Description
reserved	`s_trap 0x00`		Reserved by hardware.
`debugtrap(arg)`	`s_trap 0x01`	`SGPR0-1`: `queue_ptr`	Reserved for Finalizer HSA `debugtrap` intrinsic (not implemented).

		VGPR0: arg	
llvm.trap	s_trap 0x02	SGPR0-1: queue_ptr	Causes wave to be halted with the PC at the trap instruction. The associated queue is signalled to put it into the error state. When the queue is put in the error state, the waves executing dispatches on the queue will be terminated.
llvm.debugtrap	s_trap 0x03	*none*	◦ If debugger not enabled then behaves as a no-operation. The trap handler is entered and immediately returns to continue execution of the wavefront. ◦ If the debugger is enabled, causes the debug trap to be reported by the debugger and the wavefront is put in the halt state with the PC at the instruction. The debugger must increment the PC and resume the wave.
reserved	s_trap 0x04		Reserved.
reserved	s_trap 0x05		Reserved.
reserved	s_trap 0x06		Reserved.
reserved	s_trap 0x07		Reserved.
reserved	s_trap 0x08		Reserved.
reserved	s_trap 0xfe		Reserved.
reserved	s_trap 0xff		Reserved.

AMDGPU Trap Handler for AMDHSA OS Code Object V3

Usage	Code Sequence	Trap Handler Inputs	Description
reserved	s_trap 0x00		Reserved by hardware.
debugger breakpoint	s_trap 0x01	*none*	Reserved for debugger to use for breakpoints. Causes wave to be halted with the PC at the trap instruction. The debugger is

			responsible to resume the wave, including the instruction that the breakpoint overwrote.
`llvm.trap`	`s_trap 0x02`	`SGPR0-1`: `queue_ptr`	Causes wave to be halted with the PC at the trap instruction. The associated queue is signalled to put it into the error state. When the queue is put in the error state, the waves executing dispatches on the queue will be terminated.
`llvm.debugtrap`	`s_trap 0x03`	*none*	• If debugger not enabled then behaves as a no-operation. The trap handler is entered and immediately returns to continue execution of the wavefront. • If the debugger is enabled, causes the debug trap to be reported by the debugger and the wavefront is put in the halt state with the PC at the instruction. The debugger must increment the PC and resume the wave.
reserved	`s_trap 0x04`		Reserved.
reserved	`s_trap 0x05`		Reserved.
reserved	`s_trap 0x06`		Reserved.
reserved	`s_trap 0x07`		Reserved.
reserved	`s_trap 0x08`		Reserved.
reserved	`s_trap 0xfe`		Reserved.
reserved	`s_trap 0xff`		Reserved.

AMDGPU Trap Handler for AMDHSA OS Code Object V4 and Above

Usage	Code Sequence	GFX6-GFX8 Inputs	GFX9-GFX11 Inputs	Description

reserved	`s_trap 0x00`			Reserved by hardware.
debugger breakpoint	`s_trap 0x01`	*none*	*none*	Reserved for debugger to use for breakpoints. Causes wave to be halted with the PC at the trap instruction. The debugger is responsible to resume the wave, including the instruction that the breakpoint overwrote.
`llvm.trap`	`s_trap 0x02`	`SGPR0-1: queue_ptr`	*none*	Causes wave to be halted with the PC at the trap instruction. The associated queue is signalled to put it into the error state. When the queue is put in the error state, the waves executing dispatches on the queue will be terminated.
`llvm.debugtrap`	`s_trap 0x03`	*none*	*none*	• If debugger not enabled then behaves as a no-operation. The trap handler is entered and immediately returns to continue execution of the wavefront. • If the debugger is enabled, causes the debug trap to be reported by the debugger and the wavefront is put in the halt state with the PC at the instruction. The debugger must increment the PC and resume the wave.

reserved	`s_trap 0x04`	Reserved.
reserved	`s_trap 0x05`	Reserved.
reserved	`s_trap 0x06`	Reserved.
reserved	`s_trap 0x07`	Reserved.
reserved	`s_trap 0x08`	Reserved.
reserved	`s_trap 0xfe`	Reserved.
reserved	`s_trap 0xff`	Reserved.

Call Convention

Note

This section is currently incomplete and has inaccuracies. It is WIP that will be updated as information is determined.

See Address Space Identifier for information on swizzled addresses. Unswizzled addresses are normal linear addresses.

Kernel Functions

This section describes the call convention ABI for the outer kernel function.

See Initial Kernel Execution State for the kernel call convention.

The following is not part of the AMDGPU kernel calling convention but describes how the AMDGPU implements function calls:

1. Clang decides the kernarg layout to match the *HSA Programmer's Language Reference* [HSA].
   - All structs are passed directly.
   - Lambda values are passed *TBA*.

4. The kernel performs certain setup in its prolog, as described in Kernel Prolog.

Non-Kernel Functions

This section describes the call convention ABI for functions other than the outer kernel function.

If a kernel has function calls then scratch is always allocated and used for the call stack which grows from low address to high address using the swizzled scratch address space.

On entry to a function:

1. SGPR0-3 contain a V# with the following properties (see Private Segment Buffer):
    - Base address pointing to the beginning of the wavefront scratch backing memory.
    - Swizzled with dword element size and stride of wavefront size elements.
2. The FLAT_SCRATCH register pair is setup. See Flat Scratch.
3. GFX6-GFX8: M0 register set to the size of LDS in bytes. See M0.
4. The EXEC register is set to the lanes active on entry to the function.
5. MODE register: *TBD*
6. VGPR0-31 and SGPR4-29 are used to pass function input arguments as described below.
7. SGPR30-31 return address (RA). The code address that the function must return to when it completes. The value is undefined if the function is *no return*.
8. SGPR32 is used for the stack pointer (SP). It is an unswizzled scratch offset relative to the beginning of the wavefront scratch backing memory.

   The unswizzled SP can be used with buffer instructions as an unswizzled SGPR offset with the scratch V# in SGPR0-3 to access the stack in a swizzled manner.

   The unswizzled SP value can be converted into the swizzled SP value by:

   swizzled SP = unswizzled SP / wavefront size

   This may be used to obtain the private address space address of stack objects and to convert this address to a flat address by adding the flat scratch aperture base address.

   The swizzled SP value is always 4 bytes aligned for the `r600` architecture and 16 byte aligned for the `amdgcn` architecture.

   Note

   The `amdgcn` value is selected to avoid dynamic stack alignment for the OpenCL language which has the largest base type defined as 16 bytes.

On entry, the swizzled SP value is the address of the first function argument passed on the stack. Other stack passed arguments are positive offsets from the entry swizzled SP value.

The function may use positive offsets beyond the last stack passed argument for stack allocated local variables and register spill slots. If necessary, the function may align these to greater alignment than 16 bytes. After these the function may dynamically allocate space for such things as runtime sized `alloca` local allocations.

If the function calls another function, it will place any stack allocated arguments after the last local allocation and adjust SGPR32 to the address after the last local allocation.

9. All other registers are unspecified.
10. Any necessary `s_waitcnt` has been performed to ensure memory is available to the function.

On exit from a function:

1. VGPR0-31 and SGPR4-29 are used to pass function result arguments as described below. Any registers used are considered clobbered registers.
2. The following registers are preserved and have the same value as on entry:
    - FLAT_SCRATCH
    - EXEC
    - GFX6-GFX8: M0
    - All SGPR registers except the clobbered registers of SGPR4-31.
    - VGPR40-47
    - VGPR56-63
    - VGPR72-79
    - VGPR88-95
    - VGPR104-111
    - VGPR120-127
    - VGPR136-143
    - VGPR152-159
    - VGPR168-175
    - VGPR184-191
    - VGPR200-207

- VGPR216-223
- VGPR232-239
- VGPR248-255

> **Note**
>
> Except the argument registers, the VGPRs clobbered and the preserved registers are intermixed at regular intervals in order to keep a similar ratio independent of the number of allocated VGPRs.

- GFX90A: All AGPR registers except the clobbered registers AGPR0-31.
- Lanes of all VGPRs that are inactive at the call site.

  For the AMDGPU backend, an inter-procedural register allocation (IPRA) optimization may mark some of clobbered SGPR and VGPR registers as preserved if it can be determined that the called function does not change their value.

2. The PC is set to the RA provided on entry.
3. MODE register: *TBD*.
4. All other registers are clobbered.
5. Any necessary `s_waitcnt` has been performed to ensure memory accessed by function is available to the caller.

The function input arguments are made up of the formal arguments explicitly declared by the source language function plus the implicit input arguments used by the implementation.

The source language input arguments are:

1. Any source language implicit `this` or `self` argument comes first as a pointer type.
2. Followed by the function formal arguments in left to right source order.

The source language result arguments are:

1. The function result argument.

The source language input or result struct type arguments that are less than or equal to 16 bytes, are decomposed recursively into their base type fields, and each field is passed as if a separate argument. For input arguments, if the called function requires the struct to be in memory, for example because its address is taken, then the function body is responsible for allocating a stack location and copying the field arguments into it. Clang terms this *direct struct*.

The source language input struct type arguments that are greater than 16 bytes, are passed by reference. The caller is responsible for allocating a stack location to make a copy of the struct value and pass the address as the input argument. The called function is responsible to perform the dereference when accessing the input argument. Clang terms this *by-value struct*.

A source language result struct type argument that is greater than 16 bytes, is returned by reference. The caller is responsible for allocating a stack location to hold the result value and passes the address as the last input argument (before the implicit input arguments). In this case there are no result arguments. The called function is responsible to perform the dereference when storing the result value. Clang terms this *structured return (sret)*.

TODO: correct the ``sret`` definition.

Lambda argument types are treated as struct types with an implementation defined set of fields.

For AMDGPU backend all source language arguments (including the decomposed struct type arguments) are passed in VGPRs unless marked `inreg` in which case they are passed in SGPRs.

The AMDGPU backend walks the function call graph from the leaves to determine which implicit input arguments are used, propagating to each caller of the function. The used implicit arguments are appended to the function arguments after the source language arguments in the following order:

1. Work-Item ID (1 VGPR)

   The X, Y and Z work-item ID are packed into a single VGRP with the following layout. Only fields actually used by the function are set. The other bits are undefined.

   The values come from the initial kernel execution state. See Initial Kernel Execution State.

   Work-item implicit argument layout

Bits	Size	Field Name
9:0	10 bits	X Work-Item ID
19:10	10 bits	Y Work-Item ID
29:20	10 bits	Z Work-Item ID
31:30	2 bits	Unused

2. Dispatch Ptr (2 SGPRs)

   The value comes from the initial kernel execution state. See SGPR Register Set Up Order.

3. Queue Ptr (2 SGPRs)

The value comes from the initial kernel execution state. See SGPR Register Set Up Order.

4. Kernarg Segment Ptr (2 SGPRs)

   The value comes from the initial kernel execution state. See SGPR Register Set Up Order.

5. Dispatch id (2 SGPRs)

   The value comes from the initial kernel execution state. See SGPR Register Set Up Order.

6. Work-Group ID X (1 SGPR)

   The value comes from the initial kernel execution state. See SGPR Register Set Up Order.

7. Work-Group ID Y (1 SGPR)

   The value comes from the initial kernel execution state. See SGPR Register Set Up Order.

8. Work-Group ID Z (1 SGPR)

   The value comes from the initial kernel execution state. See SGPR Register Set Up Order.

9. Implicit Argument Ptr (2 SGPRs)

   The value is computed by adding an offset to Kernarg Segment Ptr to get the global address space pointer to the first kernarg implicit argument.

The input and result arguments are assigned in order in the following manner:

Note
There are likely some errors and omissions in the following description that need correction.

- VGPR arguments are assigned to consecutive VGPRs starting at VGPR0 up to VGPR31.

  If there are more arguments than will fit in these registers, the remaining arguments are allocated on the stack in order on naturally aligned addresses.

- SGPR arguments are assigned to consecutive SGPRs starting at SGPR0 up to SGPR29.

  If there are more arguments than will fit in these registers, the remaining arguments are allocated on the stack in order on naturally aligned addresses.

Note that decomposed struct type arguments may have some fields passed in registers and some in memory.

The following is not part of the AMDGPU function calling convention but describes how the AMDGPU implements function calls:

1. SGPR33 is used as a frame pointer (FP) if necessary. Like the SP it is an unswizzled scratch address. It is only needed if runtime sized `alloca` are used, or for the reasons defined in `SIFrameLowering`.
2. Runtime stack alignment is supported. SGPR34 is used as a base pointer (BP) to access the incoming stack arguments in the function. The BP is needed only when the function requires the runtime stack alignment.
3. Allocating SGPR arguments on the stack are not supported.
4. No CFI is currently generated. See A.6.4 Call Frame Information.

   Note

   CFI will be generated that defines the CFA as the unswizzled address relative to the wave scratch base in the unswizzled private address space of the lowest address stack allocated local variable.

   `DW_AT_frame_base` will be defined as the swizzled address in the swizzled private address space by dividing the CFA by the wavefront size (since CFA is always at least dword aligned which matches the scratch swizzle element size).

   If no dynamic stack alignment was performed, the stack allocated arguments are accessed as negative offsets relative to `DW_AT_frame_base`, and the local variables and register spill slots are accessed as positive offsets relative to `DW_AT_frame_base`.

5. Function argument passing is implemented by copying the input physical registers to virtual registers on entry. The register allocator can spill if necessary. These are copied back to physical registers at call sites. The net effect is that each function call can have these values in entirely distinct locations. The IPRA can help avoid shuffling argument registers.
6. Call sites are implemented by setting up the arguments at positive offsets from SP. Then SP is incremented to account for the known frame size before the call and decremented after the call.

   Note

   The CFI will reflect the changed calculation needed to compute the CFA from SP.

7. 4 byte spill slots are used in the stack frame. One slot is allocated for an emergency spill slot. Buffer instructions are used for stack accesses and not the `flat_scratch` instruction.

AMDPAL

This section provides code conventions used when the target triple OS is `amdpal` (see Target Triples).

Code Object Metadata

> **Note**
>
> The metadata is currently in development and is subject to major changes. Only the current version is supported. *When this document was generated the version was 2.6.*

Code object metadata is specified by the `NT_AMDGPU_METADATA` note record (see Code Object V3 and Above Note Records).

The metadata is represented as Message Pack formatted binary data (see [MsgPack]). The top level is a Message Pack map that includes the keys defined in table AMDPAL Code Object Metadata Map and referenced tables.

Additional information can be added to the maps. To avoid conflicts, any key names should be prefixed by "*vendor-name*." where `vendor-name` can be the name of the vendor and specific vendor tool that generates the information. The prefix is abbreviated to simply "." when it appears within a map that has been added by the same *vendor-name*.

AMDPAL Code Object Metadata Map

String Key	Value Type	Required?	Description
"amdpal.version"	sequence of 2 integers	Required	PAL code object metadata (major, minor) version. The current values are defined by *Util::Abi::PipelineMetadata(Major|Minor)Version*.
"amdpal.pipelines"	sequence of map	Required	Per-pipeline metadata. See AMDPAL Code Object Pipeline Metadata Map for the definition of the keys included in that map.

AMDPAL Code Object Pipeline Metadata Map

String Key	Value Type	Required?	Description
".name"	string		Source name of the pipeline.
".type"	string		Pipeline type, e.g. VsPs. Values include: • "VsPs"

			• “Gs” • “Cs” • “Ngg” • “Tess” • “GsTess” • “NggTess”
“.internal_pipeline_hash”	sequence of 2 integers	Required	Internal compiler hash for this pipeline. Lower 64 bits is the “stable” portion of the hash, used for e.g. shader replacement lookup. Upper 64 bits is the “unique” portion of the hash, used for e.g. pipeline cache lookup. The value is implementation defined, and can not be relied on between different builds of the compiler.
“.shaders”	map		Per-API shader metadata. See AMDPAL Code Object Shader Map for the definition of the keys included in that map.
“.hardware_stages”	map		Per-hardware stage metadata. See AMDPAL Code Object Hardware Stage Map for the definition of the keys included in that map.
“.shader_functions”	map		Per-shader function metadata. See AMDPAL Code Object Shader Function Map for the definition of the keys included in that map.
“.registers”	map	Required	Hardware register configuration. See AMDPAL Code Object Register Map for the definition of the keys included in that map.

".user_data_limit"	integer	Number of user data entries accessed by this pipeline.
".spill_threshold"	integer	The user data spill threshold. 0xFFFF for NoUserDataSpilling.
".uses_viewport_array_index"	boolean	Indicates whether or not the pipeline uses the viewport array index feature. Pipelines which use this feature can render into all 16 viewports, whereas pipelines which do not use it are restricted to viewport #0.
".es_gs_lds_size"	integer	Size in bytes of LDS space used internally for handling data-passing between the ES and GS shader stages. This can be zero if the data is passed using off-chip buffers. This value should be used to program all user-SGPRs which have been marked with "UserDataMapping::EsGsLdsSize" (typically only the GS and VS HW stages will ever have a user-SGPR so marked).
".nggSubgroupSize"	integer	Explicit maximum subgroup size for NGG shaders (maximum number of threads in a subgroup).
".num_interpolants"	integer	Graphics only. Number of PS interpolants.
".mesh_scratch_memory_size"	integer	Max mesh shader scratch memory used.
".api"	string	Name of the client graphics API.
".api_create_info"	binary	Graphics API shader create info binary blob. Can be defined by the driver using the compiler if they want to be able to correlate

	API-specific information used during creation at a later time.

AMDPAL Code Object Shader Map

String Key	Value Type	Description
• ".compute" • ".vertex" • ".hull" • ".domain" • ".geometry" • ".pixel"	map	See AMDPAL Code Object API Shader Metadata Map for the definition of the keys included in that map.

AMDPAL Code Object API Shader Metadata Map

String Key	Value Type	Required?	Description
".api_shader_hash"	sequence of 2 integers	Required	Input shader hash, typically passed in from the client. The value is implementation defined, and can not be relied on between different builds of the compiler.
".hardware_mapping"	sequence of string	Required	Flags indicating the HW stages this API shader maps to. Values include: • ".ls" • ".hs" • ".es" • ".gs" • ".vs" • ".ps" • ".cs"

AMDPAL Code Object Hardware Stage Map

String Key	Value Type	Description
• ".ls" • ".hs"	map	See AMDPAL Code Object Hardware Stage Metadata Map for the definition of the keys included in that map.

- ".es"
- ".gs"
- ".vs"
- ".ps"
- ".cs"

AMDPAL Code Object Hardware Stage Metadata Map

String Key	Value Type	Required?	Description
".entry_point"	string		The ELF symbol pointing to this pipeline's stage entry point.
".scratch_memory_size"	integer		Scratch memory size in bytes.
".lds_size"	integer		Local Data Share size in bytes.
".perf_data_buffer_size"	integer		Performance data buffer size in bytes.
".vgpr_count"	integer		Number of VGPRs used.
".agpr_count"	integer		Number of AGPRs used.
".sgpr_count"	integer		Number of SGPRs used.
".vgpr_limit"	integer		If non-zero, indicates the shader was compiled with a directive to instruct the compiler to limit the VGPR usage to be less than or equal to the specified value (only set if different from HW default).
".sgpr_limit"	integer		SGPR count upper limit (only set if different from HW default).
".threadgroup_dimensions"	sequence of 3 integers		Thread-group X/Y/Z dimensions (Compute only).
".wavefront_size"	integer		Wavefront size (only set if different from HW default).
".uses_uavs"	boolean		The shader reads or writes UAVs.
".uses_rovs"	boolean		The shader reads or writes ROVs.
".writes_uavs"	boolean		The shader writes to one or more UAVs.

".writes_depth"	boolean	The shader writes out a depth value.
".uses_append_consume"	boolean	The shader uses append and/or consume operations, either memory or GDS.
".uses_prim_id"	boolean	The shader uses PrimID.

AMDPAL Code Object Shader Function Map

String Key	Value Type	Description
symbol name	map	*symbol name* is the ELF symbol name of the shader function code entry address. The value is the function's metadata. See AMDPAL Code Object Shader Function Metadata Map.

AMDPAL Code Object Shader Function Metadata Map

String Key	Value Type	Description
".api_shader_hash"	sequence of 2 integers	Input shader hash, typically passed in from the client. The value is implementation defined, and can not be relied on between different builds of the compiler.
".scratch_memory_size"	integer	Size in bytes of scratch memory used by the shader.
".lds_size"	integer	Size in bytes of LDS memory.
".vgpr_count"	integer	Number of VGPRs used by the shader.
".sgpr_count"	integer	Number of SGPRs used by the shader.
".stack_frame_size_in_bytes"	integer	Amount of stack size used by the shader.
".shader_subtype"	string	Shader subtype/kind. Values include: • "Unknown"

AMDPAL Code Object Register Map

32-bit Integer Key	Value Type	Description
`reg offset`	32-bit integer	`reg offset` is the dword offset into the GFXIP register space of a GRBM register (i.e., driver accessible GPU register number, not shader GPR register number).

	The driver is required to program each specified register to the corresponding specified value when executing this pipeline. Typically, the `reg offsets` are the `uint16_t` offsets to each register as defined by the hardware chip headers. The register is set to the provided value. However, a `reg offset` that specifies a user data register (e.g., COMPUTE_USER_DATA_0) needs special treatment. See User Data section for more information.

User Data

Each hardware stage has a set of 32-bit physical SPI *user data registers* (either 16 or 32 based on graphics IP and the stage) which can be written from a command buffer and then loaded into SGPRs when waves are launched via a subsequent dispatch or draw operation. This is the way most arguments are passed from the application/runtime to a hardware shader.

PAL abstracts this functionality by exposing a set of 128 *user data entries* per pipeline a client can use to pass arguments from a command buffer to one or more shaders in that pipeline. The ELF code object must specify a mapping from virtualized *user data entries* to physical *user data registers*, and PAL is responsible for implementing that mapping, including spilling overflow *user data entries* to memory if needed.

Since the *user data registers* are GRBM-accessible SPI registers, this mapping is actually embedded in the `.registers` metadata entry. For most registers, the value in that map is a literal 32-bit value that should be written to the register by the driver. However, when the register is a *user data register* (any USER_DATA register e.g., SPI_SHADER_USER_DATA_PS_5), the value is instead an encoding that tells the driver to write either a *user data entry* value or one of several driver-internal values to the register. This encoding is described in the following table:

Note

Currently, *user data registers* 0 and 1 (e.g., SPI_SHADER_USER_DATA_PS_0, and SPI_SHADER_USER_DATA_PS_1) are reserved. *User data register* 0 must always be programmed to the address of the GlobalTable, and *user data register* 1 must always be programmed to the address of the PerShaderTable.

AMDPAL User Data Mapping

Value	Name	Description
0..127	*User Data Entry*	32-bit value of user_data_entry[N] as specified via

		CmdSetUserData()
0x10000000	GlobalTable	32-bit pointer to GPU memory containing the global internal table (should always point to *user data register* 0).
0x10000001	PerShaderTable	32-bit pointer to GPU memory containing the per-shader internal table. See Per-Shader Table for more detail (should always point to *user data register* 1).
0x10000002	SpillTable	32-bit pointer to GPU memory containing the user data spill table. See Spill Table for more detail.
0x10000003	BaseVertex	Vertex offset (32-bit unsigned integer). Not needed if the pipeline doesn't reference the draw index in the vertex shader. Only supported by the first stage in a graphics pipeline.
0x10000004	BaseInstance	Instance offset (32-bit unsigned integer). Only supported by the first stage in a graphics pipeline.
0x10000005	DrawIndex	Draw index (32-bit unsigned integer). Only supported by the first stage in a graphics pipeline.
0x10000006	Workgroup	Thread group count (32-bit unsigned integer). Low half of a 64-bit address of a buffer containing the grid dimensions for a Compute dispatch operation. The high half of the address is stored in the next sequential user-SGPR. Only supported by compute pipelines.
0x1000000A	EsGsLdsSize	Indicates that PAL will program this user-SGPR to contain the amount of LDS space used for the ES/GS pseudo-ring-buffer for passing data between shader stages.
0x1000000B	ViewId	View id (32-bit unsigned integer) identifies a view of graphic pipeline instancing.
0x1000000C	StreamOutTable	32-bit pointer to GPU memory containing the stream out target SRD table. This can only appear for one shader stage per pipeline.
0x1000000D	PerShaderPerfData	32-bit pointer to GPU memory containing the per-shader performance data buffer.
0x1000000F	VertexBufferTable	32-bit pointer to GPU memory containing the vertex

		buffer SRD table. This can only appear for one shader stage per pipeline.
0x10000010	UavExportTable	32-bit pointer to GPU memory containing the UAV export SRD table. This can only appear for one shader stage per pipeline (PS). These replace color targets and are completely separate from any UAVs used by the shader. This is optional, and only used by the PS when UAV exports are used to replace color-target exports to optimize specific shaders.
0x10000011	NggCullingData	64-bit pointer to GPU memory containing the hardware register data needed by some NGG pipelines to perform culling. This value contains the address of the first of two consecutive registers which provide the full GPU address.
0x10000015	FetchShaderPtr	64-bit pointer to GPU memory containing the fetch shader subroutine.

Per-Shader Table

Low 32 bits of the GPU address for an optional buffer in the `.data` section of the ELF. The high 32 bits of the address match the high 32 bits of the shader's program counter.

The buffer can be anything the shader compiler needs it for, and allows each shader to have its own region of the `.data` section. Typically, this could be a table of buffer SRD's and the data pointed to by the buffer SRD's, but it could be a flat-address region of memory as well. Its layout and usage are defined by the shader compiler.

Each shader's table in the `.data` section is referenced by the symbol `_amdgpu_`*`xs`*`_shdr_intrl_data` where *xs* corresponds with the hardware shader stage the data is for. E.g., `_amdgpu_cs_shdr_intrl_data` for the compute shader hardware stage.

Spill Table

It is possible for a hardware shader to need access to more *user data entries* than there are slots available in user data registers for one or more hardware shader stages. In that case, the PAL runtime expects the necessary *user data entries* to be spilled to GPU memory and use one user data register to point to the spilled user data memory. The value of the *user data entry* must then represent the location where a shader expects to read the low 32-bits of the table's GPU virtual address. The *spill table* itself represents a set of 32-bit values managed by the PAL runtime in

GPU-accessible memory that can be made indirectly accessible to a hardware shader.

Unspecified OS

This section provides code conventions used when the target triple OS is empty (see Target Triples).

Trap Handler ABI

For code objects generated by AMDGPU backend for non-amdhsa OS, the runtime does not install a trap handler. The `llvm.trap` and `llvm.debugtrap` instructions are handled as follows:

AMDGPU Trap Handler for Non-AMDHSA OS

Usage	Code Sequence	Description
llvm.trap	s_endpgm	Causes wavefront to be terminated.
llvm.debugtrap	*none*	Compiler warning given that there is no trap handler installed.

Source Languages

OpenCL

When the language is OpenCL the following differences occur:

1. The OpenCL memory model is used (see Memory Model).
2. The AMDGPU backend appends additional arguments to the kernel's explicit arguments for the AMDHSA OS (see OpenCL kernel implicit arguments appended for AMDHSA OS).
3. Additional metadata is generated (see Code Object Metadata).

OpenCL kernel implicit arguments appended for AMDHSA OS

Position	Byte Size	Byte Alignment	Description
1	8	8	OpenCL Global Offset X
2	8	8	OpenCL Global Offset Y
3	8	8	OpenCL Global Offset Z
4	8	8	OpenCL address of printf buffer
5	8	8	OpenCL address of virtual queue used by enqueue_kernel.

6	8	8	OpenCL address of AqlWrap struct used by enqueue_kernel.
7	8	8	Pointer argument used for Multi-gird synchronization.

HCC

When the language is HCC the following differences occur:

1. The HSA memory model is used (see Memory Model).

Assembler

AMDGPU backend has LLVM-MC based assembler which is currently in development. It supports AMDGCN GFX6-GFX11.

This section describes general syntax for instructions and operands.

Instructions

An instruction has the following syntax:

<opcode> <operand0>, <operand1>,... <modifier0> <modifier1>...

Operands are comma-separated while modifiers are space-separated.

The order of operands and modifiers is fixed. Most modifiers are optional and may be omitted.

Links to detailed instruction syntax description may be found in the following table. Note that features under development are not included in this description.

Architecture	Core ISA	ISA Variants and Extensions
GCN 2	GFX7	-
GCN 3, GCN 4	GFX8	-
GCN 5	GFX9	gfx900 gfx902 gfx904 gfx906

		gfx909
		gfx90c
CDNA 1	GFX9	gfx908
CDNA 2	GFX9	gfx90a
CDNA 3	GFX9	gfx940
RDNA 1	GFX10 RDNA1	gfx1010
		gfx1011
		gfx1012
		gfx1013
RDNA 2	GFX10 RDNA2	gfx1030
		gfx1031
		gfx1032
		gfx1033
		gfx1034
		gfx1035
		gfx1036
RDNA 3	GFX11	gfx1100
		gfx1101
		gfx1102
		gfx1103

For more information about instructions, their semantics and supported combinations of operands, refer to one of instruction set architecture manuals [AMD-GCN-GFX6], [AMD-GCN-GFX7], [AMD-GCN-GFX8], [AMD-GCN-GFX900-GFX904-VEGA], [AMD-GCN-GFX906-VEGA7NM], [AMD-GCN-GFX908-CDNA1], [AMD-GCN-GFX90A-CDNA2], [AMD-GCN-GFX10-RDNA1], [AMD-GCN-GFX10-RDNA2] and [AMD-GCN-GFX11-RDNA3].

Operands

Detailed description of operands may be found here.

Modifiers

Detailed description of modifiers may be found here.

Instruction Examples

DS

```
ds_add_u32 v2, v4 offset:16
ds_write_src2_b64 v2 offset0:4 offset1:8
ds_cmpst_f32 v2, v4, v6
ds_min_rtn_f64 v[8:9], v2, v[4:5]
```

For full list of supported instructions, refer to “LDS/GDS instructions” in ISA Manual.

FLAT

```
flat_load_dword v1, v[3:4]
flat_store_dwordx3 v[3:4], v[5:7]
flat_atomic_swap v1, v[3:4], v5 glc
flat_atomic_cmpswap v1, v[3:4], v[5:6] glc slc
flat_atomic_fmax_x2 v[1:2], v[3:4], v[5:6] glc
```

For full list of supported instructions, refer to “FLAT instructions” in ISA Manual.

MUBUF

```
buffer_load_dword v1, off, s[4:7], s1
buffer_store_dwordx4 v[1:4], v2, ttmp[4:7], s1 offen offset:4 glc tfe
buffer_store_format_xy v[1:2], off, s[4:7], s1
buffer_wbinvl1
buffer_atomic_inc v1, v2, s[8:11], s4 idxen offset:4 slc
```

For full list of supported instructions, refer to “MUBUF Instructions” in ISA Manual.

SMRD/SMEM

```
s_load_dword s1, s[2:3], 0xfc
s_load_dwordx8 s[8:15], s[2:3], s4
```

```
s_load_dwordx16 s[88:103], s[2:3], s4
s_dcache_inv_vol
s_memtime s[4:5]
```

For full list of supported instructions, refer to “Scalar Memory Operations” in ISA Manual.

SOP1

```
s_mov_b32 s1, s2
s_mov_b64 s[0:1], 0x80000000
s_cmov_b32 s1, 200
s_wqm_b64 s[2:3], s[4:5]
s_bcnt0_i32_b64 s1, s[2:3]
s_swappc_b64 s[2:3], s[4:5]
s_cbranch_join s[4:5]
```

For full list of supported instructions, refer to “SOP1 Instructions” in ISA Manual.

SOP2

```
s_add_u32 s1, s2, s3
s_and_b64 s[2:3], s[4:5], s[6:7]
s_cselect_b32 s1, s2, s3
s_andn2_b32 s2, s4, s6
s_lshr_b64 s[2:3], s[4:5], s6
s_ashr_i32 s2, s4, s6
s_bfm_b64 s[2:3], s4, s6
s_bfe_i64 s[2:3], s[4:5], s6
s_cbranch_g_fork s[4:5], s[6:7]
```

For full list of supported instructions, refer to “SOP2 Instructions” in ISA Manual.

SOPC

```
s_cmp_eq_i32 s1, s2
s_bitcmp1_b32 s1, s2
s_bitcmp0_b64 s[2:3], s4
s_setvskip s3, s5
```

For full list of supported instructions, refer to “SOPC Instructions” in ISA Manual.

SOPP

```
s_barrier
s_nop 2
s_endpgm
s_waitcnt 0 ; Wait for all counters to be 0
s_waitcnt vmcnt(0) & expcnt(0) & lgkmcnt(0) ; Equivalent to above
s_waitcnt vmcnt(1) ; Wait for vmcnt counter to be 1.
s_sethalt 9
s_sleep 10
s_sendmsg 0x1
s_sendmsg sendmsg(MSG_INTERRUPT)
s_trap 1
```

For full list of supported instructions, refer to “SOPP Instructions” in ISA Manual.

Unless otherwise mentioned, little verification is performed on the operands of SOPP Instructions, so it is up to the programmer to be familiar with the range or acceptable values.

VALU

For vector ALU instruction opcodes (VOP1, VOP2, VOP3, VOPC, VOP_DPP, VOP_SDWA), the assembler will automatically use optimal encoding based on its operands. To force specific encoding, one can add a suffix to the opcode of the instruction:

- _e32 for 32-bit VOP1/VOP2/VOPC
- _e64 for 64-bit VOP3
- _dpp for VOP_DPP
- _sdwa for VOP_SDWA

VOP1/VOP2/VOP3/VOPC examples:

```
v_mov_b32 v1, v2
v_mov_b32_e32 v1, v2
v_nop
v_cvt_f64_i32_e32 v[1:2], v2
v_floor_f32_e32 v1, v2
v_bfrev_b32_e32 v1, v2
v_add_f32_e32 v1, v2, v3
v_mul_i32_i24_e64 v1, v2, 3
v_mul_i32_i24_e32 v1, -3, v3
v_mul_i32_i24_e32 v1, -100, v3
v_addc_u32 v1, s[0:1], v2, v3, s[2:3]
v_max_f16_e32 v1, v2, v3
```

VOP_DPP examples:

```
v_mov_b32 v0, v0 quad_perm:[0,2,1,1]
v_sin_f32 v0, v0 row_shl:1 row_mask:0xa bank_mask:0x1 bound_ctrl:0
v_mov_b32 v0, v0 wave_shl:1
v_mov_b32 v0, v0 row_mirror
v_mov_b32 v0, v0 row_bcast:31
v_mov_b32 v0, v0 quad_perm:[1,3,0,1] row_mask:0xa bank_mask:0x1 bound_ctrl:0
v_add_f32 v0, v0, |v0| row_shl:1 row_mask:0xa bank_mask:0x1 bound_ctrl:0
v_max_f16 v1, v2, v3 row_shl:1 row_mask:0xa bank_mask:0x1 bound_ctrl:0
```

VOP_SDWA examples:

```
v_mov_b32 v1, v2 dst_sel:BYTE_0 dst_unused:UNUSED_PRESERVE src0_sel:DWORD
v_min_u32 v200, v200, v1 dst_sel:WORD_1 dst_unused:UNUSED_PAD src0_sel:BYTE_1 src1_sel:DWORD
v_sin_f32 v0, v0 dst_unused:UNUSED_PAD src0_sel:WORD_1
v_fract_f32 v0, |v0| dst_sel:DWORD dst_unused:UNUSED_PAD src0_sel:WORD_1
v_cmpx_le_u32 vcc, v1, v2 src0_sel:BYTE_2 src1_sel:WORD_0
```

For full list of supported instructions, refer to “Vector ALU instructions”.

Code Object V2 Predefined Symbols

Warning

Code object V2 is not the default code object version emitted by this version of LLVM.

The AMDGPU assembler defines and updates some symbols automatically. These symbols do not affect code generation.

.option.machine_version_major

Set to the GFX major generation number of the target being assembled for. For example, when assembling for a “GFX9” target this will be set to the integer value “9”. The possible GFX major generation numbers are presented in Processors.

.option.machine_version_minor

Set to the GFX minor generation number of the target being assembled for. For example, when assembling for a “GFX810” target this will be set to the integer value “1”. The possible GFX minor generation numbers are presented in Processors.

.option.machine_version_stepping

Set to the GFX stepping generation number of the target being assembled for. For example, when assembling for a “GFX704” target this will be set to the integer value “4”. The possible GFX stepping generation numbers are presented in Processors.

.kernel.vgpr_count

Set to zero each time a .amdgpu_hsa_kernel (name) directive is encountered. At each instruction, if the current value of this symbol is less than or equal to the maximum VGPR number explicitly referenced within that instruction then the symbol value is updated to equal that VGPR number plus one.

.kernel.sgpr_count

Set to zero each time a .amdgpu_hsa_kernel (name) directive is encountered. At each instruction, if the current value of this symbol is less than or equal to the maximum VGPR number explicitly referenced within that instruction then the symbol value is updated to equal that SGPR number plus one.

Code Object V2 Directives

Warning

Code object V2 is not the default code object version emitted by this version of LLVM.

AMDGPU ABI defines auxiliary data in output code object. In assembly source, one can specify them with assembler directives.

.hsa_code_object_version major, minor

major and *minor* are integers that specify the version of the HSA code object that will be generated by the assembler.

.hsa_code_object_isa [major, minor, stepping, vendor, arch]

major, *minor*, and *stepping* are all integers that describe the instruction set architecture (ISA) version of the assembly program.

vendor and *arch* are quoted strings. *vendor* should always be equal to “AMD” and *arch* should always be equal to “AMDGPU”.

By default, the assembler will derive the ISA version, *vendor*, and *arch* from the value of the -mcpu option that is passed to the assembler.

.amdgpu_hsa_kernel (name)

This directives specifies that the symbol with given name is a kernel entry point (label) and the object should contain corresponding symbol of type STT_AMDGPU_HSA_KERNEL.

.amd_kernel_code_t

This directive marks the beginning of a list of key / value pairs that are used to specify the amd_kernel_code_t object that will be emitted by the assembler. The list must be terminated by the *.end_amd_kernel_code_t* directive. For any amd_kernel_code_t values that are unspecified a default value will be used. The default value for all keys is 0, with the following exceptions:

- *amd_code_version_major* defaults to 1.
- *amd_kernel_code_version_minor* defaults to 2.
- *amd_machine_kind* defaults to 1.
- *amd_machine_version_major*, *machine_version_minor*, and *amd_machine_version_stepping* are derived from the value of the -mcpu option that is passed to the assembler.
- *kernel_code_entry_byte_offset* defaults to 256.
- *wavefront_size* defaults 6 for all targets before GFX10. For GFX10 onwards defaults to 6 if target feature `wavefrontsize64` is enabled, otherwise 5. Note that wavefront size is specified as a power of two, so a value of n means a size of 2^ n.
- *call_convention* defaults to -1.
- *kernarg_segment_alignment*, *group_segment_alignment*, and *private_segment_alignment* default to 4. Note that alignments are specified as a power of 2, so a value of n means an alignment of 2^ n.
- *enable_tg_split* defaults to 1 if target feature `tgsplit` is enabled for GFX90A onwards.
- *enable_wgp_mode* defaults to 1 if target feature `cumode` is disabled for GFX10 onwards.
- *enable_mem_ordered* defaults to 1 for GFX10 onwards.

The *.amd_kernel_code_t* directive must be placed immediately after the function label and before any instructions.

For a full list of amd_kernel_code_t keys, refer to AMDGPU ABI document, comments in lib/Target/AMDGPU/AmdKernelCodeT.h and test/CodeGen/AMDGPU/hsa.s.

Code Object V2 Example Source Code

Warning

Code Object V2 is not the default code object version emitted by this version of LLVM.

Here is an example of a minimal assembly source file, defining one HSA kernel:

```
 1 .hsa_code_object_version 1,0
 2 .hsa_code_object_isa
 3
 4 .hsatext
 5 .globl  hello_world
 6 .p2align 8
 7 .amdgpu_hsa_kernel hello_world
 8
 9 hello_world:
10
11    .amd_kernel_code_t
12       enable_sgpr_kernarg_segment_ptr = 1
13       is_ptr64 = 1
14       compute_pgm_rsrc1_vgprs = 0
15       compute_pgm_rsrc1_sgprs = 0
16       compute_pgm_rsrc2_user_sgpr = 2
17       compute_pgm_rsrc1_wgp_mode = 0
18       compute_pgm_rsrc1_mem_ordered = 0
19       compute_pgm_rsrc1_fwd_progress = 1
20   .end_amd_kernel_code_t
21
22   s_load_dwordx2 s[0:1], s[0:1] 0x0
23   v_mov_b32 v0, 3.14159
24   s_waitcnt lgkmcnt(0)
25   v_mov_b32 v1, s0
26   v_mov_b32 v2, s1
27   flat_store_dword v[1:2], v0
28   s_endpgm
29 .Lfunc_end0:
30      .size   hello_world, .Lfunc_end0-hello_world
```

Code Object V3 and Above Predefined Symbols

The AMDGPU assembler defines and updates some symbols automatically. These symbols do not affect code generation.

.amdgcn.gfx_generation_number

Set to the GFX major generation number of the target being assembled for. For example, when assembling for a “GFX9” target this will be set to the integer value “9”. The possible GFX major

generation numbers are presented in Processors.

.amdgcn.gfx_generation_minor

Set to the GFX minor generation number of the target being assembled for. For example, when assembling for a “GFX810” target this will be set to the integer value “1”. The possible GFX minor generation numbers are presented in Processors.

.amdgcn.gfx_generation_stepping

Set to the GFX stepping generation number of the target being assembled for. For example, when assembling for a “GFX704” target this will be set to the integer value “4”. The possible GFX stepping generation numbers are presented in Processors.

.amdgcn.next_free_vgpr

Set to zero before assembly begins. At each instruction, if the current value of this symbol is less than or equal to the maximum VGPR number explicitly referenced within that instruction then the symbol value is updated to equal that VGPR number plus one.

May be used to set the *.amdhsa_next_free_vgpr* directive in AMDHSA Kernel Assembler Directives.

May be set at any time, e.g. manually set to zero at the start of each kernel.

.amdgcn.next_free_sgpr

Set to zero before assembly begins. At each instruction, if the current value of this symbol is less than or equal the maximum SGPR number explicitly referenced within that instruction then the symbol value is updated to equal that SGPR number plus one.

May be used to set the *.amdhsa_next_free_spgr* directive in AMDHSA Kernel Assembler Directives.

May be set at any time, e.g. manually set to zero at the start of each kernel.

Code Object V3 and Above Directives

Directives which begin with `.amdgcn` are valid for all `amdgcn` architecture processors, and are not OS-specific. Directives which begin with `.amdhsa` are specific to `amdgcn` architecture processors when the `amdhsa` OS is specified. See Target Triples and Processors.

.amdgcn_target <target-triple> “-” <target-id>

Optional directive which declares the `<target-triple>-<target-id>` supported by the containing assembler source file. Used by the assembler to validate command-line options such as `-triple`, `-mcpu`, and `--offload-arch=<target-id>`. A non-canonical target ID is allowed. See Target Triples and Target ID.

> **Note**
>
> The target ID syntax used for code object V2 to V3 for this directive differs from that used elsewhere. See Code Object V2 to V3 Target ID.

.amdhsa_kernel <name>

Creates a correctly aligned AMDHSA kernel descriptor and a symbol, `<name>.kd`, in the current location of the current section. Only valid when the OS is `amdhsa`. `<name>` must be a symbol that labels the first instruction to execute, and does not need to be previously defined.

Marks the beginning of a list of directives used to generate the bytes of a kernel descriptor, as described in Kernel Descriptor. Directives which may appear in this list are described in AMDHSA Kernel Assembler Directives. Directives may appear in any order, must be valid for the target being assembled for, and cannot be repeated. Directives support the range of values specified by the field they reference in Kernel Descriptor. If a directive is not specified, it is assumed to have its default value, unless it is marked as "Required", in which case it is an error to omit the directive. This list of directives is terminated by an `.end_amdhsa_kernel` directive.

AMDHSA Kernel Assembler Directives

Directive	Default	Supported On	Description
`.amdhsa_group_segment_fixed_size`	0	GFX6-GFX11	Controls GROUP_SEGMENT_FIXED_SIZE in Code Object V3 Kernel Descriptor.
`.amdhsa_private_segment_fixed_size`	0	GFX6-GFX11	Controls PRIVATE_SEGMENT_FIXED_SIZE in Code Object V3 Kernel Descriptor.
`.amdhsa_kernarg_size`	0	GFX6-GFX11	Controls KERNARG_SIZE in Code Object V3 Kernel Descriptor.
`.amdhsa_user_sgpr_count`	0	GFX6-GFX11	Controls USER_SGPR_COUNT in COMPUTE_PGM_RSRC2 compute_pgm_rsrc2 for GFX6-GFX11
`.amdhsa_user_sgpr_private_segment_buffer`	0	GFX6-GFX10 (except	Controls ENABLE_SGPR_PRIVATE_SEGMENT_BUFFER in Code Object V3 Kernel Descriptor.

		GFX940)	
.amdhsa_user_sgpr_dispatch_ptr	0	GFX6-GFX11	Controls ENABLE_SGPR_DISPATCH_PTR in Code Object V3 Kernel Descriptor.
.amdhsa_user_sgpr_queue_ptr	0	GFX6-GFX11	Controls ENABLE_SGPR_QUEUE_PTR in Code Object V3 Kernel Descriptor.
.amdhsa_user_sgpr_kernarg_segment_ptr	0	GFX6-GFX11	Controls ENABLE_SGPR_KERNARG_SEGMENT_PTR in Code Object V3 Kernel Descriptor.
.amdhsa_user_sgpr_dispatch_id	0	GFX6-GFX11	Controls ENABLE_SGPR_DISPATCH_ID in Code Object V3 Kernel Descriptor.
.amdhsa_user_sgpr_flat_scratch_init	0	GFX6-GFX10 (except GFX940)	Controls ENABLE_SGPR_FLAT_SCRATCH_INIT in Code Object V3 Kernel Descriptor.
.amdhsa_user_sgpr_private_segment_size	0	GFX6-GFX11	Controls ENABLE_SGPR_PRIVATE_SEGMENT_SIZE in Code Object V3 Kernel Descriptor.
.amdhsa_wavefront_size32	Target Feature Specific (wavefrontsize64)	GFX10-GFX11	Controls ENABLE_WAVEFRONT_SIZE32 in Code Object V3 Kernel Descriptor.
.amdhsa_uses_dynamic_stack	0	GFX6-GFX11	Controls USES_DYNAMIC_STACK in Code Object V3 Kernel Descriptor.
.amdhsa_system_sgpr_private_segment_wavefront_offset	0	GFX6-GFX10 (except GFX940)	Controls ENABLE_PRIVATE_SEGMENT in compute_pgm_rsrc2 for GFX6-GFX11.
.amdhsa_enable_private_segment	0	GFX940, GFX11	Controls ENABLE_PRIVATE_SEGMENT in compute_pgm_rsrc2 for GFX6-GFX11.
.amdhsa_system_sgpr_workgroup_id_x	1	GFX6-GFX11	Controls ENABLE_SGPR_WORKGROUP_ID_X in compute_pgm_rsrc2 for GFX6-GFX11.
.amdhsa_system_sgpr_workgroup_id_y	0	GFX6-GFX11	Controls ENABLE_SGPR_WORKGROUP_ID_Y in compute_pgm_rsrc2 for GFX6-GFX11.
.amdhsa_system_sgpr_workgroup_id_z	0	GFX6-GFX11	Controls ENABLE_SGPR_WORKGROUP_ID_Z in compute_pgm_rsrc2 for GFX6-GFX11.
.amdhsa_system_sgpr_workgroup_info	0	GFX6-GFX11	Controls ENABLE_SGPR_WORKGROUP_INFO in compute_pgm_rsrc2 for GFX6-GFX11.
.amdhsa_system_vgpr_workitem_id	0	GFX6-GFX11	Controls ENABLE_VGPR_WORKITEM_ID in compute_pgm_rsrc2 for GFX6-GFX11. Possible values

			are defined in System VGPR Work-Item ID Enumeration Values.
.amdhsa_next_free_vgpr	Required	GFX6-GFX11	Maximum VGPR number explicitly referenced, plus one. Used to calculate GRANULATED_WORKITEM_VGPR_COUNT in compute_pgm_rsrc1 for GFX6-GFX11.
.amdhsa_next_free_sgpr	Required	GFX6-GFX11	Maximum SGPR number explicitly referenced, plus one. Used to calculate GRANULATED_WAVEFRONT_SGPR_COUNT in compute_pgm_rsrc1 for GFX6-GFX11.
.amdhsa_accum_offset	Required	GFX90A, GFX940	Offset of a first AccVGPR in the unified register file. Used to calculate ACCUM_OFFSET in compute_pgm_rsrc3 for GFX90A, GFX940.
.amdhsa_reserve_vcc	1	GFX6-GFX11	Whether the kernel may use the special VCC SGPR. Used to calculate GRANULATED_WAVEFRONT_SGPR_COUNT in compute_pgm_rsrc1 for GFX6-GFX11.
.amdhsa_reserve_flat_scratch	1	GFX7-GFX10 (except GFX940)	Whether the kernel may use flat instructions to access scratch memory. Used to calculate GRANULATED_WAVEFRONT_SGPR_COUNT in compute_pgm_rsrc1 for GFX6-GFX11.
.amdhsa_reserve_xnack_mask	Target Feature Specific (xnack)	GFX8-GFX10	Whether the kernel may trigger XNACK replay. Used to calculate GRANULATED_WAVEFRONT_SGPR_COUNT in compute_pgm_rsrc1 for GFX6-GFX11.
.amdhsa_float_round_mode_32	0	GFX6-GFX11	Controls FLOAT_ROUND_MODE_32 in compute_pgm_rsrc1 for GFX6-GFX11. Possible values are defined in Floating Point Rounding Mode Enumeration Values.
.amdhsa_float_round_mode_16_64	0	GFX6-GFX11	Controls FLOAT_ROUND_MODE_16_64 in compute_pgm_rsrc1 for GFX6-GFX11. Possible values are defined in Floating Point Rounding Mode Enumeration Values.
.amdhsa_float_denorm_mode_32	0	GFX6-GFX11	Controls FLOAT_DENORM_MODE_32 in compute_pgm_rsrc1 for GFX6-GFX11. Possible values are defined in Floating Point Denorm Mode Enumeration Values.
.amdhsa_float_denorm_mode_16_64	3	GFX6-	Controls FLOAT_DENORM_MODE_16_64 in

		GFX11	compute_pgm_rsrc1 for GFX6-GFX11. Possible values are defined in Floating Point Denorm Mode Enumeration Values.
.amdhsa_dx10_clamp	1	GFX6-GFX11	Controls ENABLE_DX10_CLAMP in compute_pgm_rsrc1 for GFX6-GFX11.
.amdhsa_ieee_mode	1	GFX6-GFX11	Controls ENABLE_IEEE_MODE in compute_pgm_rsrc1 for GFX6-GFX11.
.amdhsa_fp16_overflow	0	GFX9-GFX11	Controls FP16_OVFL in compute_pgm_rsrc1 for GFX6-GFX11.
.amdhsa_tg_split	Target Feature Specific (tgsplit)	GFX90A, GFX940, GFX11	Controls TG_SPLIT in compute_pgm_rsrc3 for GFX90A, GFX940.
.amdhsa_workgroup_processor_mode	Target Feature Specific (cumode)	GFX10-GFX11	Controls ENABLE_WGP_MODE in Code Object V3 Kernel Descriptor.
.amdhsa_memory_ordered	1	GFX10-GFX11	Controls MEM_ORDERED in compute_pgm_rsrc1 for GFX6-GFX11.
.amdhsa_forward_progress	0	GFX10-GFX11	Controls FWD_PROGRESS in compute_pgm_rsrc1 for GFX6-GFX11.
.amdhsa_shared_vgpr_count	0	GFX10-GFX11	Controls SHARED_VGPR_COUNT in compute_pgm_rsrc3 for GFX10-GFX11.
.amdhsa_exception_fp_ieee_invalid_op	0	GFX6-GFX11	Controls ENABLE_EXCEPTION_IEEE_754_FP_INVALID_OPERATION in compute_pgm_rsrc2 for GFX6-GFX11.
.amdhsa_exception_fp_denorm_src	0	GFX6-GFX11	Controls ENABLE_EXCEPTION_FP_DENORMAL_SOURCE in compute_pgm_rsrc2 for GFX6-GFX11.
.amdhsa_exception_fp_ieee_div_zero	0	GFX6-GFX11	Controls ENABLE_EXCEPTION_IEEE_754_FP_DIVISION_BY_ZERO in compute_pgm_rsrc2 for GFX6-GFX11.
.amdhsa_exception_fp_ieee_overflow	0	GFX6-GFX11	Controls ENABLE_EXCEPTION_IEEE_754_FP_OVERFLOW in compute_pgm_rsrc2 for GFX6-GFX11.
.amdhsa_exception_fp_ieee_underflow	0	GFX6-GFX11	Controls ENABLE_EXCEPTION_IEEE_754_FP_UNDERFLOW in compute_pgm_rsrc2 for GFX6-GFX11.
.amdhsa_exception_fp_ieee_inexact	0	GFX6-GFX11	Controls ENABLE_EXCEPTION_IEEE_754_FP_INEXACT in compute_pgm_rsrc2 for GFX6-GFX11.
.amdhsa_exception_int_div_zero	0	GFX6-	Controls ENABLE_EXCEPTION_INT_DIVIDE_BY_ZERO in

GFX11	compute_pgm_rsrc2 for GFX6-GFX11.

.amdgpu_metadata

Optional directive which declares the contents of the `NT_AMDGPU_METADATA` note record (see AMDGPU Code Object V3 and Above ELF Note Records).

The contents must be in the [YAML] markup format, with the same structure and semantics described in Code Object V3 Metadata, Code Object V4 Metadata or Code Object V5 Metadata.

This directive is terminated by an `.end_amdgpu_metadata` directive.

Code Object V3 and Above Example Source Code

Here is an example of a minimal assembly source file, defining one HSA kernel:

```
 1 .amdgcn_target "amdgcn-amd-amdhsa--gfx900+xnack" // optional
 2
 3 .text
 4 .globl hello_world
 5 .p2align 8
 6 .type hello_world,@function
 7 hello_world:
 8   s_load_dwordx2 s[0:1], s[0:1] 0x0
 9   v_mov_b32 v0, 3.14159
10   s_waitcnt lgkmcnt(0)
11   v_mov_b32 v1, s0
12   v_mov_b32 v2, s1
13   flat_store_dword v[1:2], v0
14   s_endpgm
15 .Lfunc_end0:
16   .size   hello_world, .Lfunc_end0-hello_world
17
18 .rodata
19 .p2align 6
20 .amdhsa_kernel hello_world
21   .amdhsa_user_sgpr_kernarg_segment_ptr 1
22   .amdhsa_next_free_vgpr .amdgcn.next_free_vgpr
23   .amdhsa_next_free_sgpr .amdgcn.next_free_sgpr
24 .end_amdhsa_kernel
25
26 .amdgpu_metadata
27 ---
28 amdhsa.version:
29   - 1
30   - 0
```

```
31 amdhsa.kernels:
32   - .name: hello_world
33     .symbol: hello_world.kd
34     .kernarg_segment_size: 48
35     .group_segment_fixed_size: 0
36     .private_segment_fixed_size: 0
37     .kernarg_segment_align: 4
38     .wavefront_size: 64
39     .sgpr_count: 2
40     .vgpr_count: 3
41     .max_flat_workgroup_size: 256
42     .args:
43       - .size: 8
44         .offset: 0
45         .value_kind: global_buffer
46         .address_space: global
47         .actual_access: write_only
48 //...
49 .end_amdgpu_metadata
```

This kernel is equivalent to the following HIP program:

```
1 __global__ void hello_world(float *p) {
2     *p = 3.14159f;
3 }
```

If an assembly source file contains multiple kernels and/or functions, the .amdgcn.next_free_vgpr and .amdgcn.next_free_sgpr symbols may be reset using the `.set <symbol>, <expression>` directive. For example, in the case of two kernels, where `function1` is only called from `kernel1` it is sufficient to group the function with the kernel that calls it and reset the symbols between the two connected components:

```
 1 .amdgcn_target "amdgcn-amd-amdhsa--gfx900+xnack" // optional
 2
 3 // gpr tracking symbols are implicitly set to zero
 4
 5 .text
 6 .globl kern0
 7 .p2align 8
 8 .type kern0,@function
 9 kern0:
10   // ...
11   s_endpgm
12 .Lkern0_end:
13   .size   kern0, .Lkern0_end-kern0
14
15 .rodata
```

```
.p2align 6
.amdhsa_kernel kern0
  // ...
  .amdhsa_next_free_vgpr .amdgcn.next_free_vgpr
  .amdhsa_next_free_sgpr .amdgcn.next_free_sgpr
.end_amdhsa_kernel

// reset symbols to begin tracking usage in func1 and kern1
.set .amdgcn.next_free_vgpr, 0
.set .amdgcn.next_free_sgpr, 0

.text
.hidden func1
.global func1
.p2align 2
.type func1,@function
func1:
  // ...
  s_setpc_b64 s[30:31]
.Lfunc1_end:
.size func1, .Lfunc1_end-func1

.globl kern1
.p2align 8
.type kern1,@function
kern1:
  // ...
  s_getpc_b64 s[4:5]
  s_add_u32 s4, s4, func1@rel32@lo+4
  s_addc_u32 s5, s5, func1@rel32@lo+4
  s_swappc_b64 s[30:31], s[4:5]
  // ...
  s_endpgm
.Lkern1_end:
  .size   kern1, .Lkern1_end-kern1

.rodata
.p2align 6
.amdhsa_kernel kern1
  // ...
  .amdhsa_next_free_vgpr .amdgcn.next_free_vgpr
  .amdhsa_next_free_sgpr .amdgcn.next_free_sgpr
.end_amdhsa_kernel
```

These symbols cannot identify connected components in order to automatically track the usage for each kernel. However, in some cases careful organization of the kernels and functions in the source file means there is minimal additional effort required to accurately calculate GPR usage.

Additional Documentation

[AMD-GCN-GFX6] *(1, 2)* AMD Southern Islands Series ISA
[AMD-GCN-GFX7] *(1, 2)* AMD Sea Islands Series ISA
[AMD-GCN-GFX8] *(1, 2)* AMD GCN3 Instruction Set Architecture
[AMD-GCN-GFX900-GFX904-VEGA] *(1, 2)* AMD Vega Instruction Set Architecture
[AMD-GCN-GFX906-VEGA7NM] *(1, 2)* AMD Vega 7nm Instruction Set Architecture
[AMD-GCN-GFX908-CDNA1] *(1, 2)* AMD Instinct MI100 Instruction Set Architecture
[AMD-GCN-GFX90A-CDNA2] *(1, 2)* AMD Instinct MI200 Instruction Set Architecture
[AMD-GCN-GFX10-RDNA1] *(1, 2)* AMD RDNA 1.0 Instruction Set Architecture
[AMD-GCN-GFX10-RDNA2] *(1, 2)* AMD RDNA 2 Instruction Set Architecture
[AMD-GCN-GFX11-RDNA3] *(1, 2)* AMD RDNA 3 Instruction Set Architecture
[AMD-RADEON-HD-2000-3000] AMD R6xx shader ISA
[AMD-RADEON-HD-4000] AMD R7xx shader ISA
[AMD-RADEON-HD-5000] AMD Evergreen shader ISA
[AMD-RADEON-HD-6000] AMD Cayman/Trinity shader ISA
[AMD-ROCm] AMD ROCm™ Platform
[AMD-ROCm-github] *(1, 2)* AMD ROCm™ github
[AMD-ROCm-Release-Notes] AMD ROCm Release Notes
[CLANG-ATTR] *(1, 2, 3, 4)* Attributes in Clang
[DWARF] *(1, 2)* DWARF Debugging Information Format
[ELF] *(1, 2)* Executable and Linkable Format (ELF)
[HRF] Heterogeneous-race-free Memory Models
[HSA] *(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)* Heterogeneous System Architecture (HSA) Foundation
[MsgPack] *(1, 2, 3, 4)* Message Pack
[OpenCL] *(1, 2)* The OpenCL Specification Version 2.0
[SEMVER] *(1, 2, 3)* Semantic Versioning
[YAML] *(1, 2, 3)* YAML Ain't Markup Language (YAML™) Version 1.2