HSA Platform System Architecture Specification

Revision: Version 1.2 • Issue Date: 2 May 2018

Acknowledgments

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

AMD

- Brad Beckmann
- Paul Blinzer (Workgroup Chair)
- Mark Fowler
- Michael Mantor
- Rex McCrary
- Ben Sander
- Vinod Tipparaju
- Tony Tye

ARM

- Ian Bratt
- Ian Devereux
- Richard Grisenthwaite
- Djordje Kovacevic
- Jason Parker
- Håkan Persson (Spec Editor)
- Andrew Rose

General Processor Technologies

- John Glossner

Imagination

- James Aldis
- Andy Glew
- John Howson
- Georg Kolling
- Jason Meredith
- Mark Rankilor

Mediatek

- Richard Bagley
- Stephen Huang
- Roy Ju
- Fred Liao
- Chien-Ping Lu

Qualcomm

- Greg Bellows
- Lihan Bin
- PJ Bosley
- Alex Bourd
- Jamie Esliger
- Benedict Gaster
- Derek Hower
- Lee Howes
- Wilson Kwan
- Bob Rychlik
- Robert J. Simpson
- Michael Weber

Samsung

- Ignacio Llamas
- Soojung Ryu
- Michael C. Shebanow

Sony

- Jim Rasmusson

ST Microelectronics

- Marcello Coppola

Contents

Figures

Tables

About the HSA Platform System Architecture Specification

This document identifies, from a hardware point of view, system architecture requirements necessary to support the Heterogeneous System Architecture (HSA) programming model and HSA application and system software infrastructure.

It defines a set of functionality and features for HSA hardware product deliverables to meet the minimum specified requirements to qualify for a valid HSA product.

Where necessary, the document illustrates possible design implementations to clarify expected operation. Unless otherwise specified, these implementations are not intended to imply a specific hardware or software design.

Audience

This document is written for system and component architects interested in supporting the HSA infrastructure (hardware and software) within platform designs.

Terminology

See Appendix B Glossary (on page 75) for definitions of terminology.

This specification uses terminology and syntax from the C family of programming languages. For example, type names such as `uint64_t` are defined in the C99 and C++ specifications.

HSA Information Sources

- *HSA Programmer's Reference Manual Version 1.2*
- *HSA Runtime Programmer's Reference Manual Version 1.2*
- *HSA Platform System Architecture Specification Version 1.2*

CHAPTER 1.
System Architecture Requirements: Overview

1.1 What is HSA?

The Heterogeneous System Architecture (HSA) is designed to efficiently support a wide assortment of data-parallel and task-parallel programming models. A single HSA system can support multiple instruction sets based on host CPUs and kernel agents.

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

An HSA-compliant system will meet the requirements for a queuing model, a memory model, quality of service, and an instruction set for parallel processing. It also meets the requirements for enabling heterogeneous programming models for computing platforms using standardized interfaces, processes, communication protocols, and memory models.

1.2 Keywords

This document specifies HSA system requirements at different levels using the keywords defined below:

- *"Must"*: This word, or the terms "required" or "shall," mean that the definition is an absolute requirement of the specification.
- *"Must not"*: This phrase, or the phrase "shall not," mean that the definition is an absolute prohibition of the specification.
- *"Should"*: This word, or the adjective "recommended," mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.
- *"Should not"*: This phrase, or the phrase "not recommended," mean that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood and the case carefully weighed before implementing any behavior described with this label.
- *"May"*: This word, or the adjective "optional," means that an item is truly optional. One vendor may choose to include the item because a particular marketplace requires it or because the vendor feels that it enhances the product while another vendor may omit the same item. An implementation which does not include a particular option MUST be prepared to interoperate with another implementation which does include the option, though perhaps with reduced functionality. In the same vein an implementation which does include a particular option MUST be prepared to interoperate with another implementation which does not include the option (except, of course, for the feature the option provides).

These definitions are exactly as described in the Internet Engineering Task Force RFC 2119, BCP 14, Key words for use in RFCs to Indicate Requirement Levels.

One vendor might choose to include an optional item because a particular marketplace requires it or because the vendor feels that it enhances the product, while another vendor might omit the same item.

1.3 Minimum vs. complete HSA software product

This document also provides guidance for the HSA product to be more complete, enhanced, and competitive:

- The minimum HSA software product is defined by the mandatory requirements highlighted with the key words “shall,” “must,” and “required.”
- The complete HSA software product feature set is defined by the key words “should” and “recommended.”

Unless otherwise stated, functionality referred to by this document that is outside of the HSA software, such as a software component or an API, indicates the version that was current at the time of HSA software delivery.

NOTE: A higher-level requirement specification shall supersede a lower-level (kernel agent) requirement if there is an implied contradiction.

1.4 HSA programming model

The HSA programming model is enabled through the presence of a select number of key hardware and system features for the heterogeneous system components. Examples are kernel agents and other agents, interface connection fabric, memory, and so forth. The presence of these features on an HSA-compatible system simplifies the number of permutations that the software stack needs to deal with. Thus, the HSA programming model is much simpler than heterogeneous system programming models based on more traditional system design.

1.5 Requirements for an HSA-compliant system

A system capable of passing all HSA conformance tests is a combination of:

- one or more host CPU agents executing the HSA runtime (see *HSA Runtime Programmer’s Reference Manual Version 1.2*),
- one or more kernel agents able to execute HSAIL programs (see *HSA Programmer's Reference Manual Version 1.2*), and
- zero or more other agents that participate in the HSA memory model (see *HSA Platform System Architecture Specification Version 1.2*)

that also comply with the following requirements defined in this document:

- Shared virtual memory, including adherence to the HSA memory model. See 2.1 Requirement: Shared virtual memory (on page 11).
- Cache coherency domains, including host CPUs, kernel agents and other agents and interconnecting I/O bus fabric. See 2.2 Requirement: Cache coherency domains (on page 13).
- Flat addressing of memory. See 2.3 Requirement: Flat addressing (on page 13).
- Consistent system endianess. See 2.4 Requirement: Endianess (on page 13).
- Memory-based signaling and synchronization primitives between all HSA-enabled system

components, including support for platform atomics. See 2.5 Requirement: Signaling and synchronization (on page 14).

- Atomic memory operations, see 2.6 Requirement: Atomic memory operations (on page 16).
- HSA system timestamp providing a uniform view of time across the HSA system. See 2.7 Requirement: HSA system timestamp (on page 17).
- User mode queues with low-latency application-level dispatch to hardware. See 2.8 Requirement: User mode queuing (on page 17).
- Use of Architected Queuing Language (AQL), which reduces launch latency by allowing applications to enqueue tasks to kernel agents and other agents. See 2.9 Requirement: Architected Queuing Language (AQL) (on page 24).
- Agent Scheduling, see 2.10 Requirement: Agent scheduling (on page 32).
- Kernel dispatch forward progress as defined in 2.11 Requirement: Kernel dispatch forward progress (on page 32).
- Kernel agent floating point support. See 2.12 Requirement: Floating point support (on page 33).
- Kernel agent error reporting mechanism that meets a similar level of detail as provided by host CPUs, including adherence to the policies specified in 2.13 Requirement: IEEE754-2008 floating point exceptions (on page 33).
- Kernel agent debug infrastructure that meets the specified level of functional support. See 2.14 Requirement: Kernel agent hardware debug infrastructure (on page 35).
- Architected kernel agent discovery by means of system firmware tables provided by ACPI or equivalent architected firmware infrastructure. This allows system software and, in turn, application software to discover and leverage platform topology independent of the system-specific bus fabric and host CPU infrastructure, as long as they support the other HSA-relevant features. See 2.15 Requirement: HSA platform topology discovery (on page 35).
- Optionally an HSA platform supports image operations. See 2.16 Requirement: Images (on page 42).
- Optionally an HSA platform supports profiling of HSA software. See 2.17 Requirement: Profiling (on page 45).

Note that there are a wide variety of methods for implementing these requirements.

CHAPTER 2.
System Architecture Requirements: Details

2.1 Requirement: Shared virtual memory

A compliant HSA system shall allow agents to access shared system memory through the common HSA unified virtual address space. The minimum virtual address width that must be supported by all agents is 48 bits for a 64-bit HSA system implementation and 32 bits for a 32-bit HSA system implementation.[1] The full addressable virtual address space shall be available for both instructions and data.

Pointers are stored in 32-bit words in 32-bit HSA systems and in 64-bit words in 64-bit HSA systems.

System Software may reserve ranges of the virtual address space for agent or system internal use, e.g., private and group memory segments. Access to locations within these ranges from a particular agent follow implementation-specific system software policy and are not subject to the shared virtual memory requirements and access properties further defined in this specification. The reserved ranges must be discoverable or configurable by system software. System software is expected not to allocate pages for HSA application access in these non-shareable ranges.

The requirement on shared virtual memory is relaxed in the base profile to only apply to buffers allocated through the HSA runtime memory allocator. Base profile kernel agents must support fine-grained sharing in buffers for all global segment memory that can be allocated. An application using a base profile kernel agent may choose to not allocate all global segment buffers with fine-grained sharing.

Similarly global segment memory can also be allocated for use for kernel arguments without any restrictions on the total amount of such memory other than the total amount of global segment memory in the system.

Each agent shall handle shared memory virtual address translation through page tables managed by system software. System software is responsible for setting, maintaining, and invalidating these page tables according to its policy. Agents must observe the shared memory page properties as established by system software. The observed shared memory page properties shall be consistent across all agents.

The shared memory virtual address translation shall:

- Interpret shared memory attributes consistently for all agents and ensure that memory protection mechanisms cannot be circumvented. In particular:
    - The same page sizes shall be supported for all agents.
    - Read and write permissions apply to all agents equally.
    - Execute restrictions are not required to apply to kernel agents.
    - Agents must support shared virtual memory for the lowest privilege level. Agents are not required to support shared virtual memory for higher privilege levels that may be used by

[1] There is no minimum physical memory size for an HSA system implementation.

host CPU operating system or hypervisor.

- Execute accesses from agents to shared virtual memory must use only the lowest privilege level
- If implemented in the HSA system, page status bit updates (e.g., “access” and “dirty”) must be tracked across all agents to insure proper paging behavior.
- For the primary memory type, all agents (including the host CPUs) must interpret cacheability and data coherency properties (excluding those for read-only image data) in the same way.
- For the primary memory type, an agent shall interpret the memory type in a common way and in the same way as host CPUs, with the following caveats:
  - End-point ordering properties.
  - Observation ordering properties.
  - Multi-copy atomicity properties.
- For any memory type other than the primary memory type, an agent shall either:
  A. Generate a memory fault on use of that memory type, or
  B. Interpret the memory type in a common way and in the same way as the host CPU, with the following caveats:
     - End-point ordering properties.
     - Observation ordering properties.
     - Multi-copy atomicity properties.
     - Cacheability and data coherency properties.
- For all memory types, there is a requirement of the same interpretation of speculation permission properties by all agents and the host CPU.

- Provide a mechanism to notify system software in case of a translation fault. This notification shall include the virtual address and a device tag to identify the agent that issued the translation request.
- Provide a mechanism to handle a recoverable translation fault (e.g., page not present). System software shall be able to initiate a retry of the address translation request which triggered the translation fault.
- Provide the concept of a process address space ID (PASID) in the protocol to service separate, per-process virtual address spaces within the system.[1] For systems that support hardware virtualization, the PASID implementation shall allow for a separation of PASID bits for virtualization (e.g., as "PartitionID") and PASID bits for HSA Memory Management Unit (HSA MMU) use. The number of bits used for PASID for HSA MMU functionality shall be at least 8. It is recommended to support a higher minimum number of PASID bits (16 bits) for all agents if the HSA system is targeted by more advanced system software, running many processes concurrently.

[1]The PASID ECNs of PCI Express® 3.0 provides an example of the implementation and use.

2.2 Requirement: Cache coherency domains

Data accesses to global memory from all agents shall be coherent without the need for explicit cache maintenance. This only applies to global memory locations with the primary memory type of the translation system and does not apply to image accesses (see 2.16 Requirement: Images (on page 42)) for details on images). An HSA application may limit the scope of coherency for data items as a performance optimization. See Chapter 3 HSA Memory Consistency Model (on page 47) for details.

The specification does not require that data memory accesses from agents are coherent for any memory location with any memory type other than the primary memory type supported by the translation system.

The specification does not require that instruction memory accesses to any memory type by agents are coherent.

The specification does not require that agents have a coherent view of any memory location where the agents do not specify the same memory attributes.

Coherency and ordering between HSA shared virtual memory aliases to the same physical address are not guaranteed.

2.2.1 Read-only image data

Read-only image data is required to remain static during the execution of an HSA kernel. Implications of this include (but are not limited to) that it is not allowed to modify the data for a read-only image view using a read/write image view for the same data, or to directly access the data array from either the same kernel, the host CPU, or another kernel running in parallel.

2.3 Requirement: Flat addressing

A kernel agent must support a flat virtual address space for all memory it can access. An HSAIL operation on a flat address is equivalent to the same HSAIL operation specifying the corresponding segment. Specifying a segment for an HSAIL memory operation may improve performance.

Synchronizing memory operations using a flat address apply to the segment designated by the flat address.

The effective memory scope for a synchronizing memory operation using a flat address is the minimum of the scope specified by the operation and the widest scope supported by the segment of the flat address.

The base address of a segment in the flat address space must be aligned to 4kB.

The address value of a location in the global segment is identical to its equivalent location in flat address space.

The address value of a location in the flat address space is identical to its equivalent location in host virtual address space.

The NULL value is identical for global segment and the flat address space and identical to the NULL value of the host platform. NULL may be translated to private values for other segments.

2.4 Requirement: Endianess

An HSA system can be implemented as either little or big endian, it is assumed only one endianess is used by the agents collaborating in the system. The following conventions regarding endianess are used in this document (and in particular for tables defining data structures):

- Allocation of bits into bytes: Bits 0..7 are stored in byte 0, bits 8..15 in byte 1, and so on.
- Bit value in multi-bit fields: The method of assigning a numeric value for each bit in a multi-bit field is not defined in this document, but all agents in a system must have the same definition.

2.5 Requirement: Signaling and synchronization

An HSA-compliant platform shall provide for the ability of HSA software to define and use signaling and synchronization primitives accessible from all agents in a non-discriminatory way. HSA signaling primitives are used to operate on a signal value which is referenced using an opaque 64-bit signal handle. In addition to the signal value, the signal handle may also have associated implementation-defined data. Collectively, the signal value and implementation data are referred to in this document as an HSA signal or simply a signal.

The signaling mechanism has the following properties:

- An HSA signal value must only be manipulated by kernel agents using the specific HSAIL mechanisms, and by the host CPU using the HSA runtime mechanisms. Other agents may use implementation-specific mechanisms that are compatible with the HSA runtime and HSAIL mechanisms for manipulating signals. The required mechanisms for HSAIL and the HSA runtime are:
    - Allocate an HSA signal:
        - The HSA system shall support allocation of signals through the HSA runtime by means of a dedicated runtime API. An application may expose this through a runtime agent dispatch.
        - Allocation of an HSA signal must include initialization of the signal value.
        - The application may optionally specify the set of agents required to be notified by a send or atomic read-modify-write operation on the signal. If the application doesn't specify any set of Agents to be notified then all agents are required to be notified by a send or atomic read-modify-write operation on the signal.
    - Destroy an HSA signal:
        - The HSA system shall support deallocation of signals through the HSA runtime by means of a dedicated runtime API. An application may expose this through a runtime agent dispatch.
    - Read the current HSA signal value:
        - The signal read mechanism must support relaxed and acquire memory ordering semantics.
        - Reading the value of an HSA signal returns the momentary value of the HSA signal, as visible to that unit of execution (see 2.16 Requirement: Images (on page 42)) for information on memory consistency). There is no guarantee that an agent will see all sequential values of a signal.
    - Wait on an HSA signal to meet a specified condition:
        - All signal wait operations specify a condition. The conditions supported are equal, not equal, less than, and greater than or equal.
        - The signal value is considered a signed integer for assessing the wait condition.

        - When a signal is modified by a send or atomic read-modify-write operation then a check of the condition is scheduled for all wait operations on that signal. The check of the condition and potential resumption of the wait operation unit of execution may also be blocked by other factors (such as OS process scheduling).
        - A wait operation does not necessary see all sequential values of a signal. The application must ensure that signals are used such that wait wakeup conditions are not invalidated before dependent unit of executions have woken up, otherwise the wakeup may not occur.
        - Signal wait operations may return sporadically. There is no guarantee that the signal value satisfies the wait condition on return. The caller must confirm whether the signal value has satisfied the wait condition.
        - Wait operations have a maximum duration before returning, which can be discovered through the HSA runtime as a count based on the HSA system timestamp frequency. The resumption of the wait operation unit of execution may also be blocked by other factors, such as OS process scheduling.
        - The signal wait mechanism must support relaxed and acquire memory ordering semantics.
        - Multiple agents must be allowed to simultaneously wait on the same HSA signal.
    - Wait on an HSA signal to meet a specified condition, with a maximum wait duration requested:
        - The requirements on the ordinary wait on signal-condition operation apply also to this operation.
        - The maximum wait duration requested is a hint. The operation may block for a shorter or longer time even if the condition is not met.
        - The maximum wait duration request is specified in the same units as the HSA system timestamp.
    - Send an HSA signal value:
        - Sending a signal value should wakeup any agents waiting on the signal in which the wait condition is satisfied.
        - It is implementation defined whether the wakeup is only issued to agents for which the wait condition is met.
        - The signal send mechanism must support relaxed and release memory ordering semantics.
    - Atomic read-modify-write an HSA signal value:
        - With the exception of wrapinc, wrapdec, min, and max, the same set of atomic operations defined for general data in HSAIL should be supported for signals in both HSAIL and the HSA runtime.
        - Performing an atomic read-modify-write update of a signal value should wakeup any agents waiting on the signal in which the wait condition is satisfied.

        - It is implementation defined whether the wakeup is only issued to agents for which the wait condition is met.
        - The signal atomic update mechanism must support relaxed, release, acquire, and acquire-release memory ordering semantics.
        - Atomic read-modify-write operations are not supported on queue doorbell signals that are created by the runtime for user mode queues associated with a kernel agent.
    - HSA signals should only be manipulated by agents within the same user process address space in which it was allocated. The HSA signaling mechanism does not support inter-user process communication.
    - There is no architectural limitation imposed on the number of HSA signals, unless implicitly limited by other requirements outlined in this document.
    - The HSA signal handle is always a 64-bit wide quantity with an opaque format:
        - An HSA signal handle value of zero must be interpreted as the absence of a signal.
    - The HSA signal value width depends on the supported machine model:
        - An HSA signal must have a 32-bit signal value when used in a small machine model.
        - An HSA signal must have a 64-bit signal value when used in a large machine model.
    - Signals are generally intended for notification between agents. Therefore, signal operations interact with the memory model as if the signal value resides in global segment memory, is naturally aligned, and is accessed using atomic memory operations at system scope with the memory ordering specified by the signal operation. See Chapter 3 HSA Memory Consistency Model (on page 47) for further information on memory consistency.

2.6 Requirement: Atomic memory operations

HSA requires the following standard atomic memory operations to be supported by kernel agents (other agents only need to support the subset of these operations required by their role in the system):

- Load from memory
- Store to memory
- Fetch from memory, apply logic operation with one additional operand, and store back. Required logic operations are bitwise AND, bitwise OR, bitwise XOR.
- Fetch from memory, apply integer arithmetic operation with one additional operand and store back. Required integer arithmetic operations are add, subtract, increment, decrement, minimum, and maximum.
- Exchange memory location with operand.
- Compare-and-swap, i.e., load memory location, compare with first operand, if equal then store second operand back to memory location.

In the small machine model, 32-bit standard atomic memory operations should be supported; the large machine model is required to support 32-bit and 64-bit standard atomic memory operations.

2.7 Requirement: HSA system timestamp

The HSA system provides for a low overhead mechanism of determining the passing of time. A system timestamp is required that can be read from HSAIL or through the HSA runtime. It is also possible to determine the system timestamp frequency through the HSA runtime.

The HSA system requires the following properties of the timestamp:

- The timestamp value is a 64-bit integer.
- The timestamp value should not roll over.
- The timestamp value should appear to increase at a constant rate to the HSA applications, regardless of any low power state the system may enter or any change to operating frequency.
- The constant observable timestamp value increase rate is in the range 1-400MHz, and can be queried through the HSA runtime.
- Provides a uniform view of time across the HSA system, i.e.:
    - The value of the system timestamp must never appear to decrease in time by a single reader.
    - The HSA system timestamp function must generate a monotonically increasing timestamp value, such that a timestamp A observed to be generated earlier than a timestamp B will have a timestamp value less than or equal to timestamp B as visible from each agent.

The requirement to not roll over can be resolved in various ways. For example, if the timestamp value is not synchronized with real-world time, and is reset on every system boot, then the roll-over period to be designed for can be the maximum expected system uptime rather than lifetime. Other implementations may want to reuse an existing real-world timestamp for the HSA system timestamp. In this case, the roll-over requirement will imply different hardware requirements.

The system timestamp function can be implemented in several ways. Examples of possible implementations are as follows, however many others are possible as long as they meet the requirements:

- A single timestamp counter peripheral in the system that generates a timestamp when it is read by an agent.
- Distributed local counters that increment at the same rate and have a synchronized startup.

An implementation may vary the amount that the timestamp is incremented by each time it is incremented, for example when entering or exiting a low power state, as long as the requirements above are met.

2.8 Requirement: User mode queuing

An HSA-compliant platform shall have support for the allocation of multiple “user-level command queues.”

For the purpose of HSA software, a user-level command queue shall be characterized as runtime-allocated, user-level, accessible virtual memory of a certain size, containing packets defined in the Architected Queueing Language (see 2.9 Requirement: Architected Queuing Language (AQL) (on page 24)). HSA software should receive memory-based structures to configure the hardware queues to allow for efficient software management of the hardware queues of the agents.

This queue memory shall be processed by the packet processor as a ring buffer, with separate memory locations defining write and read state information of that queue. The packet processor is logically a separate agent from the kernel agent, and may for example have a separate identification as a listener when allocating signals.

The user mode queue shall be defined as follows:

Table 2–1 User mode queue structure

Bits	Field Name	Description
31:0	type	Queue type for future expansion. Intended to be used for dynamic queue protocol determination. Queue types are defined in Table 2–2 (on the facing page).
63:32	features	Bit field to indicate specific features supported by the queue. HSA applications should ignore any unknown set bits. Features differ from types in that a queue of a certain type may have a certain feature (specific restrictions on behavior, additional packet types supported), but still be compatible with the basic queue type. Queue features are defined in 2.8.2 Queue features (on the facing page).
127:64	base_address	A 64-bit pointer to the base of the virtual memory which holds the AQL packets for the queue.
191:128	doorbell_signal	HSA signaling object handle, which must be signaled with the most recent write_index when a new AQL packet has been enqueued. Doorbell signals are allocated as part of the queue creation by the HSA runtime, and have restrictions on the signal operations possible. Atomic read-modify-write operations are not supported.
223:192	size	A 32-bit unsigned integer which specifies the maximum number of packets the queue can hold. The size of the queue is measured by the number of packets, which must be a power of two.
255:224		Reserved, must be 0.
319:256	id	A 64-bit ID which is unique for every queue created by an application, but is not required to be unique between processes.

agents cannot modify the user mode queue structure itself. However agents can modify the content of the packet ring buffer pointed to by the base_address field.

In addition to the data held in the queue structure defined in Table 2–1 (above), the queue defines two additional properties:

- write_index. – A 64-bit unsigned integer, initialized to 0 at queue creation time, which specifies the packet ID of the next AQL packet slot to be allocated by the application or user-level runtime. The next AQL packet is allocated at virtual address base_address + ((write_index % size) * AQL packet size).
- read_index – A 64-bit unsigned integer, initialized to 0 at queue creation time, which specifies the packet ID of the next AQL packet to be consumed by the compute unit hardware. The eldest AQL packet that is not yet released by the packet processor is at virtual address base_address + ((read_index % size) * AQL packet size).

The write_index and read_index properties cannot be accessed directly using memory operations from HSA applications or HSAIL kernels. Instead they are accessed through the HSA runtime API (HSA applications) or specific HSAIL instructions as described in 2.8.5 Queue index access (on page 22).

2.8.1 Queue types

Table 2–2 User mode queue types

ID	Name	Description
0	MULTI	Queue supports multiple producers.
1	SINGLE	Queue only supports a single producer.

2.8.2 Queue features

Table 2–3 User mode queue features

Bit	Name	Description
0	KERNEL_DISPATCH	Queue supports KERNEL_DISPATCH packet.
1	AGENT_DISPATCH	Queue supports AGENT_DISPATCH packet.
31:2	RESERVED	Must be 0.

2.8.3 Queue mechanics

HSA specifies the following rules for how the queue must be implemented using the information defined above. The rules define a contract between the submitting agent and the packet processor which enables the submitting agent to enqueue work for the associated agents in a manner that is portable across HSA implementations.

The main points of the AQL queue mechanism are as follows

- An agent allocates an AQL packet slot by incrementing the write_index. The value of the write_index before the increment operation is the packet ID of the AQL packet allocated.
- An agent assigns a packet to the packet processor by changing the AQL packet format field from INVALID. The agent is required to ensure that the rest of the AQL packet is globally visible before or at the same time as the format field is written.
    - When an agent assigns an AQL packet to the packet processor, the ownership of the AQL packet is transferred. The packet processor may update the AQL packet at any time after packet submission, and the submitting agent should not rely on the content of the AQL packet after submission.
- An agent notifies the packet processor by signaling the queue doorbell with a packet ID equal to the packet ID of the packet or to a subsequent packet ID.
    - Multiple AQL packets may be notified to the packet processor by one queue doorbell signal operation. The packet ID for the last packet notified should be used for the doorbell value.
    - When the queue doorbell is signaled, the submitting agent is required to ensure that any AQL packets referenced by the doorbell value are already globally visible.
    - It is not allowed to signal the doorbell with the packet ID for a packet not set to valid by the producer.
    - On a small machine model platform, the least significant 32 bits of the packet ID is used for the doorbell value.
    - For queues with a type restricting them to single producer mechanics, the value signaled on the doorbell must be monotonically increasing, and all packets up to the notification packet ID

must be already assigned.

- Only the packet processor or its functional equivalent is permitted to read/evaluate the doorbell signal.
- Agents producing AQL dispatches must only write to the doorbell signal.
- The value being read from a doorbell signal by agents other than the packet processor is undefined. See Chapter 3 HSA Memory Consistency Model (on page 47).

- An agent submits a task to a queue by performing the following steps:
  a. *Allocating* an AQL packet slot.
  b. *Updating* the AQL packet with the task particulars (note, the format field must remain unchanged).
  c. *Assigning* the packet to the packet processor.
  d. *Notifying* the packet processor of the packet.

In addition the following detailed constraints apply:

- The HSA signaling mechanism is restricted to within the process address space for which it is initialized, see 2.3 Requirement: Flat addressing (on page 13).
- The use of this signaling mechanism in the user mode queue implementation implies the same restriction on the user mode queues; only agents within the queue process address space are allowed to submit AQL packets to a queue.
- The size of the queue is measured by the number of AQL packets, which must be a power of two.
- The base_address must always be aligned to the AQL packet size.
- The write_index increases as new AQL packets are added to the queue by the agent. The write_index never wraps around, but the actual address offset used is ((*write_index* % *size*) * AQL packet size).
- The read_index increases as AQL packets are processed and released by the packet processor. The read_index never wraps around, but the actual address offset used is ((*read_index* % *size*) * AQL packet size).
- AQL packets are 64 bytes in size.
- When the *write_index* == *read_index*, the queue is empty.
- The format field of AQL packet slots is initialized to INVALID by the HSA system when a queue is created and is set to INVALID when a packet has been processed.
- A submitting agent may only modify an allocated AQL packet within the queue, when the "*packet ID* < *read_index* + *size*" condition is fulfilled and the AQL packet format field is set to INVALID.
- Packet processors must not process an AQL packet with the format field set to INVALID.
- The first 32 bits of an AQL packet (this includes the packet header, and thus the format field) must only be accessed using 32-bit atomic transactions.
- Packet processors are not required to process an AQL packet until the packet has been notified to the packet processor.
- Packet processors may process AQL packets after the packet format field is updated, but before the

doorbell is signaled. Agents must not rely on the doorbell as a “gate” to hold off execution of an AQL packet.

- AQL packets in a queue are dispatched in-order, but may complete in any order. An INVALID AQL packet marked in the queue will block subsequent packets from being dispatched.
- A packet processor may choose to release a packet slot (by incrementing read_index and setting the format field to INVALID) at any time after a packet has been submitted, regardless of the state of the task specified by the packet. Agents should not rely on the read_index value to imply any state of a particular task.
- The packet processor must make sure the packet slot format field is set to INVALID and made globally visible before the read_index is moved past the packet and it is made available for new task submission.

The User Mode queue base address and queue size of each of the User Mode queues may be confined by system software to system-dependent restrictions (for example, queue size and base address aligned to system page size, and so forth) as long as the imposed hardware restrictions do not impede HSA software’s use to allocate multiple such User Mode queues per application process and system software provides for APIs allowing to impose such limits on a particular operating system. Queues are allocated by HSA applications through the HSA runtime.

Packet processors shall have visibility of all their associated User Mode queues of all concurrently running processes and shall provide process-relative memory isolation and support privilege limitations for the instructions in the user-level command queue consistent with the application-level privileges as defined for the host CPU instruction set. The HSA system handles switching among the list of available queues for a packet processor, as defined by system software.

2.8.4 Multiple vs. single submitting agents

An HSA application may incorporate multiple agents or multiple unit of executions on an agent that may submit jobs to the same AQL queue. In the common case this will be multiple host CPU unit of executions, or multiple unit of executions running on a kernel agent.

In order to support job submission from multiple agents, the update of the queue write_index must use read-modify-write atomic memory operations. If only a single unit of execution in a single agent is submitting jobs to a queue then the queue write_index update can be done using an atomic store instead of a read-modify-write atomic operation.

There are several options of implementing multiple unit of execution packet submission, each with different characteristics for avoiding deadlocks if/when a queue is full (i.e., *write_index >= (read_index + size)*). Two examples are:

- Check for queue full before incrementing the write_index, and use an atomic CAS operation for the write_index increment. This allows aborting the packet slot allocation in all cases if the queue is full, but when multiple writers collide, this requires retrying the packet slot allocation.
- First, atomic-add the number of AQL packets to allocate to the write_index, then check if the queue is full. This can be more efficient when many unit of executions submit AQL packets, but it may be more difficult to manage gracefully when the queue is full.

Alternatively, an HSA application may limit the job submission to one agent unit of execution only ("Single submitting agent"), allowing the agent and packet processor to leverage this property and streamline the job submission process for such a queue.

The submission process can be simplified for queues that are only submitted to by a single agent:

- The increment of the write_index value by the job-submitting agent can be done using an atomic store, as it is the only agent permitted to update the value.
- The submitting agent may use a private variable to hold a copy of the write_index value and can assume no one else will modify the write_index to avoid reading the write_index.

Optionally the queue can be initialized to only support single-producer queue mechanics. Some HSA implementations may implement optimized packet processor behavior for single-producer queue mechanics.

Some HSA implementations may implement the same packet processor behavior for both the multi-producer queue mechanics and the single-producer queue mechanics case. Packet processor behavior can be optimized as long as it complies with the specification. For example, it is allowed for the packet processor in either case to:

- Use the packet format field to determine whether a packet has been submitted rather than reading the write_index queue field.
- Speculatively read multiple packets from the queue.
- Not update the read_index for every packet processed.

Similarly, agents that submit jobs do not need to read the read_index for every packet submitted.

The queue mechanics do not require packets to be submitted in order (except for queues restricted to single producer mechanics). However, using the doorbell, the packet processor latency can be optimized in the common case where packets are submitted in order. For this reason it is recommended that the doorbell is always signaled in order from within a single unit of execution (this is a strict requirement for queues using single producer mechanics), and that excessive contention between different unit of executions on a queue is avoided.

2.8.5 Queue index access

The queue write_index and read_index properties are accessed from HSA applications using HSA runtime API calls and from HSAIL kernels using specific instructions. The operations provided through the HSA runtime API and specific HSAIL instructions for accessing the queue write_index and read_index properties are equivalent. The operations are:

- Load the queue write_index property. Returns the 64-bit write_index value and takes the following parameters:
  - Pointer to the queue structure.
- Store new value for the write_index property. No return value, takes the following parameters:
  - Pointer to the queue structure.
  - New 64-bit value to store in the write_index property.
- Compare-and-swap operation on the write_index property. The compare-and-swap operation updates the write_index property if the original value matches the comparison parameter. Returns the original value of the write_index property and takes the following parameters:
  - Pointer to the queue structure.

    - 64-bit comparison value
    - New 64-bit value to store in the write_index property if comparison value matches original value.
- Add operation on the write_index property. Returns the original value of the write_index property and takes the following parameters:
    - Pointer to the queue structure.
    - 64-bit increment value
- Load the read_index property. Returns the 64-bit read_index value and takes the following parameters:
    - Pointer to the queue structure.
- Store new value for the read_index property. No return value, takes the following parameters:
    - Pointer to the queue structure.
    - New 64-bit value to store in the read_index property.

The following memory ordering options are required for the different operations:

- Load of read_index or write_index:
    - Relaxed atomic load with system scope
    - Atomic load acquire with system scope
- Store of read_index or write_index:
    - Relaxed atomic store with system scope
    - Atomic store release with system scope
- Add or compare-and-swap operation on write_index:
    - Relaxed atomic with system scope
    - Atomic acquire with system scope
    - Atomic release with system scope
    - Atomic acquire-release with system scope

All access operations for the queue indices are ordered in the memory model as if the queue indices are stored in the Global segment.

Agents other than host CPUs or kernel agents may access the write_index and read_index properties through implementation-specific means.

2.8.6 Services dispatch queue

An HSA system uses a queue-based mechanism to invoke defined system- and HSA services and functions via AGENT_DISPATCH packets issued to an appropriate queue.

This service queue is provided by the application to agents in an application specific way, for example as arguments to (agent) dispatch packets, or in a known memory location.

To invoke a service, or function in accordance to 2.9.7 Agent dispatch packet (on page 30) of this specification, the caller passes the arguments and function code in the AGENT_DISPATCH packet; the caller can monitor or wait on a completion signal to identify when the "call" has finished.

The caller must not assume the serviced function to be initiated or completed until completion is indicated through the AGENT_DISPATCH completion_signal (see 2.9.2 Packet process flow (on page 26)).

When the AGENT_DISPATCH is complete, the return_address referenced within the AGENT_DISPATCH packet will contain the function's return value.

The dispatch and packet processing flow of the AGENT_DISPATCH must follow all requirements set in 2.9 Requirement: Architected Queuing Language (AQL) (below) of this specification.

The caller must follow all documented function requirements of the called function, especially related to acquire & release fence scope and referenced memory.

The exact services provided by the services queue are defined by the application implementing the services queue.

2.9 Requirement: Architected Queuing Language (AQL)

An HSA-compliant system shall provide a command interface for the dispatch of agent commands. This command interface is provided by the Architected Queuing Language (AQL).

AQL allows agents to build and enqueue their own command packets, enabling fast, low-power dispatch. AQL also provides support for kernel agent queue submissions: the kernel agent *kernel* can write commands in AQL format.

AQL defines several different packet types:

- Vendor-specific packet
    - Packet format for vendor-specific packets is vendor defined apart from the format field. The behavior of the packet processor for a vendor-specific packet is implementation specific, but must not cause a privilege escalation or break out of the process context.
- Invalid packet
    - Packet format is set to invalid when the queue is initialized or the read_index is incremented, making the packet slot available to the agents.
- Kernel dispatch packet
    - Kernel dispatch packets contain jobs for the kernel agent and are created by agents. The queue features field indicates whether a queue supports kernel dispatch packets.
- Agent dispatch packet
    - Agent dispatch packets contain jobs for the agent and are created by agents. The queue features field indicates whether a queue supports agent dispatch packets.
- Barrier-AND packet
    - Barrier-AND packets can be inserted by agents to delay processing subsequent packets. All queues support barrier-AND packets.
- Barrier-OR packet
    - Barrier-OR packets can be inserted by agents to delay processing subsequent packets. All

queues support barrier-OR packets.

The packet type is determined by the format field in the AQL packet header.

2.9.1 Packet header

Table 2–4 Architected Queuing Language (AQL) packet header format

Bits	Field Name	Description
7:0	format	AQL_FORMAT: • 0=VENDOR_SPECIFIC • 1=INVALID • 2=KERNEL_DISPATCH • 3=BARRIER_AND • 4=AGENT_DISPATCH • 5=BARRIER_OR • others reserved
8	barrier:1	If set, then processing of the packet will only begin when all preceding packets are complete.
10:9	acquire_fence_scope	Determines the scope and type of the acquire memory fence operation for a packet.
12:11	release_fence_scope	Determines the scope and type of the memory fence operation applied after kernel completion but before the packet is completed.
15:13		Reserved, must be 0.

2.9.1.1 Acquire fences

The processing of the acquire fence differs between the Barrier packets and the Dispatch packets.

Barrier-OR and barrier-AND packet acquire fences are processed first in the completion phase of the packet, after the barrier condition has been met.

Kernel dispatch and agent dispatch packet acquire fence is applied at the end of the launch phase, just before the packet enters the active phase.

The packet acquire fence applies to the Global segment as well as any image data. The scope of the system that the fence applies to for the Global segment is configurable using the acquire_fence_scope header field.

See 2.9.2 Packet process flow (on the next page) for more details on the processing of the different packets.

The detailed behavior of the packet fences is described in 3.3.8 Packet processor fences (on page 54).

Note that to ensure visibility of a particular memory transaction both a release and a matching acquire fence may need to apply, as defined in the HSA Memory Consistency Model (see Chapter 3 HSA Memory Consistency Model (on page 47)). The required fences can be applied by instructions within HSA kernels or as part of an agent dispatch task as well as through the AQL packet processing.

Table 2–5 Encoding of acquire_fence_scope

acquire_ fence_ scope	Description
0	No fence is applied. The packet relies on an earlier acquire fence performed on the agent, or on acquire fences in the operation performed (e.g., by the HSAIL kernel).
1	The acquire fence is applied with agent scope for the global segment. The acquire fence also applies to image data.
2	The acquire fence is applied across both agent and system scope for the global segment. The acquire fence also applies to image data.
3	Reserved.

2.9.1.2 Release fences

The processing of the release fence differs between the Barrier packets and the Dispatch packets.

Barrier-OR and barrier-AND packet release fences are processed after the acquire fence in the completion phase of the packet.

Kernel dispatch and agent dispatch packet acquire release fences are applied at the start of the completion phase of the packet.

The packet release fence applies to the Global segment as well as read-write images modified by the previous completed packet executions. The scope of the system that the fence applies to for the Global segment is configurable using the release_fence_scope header field.

See 2.9.2 Packet process flow (below) for more details on the processing of the different packets.

The detailed behavior of the packet fences is described in 3.3.8 Packet processor fences (on page 54).

Table 2–6 Encoding of release_fence_scope

release_ fence_ scope	Description
0	No fence is applied. The packet relies on a later release fence performed on the agent, or on release fences in the operation performed (e.g., by the HSAIL kernel).
1	The release fence is applied with agent scope for the global segment. The release fence also applies to image data.
2	The release fence is applied across both agent and system scope for the global segment. The release fence also applies to image data.
3	Reserved.

2.9.2 Packet process flow

All packet types follow the same overall process. The packet processing is separated in launch, active, and completion phases:

Launch phase is initiated when launch conditions are met. Launch conditions are:

- All preceding packets in the queue must have completed their launch phase.
- If the barrier bit in the packet header is set then all preceding packets in the queue must have completed.

In the launch phase, an acquire memory fence is applied for kernel dispatch and agent dispatch packets before the packet enters the active phase. For barrier-OR and barrier-AND packets the acquire memory fence is applied later, in the completion phase.

After execution of the acquire fence (if applicable), the launch phase completes and the packet enters the active phase. In the active phase the behavior of packets differ depending on packet type:

- Kernel dispatch packets and agent dispatch packets execute on the kernel agent/agent, and the active phase ends when the task completes.
- Barrier-AND and barrier-OR packets remain in the active phase until their condition is met. Additionally no further packets are processed by the packet processor until the barrier-AND or barrier-OR packet has completed both its active and completion phase.

The active phase is followed by the completion phase.

If the packet is a barrier-AND or barrier-OR packet then an acquire memory fence is applied as the first step. For kernel dispatch and agent dispatch packets the acquire memory fence was already applied in the launch phase.

After execution of the acquire fence (if applicable), the memory release fence is applied.

After the memory release fence completes, the signal specified by the completion_signal field in the AQL packet is signaled with a decrementing atomic operation. NULL is a valid value for the completion_signal field, in which case no signal operation is performed.

2.9.3 Error handling

A queue can put into the error state by the following events:

- The packet processor detecting an error during a packet launch or completion phase.
- A packet detecting an error during its active phase.
- Calling the HSA runtime queue inactivate routine.
- Calling the HSA runtime queue destroy routine.

When a queue transitions into the error state the following sequence is followed:

1. The queue will be set into an error state.
2. If the packet processor detects an error during the launch phase, no further processing of the packet occurs.
3. If the packet processor is processing a packet for a queue in the launch phase when an error occurs for the same queue from some other source, it is implementation specific if the packet enters the active phase or if the packet is not processed.
4. No further packets will enter the launch phase for the queues that enter the error state [after reasonable finite time].
5. Packets in other queues that are not in the error state within the same process (same PASID) will still be processed.
6. If the queue has a callback registered, and the error is not the result of the HSA runtime queue inactivate or destroy call, it will be invoked in an HSA runtime thread. The callback will be given the identity of the queue that has entered the error state, and the queue status code.

7. Any packets in the active phase executing on the queue that has entered the error state will continue execution until they complete, encounter an error, or are terminated. It is implementation specific if the packet processor will terminate any packets prior to the HSA runtime queue inactivate or destroy routines being called. It is implementation specific if a packet executes the completion phase after a queue has entered the error state. Except that the completion phase is not executed if an error occurs during the execution of a packet, or if the packet is terminated.
8. A queue already in the error state does not invoke the callback, regardless of whether additional errors are detected.

The HSA runtime queue inactivate routine can be used to ensure no packets on a queue are in the active phase. It performs the following sequence:

1. If the queue is not in the error state, it is put in the error state without calling the queue callback. No further packets on the queue will be processed by the packet processor.
2. In reasonable time, it is guaranteed that no packets on the queue are in the active phase. An implementation may stop launching new wavefronts, may wait for wavefronts to complete, and may terminate wavefronts that do not complete within some grace period. It is implementation dependent if packets that leave the active phase perform the completion phase. Except that the completion phase is not executed if any wavefront of the packet reports an error or is terminated.
3. Ensure that the queue callback will not be executed once the queue is in the inactive state. If the queue callback is already executing, wait for it to complete. Otherwise, discard any pending queue errors.
4. Perform a system scope acquire fence that synchronizes with a packet processor system release fence that is applied for the queue before returning.

The HSA runtime queue destroy operation performs the following sequence:

1. Perform an implicit queue inactivate. This ensures no packets are in the active phase, and all memory is synchronized with the caller.
2. Ensure that the queue callback will not be executed once the queue is in the inactive state. If the queue callback is already executing, wait for it to complete. Otherwise, discard any pending queue errors.
3. Destroy the queue.

The queue callback may not invoke the HSA runtime queue destroy routines as it will block waiting for the queue callback to complete.

If an agent packet processor cannot determine the queue responsible for an error it may choose to put all queues into an error state. If an agent packet processor cannot recover from an error it detects, it may choose to put all queues into an error state, terminate all packets in the active phase, and reset the agent. Even if these occur the same rules for invoking the queue callback for each queue transitioned into the error state must be followed.

In order for applications to recover a queue that has entered the error state, it must destroy and re-create the queue.

Depending on the error type the process state may be corrupted resulting in undefined behavior and process termination may be necessary. For example, a segmentation fault may occur after corrupting other process memory. Such errors are termed fatal errors. It is implementation specific if a fatal error results in process termination, or ensures a queue enters the error state. If a fatal error does not terminate the process, then performing a queue deactivate or destroy on a queue that has received a fatal error, regardless of whether it was reported to the queue callback, results in returning a fatal status code.

In all cases no effects outside the process context should be visible from an error.

The following classes of errors must never cause a fatal error:

- Numerical error (NAN, DIV0, ...)
- Queue Termination
- Debug/Assert Breaks
- "Out of Memory" errors (e.g. including creating signals, queues,...)

2.9.4 Vendor-specific packet

A vendor-specific packet has the format field in the packet header set to VENDOR_SPECIFIC (e.g., zero). All other fields of the packet are implementation specific.

2.9.5 Invalid packet

An invalid packet has the format field in the packet header set to INVALID (e.g., one). All other fields of the packet are don't-care.

2.9.6 Kernel dispatch packet

Table 2–7 Architected Queuing Language (AQL) kernel dispatch packet format

Bits	Field Name	Description
15:0	header	Packet header, see 2.9.1 Packet header (on page 25).
17:16	dimensions	Number of dimensions specified in gridSize. Valid values are 1, 2, or 3.
31:18		Reserved, must be 0.
47:32	workgroup_size_x	x dimension of work-group (measured in work-items).
63:48	workgroup_size_y	y dimension of work-group (measured in work-items).
79:64	workgroup_size_z	z dimension of work-group (measured in work-items).
95:80		Reserved, must be 0.
127:96	grid_size_x	x dimension of grid (measured in work-items).
159:128	grid_size_y	y dimension of grid (measured in work-items).
191:160	grid_size_z	z dimension of grid (measured in work-items).
223:192	private_segment_size_bytes	Total size in bytes of private memory allocation request (per work-item).
255:224	group_segment_size_bytes	Total size in bytes of group memory allocation request (per work-group).
319:256	kernel_object	Handle for an object in memory that includes an implementation-defined executable ISA image for the kernel.
383:320	kernarg_address	Address of memory containing kernel arguments.
447:384		Reserved, must be 0.
511:448	completion_signal	HSA signaling object handle used to indicate completion of the job.

The kernarg_address minimum alignment granule is 16 bytes. HSAIL directives may enforce larger alignment granules for individual kernels. The memory at kernarg_address must remain allocated and not be modified until the kernel completes.

The memory referenced by the kernarg_address must be specifically allocated for this purpose through the HSA runtime API provided for this purpose, and must be visible at system scope before the kernel dispatch packet is assigned.

Kernel agents may implement a limitation on the size of the kernarg segment visible to a kernel dispatch. The minimum required size of the kernarg segment visible to a kernel dispatch is 4 kB.

The kernarg_address memory buffer should be allocated through the runtime allocation API provided for this purpose.

2.9.7 Agent dispatch packet

Table 2–8 Architected Queuing Language (AQL) agent dispatch packet format

Bits	Field Name	Description
15:0	header	Packet header, see 2.9.1 Packet header (on page 25).
31:16	type	The function to be performed by the destination agent. The function codes are application defined.
63:32		Reserved, must be 0.
127:64	return_address	Pointer to location to store the function return value(s) in.
191:128	arg0	64-bit arguments, may be values or pointers.
255:192	arg1	
319:256	arg2	
383:320	arg3	
447:384		Reserved, must be 0.
511:448	completion_signal	HSA signaling object handle used to indicate completion of the job.

2.9.8 Barrier-AND packet

Table 2–9 Architected Queuing Language (AQL) barrier-AND packet format

Bits	Field Name	Description
15:0	header	Packet header, see 2.9.1 Packet header (on page 25).
63:16		Reserved, must be 0.
127:64	dep_signal0	Handles for dependent signaling objects to be evaluated by the packet processor.
191:128	dep_signal1	
255:192	dep_signal2	
319:256	dep_signal3	
383:320	dep_signal4	
447:384		Reserved, must be 0.
511:448	completion_signal	HSA signaling object handle used to indicate completion of the job.

The barrier-AND packet defines dependencies for the packet processor to monitor. The packet processor will not launch any further packets until the barrier-AND packet is complete. Up to five dependent signals can be specified in the barrier-AND packet.

The execution phase for the barrier-AND packet is allowed to complete once all of the signals have passed through value zero at least once. The barrier-AND packet is guaranteed to complete only if all the signals' values become zero and remain zero until the barrier-AND packet completes.

The completion_signal will be signaled with an atomic decrementing operation in the completion phase.

The packet processor must guarantee individual forward progress for barrier-AND packets that are at the front of an AQL queue.

If a dependent signaling pointer has the value NULL it is considered to be constant 0 in the dependency condition evaluation.

2.9.9 Barrier-OR packet

Table 2–10 Architected Queuing Language (AQL) barrier-OR packet format

Bits	Field Name	Description
15:0	header	Packet header, see 2.9.1 Packet header (on page 25).
63:16		Reserved, must be 0.
127:64	dep_signal0	Handles for dependent signaling objects to be evaluated by the packet processor.
191:128	dep_signal1	
255:192	dep_signal2	
319:256	dep_signal3	
383:320	dep_signal4	
447:384		Reserved, must be 0.
511:448	completion_signal	HSA signaling object handle used to indicate completion of the job.

The barrier-OR packet defines dependencies for the packet processor to monitor. The packet processor will not launch any further packets until the barrier-OR packet is complete. Up to five dependent signals can be specified in the barrier-OR packet.

The execution phase for the barrier-OR packet is allowed to complete once one of the signals have passed through value zero at least once. The barrier-OR packet is guaranteed to complete only if at least one of the signals' values become zero and remain zero until the barrier-OR packet completes.

The completion_signal will be signaled with an atomic decrementing operation in the completion phase.

The packet processor must guarantee individual forward progress for barrier-OR packets that are at the front of an AQL queue.

If a dependent signaling pointer has the value NULL it is considered to be constant non-zero in the dependency condition evaluation.

2.9.10 Small machine model

The HSA small machine model has 32-bit rather than 64-bit native pointers. This affects any packet field that specifies an address:

- Kernel dispatch packet kernarg_address field
- Agent dispatch packet return_address field

For these fields only the least significant 32 bits are used in the small machine model.

2.10 Requirement: Agent scheduling

The HSA compliant system is required to schedule tasks for each agent from the HSA queues connected to the agent as well as from any additional pool of non-HSA tasks. The exact mechanism for scheduling is implementation-defined, but should meet the restrictions outlined in this section.

The HSA programming model encourages the use of multiple queues per application to enable complex sequences of tasks to be scheduled. It is expected that HSA-compliant applications use multiple queues simultaneously. To ensure consistent performance across HSA-compliant systems, it is recommended that when using multiple queues the scheduling algorithm will schedule efficiently across them and minimize the overhead for managing multiple queues.

To ensure this, the agent scheduling must have the following characteristics:

1. As a minimum, the agent scheduling should be triggered by the following events:
    - Agent task execution completes and the agent becomes available for executing another task.
    - If the agent is not executing a task then the agent scheduling must also be triggered by:
        - Packet submission to any queue connected to the agent.
        - A signal used by an AQL barrier-AND or barrier-OR packet active on a queue connected to the agent is signaled such that the AQL barrier-AND or barrier-OR packet can complete. The agent scheduling must minimally be able to monitor the forefront AQL packet being active on each queue connected to it. Monitoring includes any combination of supported AQL packets and their dependencies. To ensure deadlock is avoided, the monitoring of dependencies must not occupy any execution resources required to make forward progress on tasks (e.g., a barrier-AND or barrier-OR packet active on one queue must not block task execution on other queues).
2. The agent scheduling should respond to the events listed in point 1 in a finite time.
3. When the agent scheduling mechanism is triggered and the agent has available execution resources, the scheduling mechanism must dispatch a task if the oldest packet on any queue connected to the agent is a runnable kernel dispatch or agent dispatch packet. Note that barrier-AND and barrier-OR packets are not allowed to occupy agent/kernel agent execution resources.
4. An agent supporting full profile must appear to support at least the oldest packet for each connected queue to be in the active phase (see 2.9.2 Packet process flow (on page 26)) simultaneously. This may be implemented through use of context switching of the agent to allocate time-shares to active kernel dispatch or agent dispatch packets across multiple queues.
5. An agent supporting base profile is required to support at least the oldest packet for one connected queue in the system to be in the active phase (see 2.9.2 Packet process flow (on page 26)), but is not required to provide independent forward progress across multiple queues.

2.11 Requirement: Kernel dispatch forward progress

The following rules apply to the required forward progress of kernel dispatches:

- For a full profile agent, the incomplete work-group with the lowest flattened work-group ID for the oldest active dispatch of each connected queue must be active.
- For a base profile agent, the incomplete work-group with the lowest flattened work-group ID for the

oldest active dispatch of one connected queue must be active.

- Work-groups become active when they are started, or when a suspended work-group is resumed.
- Work-groups cease to be active when they complete, or when the work-group is suspended.
- A wavefront is active when the work-group it belongs to is active.
- Completion
    - A wavefront is complete after all work-items have executed their last instruction.
    - A work-group is complete after all the wavefronts of the work-group are complete.
    - The active phase of a kernel dispatch (see 2.9.2 Packet process flow (on page 26)) is complete when all work-groups of the kernel dispatch are complete.
- Among the wavefronts of the active work-groups, at least one non-waiting wavefront must make forward progress.
    - Wavefront waiting if:
        - Executing non-satisfied barrier
        - Executing non-satisfied fbarrier wait
        - Any lane is executing non-satisfied signal wait
    - Wavefront making forward progress if:
        - At least one work-item is executing instructions

The active work-groups of a kernel dispatch that is in the active phase may change during its execution as a result of agent scheduling and load balancing. The precise ordering of work-groups being started, completed, suspended, or resumed, and the consequential number of active work-groups, is considered platform-specific and is not mandated by the HSA specification; however, a minimum of active work-groups are defined above for dispatches in the active phase.

The requirements on which dispatches must be active are defined for full and basic profile in section 2.10 Requirement: Agent scheduling (on the previous page).

Flattened work-group ID and wavefront divergence/convergence are defined in the *HSA Programmer's Reference Manual Version 1.2*.

2.12 Requirement: Floating point support

Kernel agents shall support floating point operations based on the IEEE754-2008 standard. The specific floating point operations required in the HSAIL instruction set are defined in the *HSA Programmer's Reference Manual Version 1.2*.

2.13 Requirement: IEEE754-2008 floating point exceptions

A kernel agent shall report certain defined exceptions related to the execution of the HSAIL code to the HSA Runtime. At minimum the following exceptions shall be reported for IEEE758-2008 floating point errors:

Table 2–11 IEEE754-2008 exceptions

Operation	Exception Code	Description
INVALID_OPERATION	0	Invalid operation that raises a flag under default exception handling as specified in IEEE754-2008.
DIVIDE_BY_ZERO	1	Divide-by-zero that raises a flag under default exception handling as specified in IEEE754-2008.
OVERFLOW	2	Overflow that raises a flag under default exception handling as specified in IEEE754-2008.
UNDERFLOW	3	Underflow that raises a flag under default exception handling as specified in IEEE754-2008.
INEXACT	4	Inexact that raises a flag under default exception handling as specified in IEEE754-2008.

- DETECT:

  Support for DETECT is not required in the base HSA profile. Support for DETECT is required in the full HSA profile.

  The kernel agent must maintain a status bit for each of the DETECT-enabled arithmetic exceptions for each work-group. The status bits are reset to zero at the start of a dispatch. If an exception occurs and the DETECT policy is enabled for that exception, then the kernel agent will set the corresponding status bit. The kernel agent must:

  - Support HSAIL instructions that can read or write the exception status field while the kernel is running.

- BREAK:

  Support for BREAK is not required in the base HSA profile, and is optional in the full HSA profile.

  When an exception occurs and the BREAK policy is enabled for that exception, the kernel agent must stop the excepting wavefront's execution of the HSAIL instruction stream precisely at the instruction which generated the exception. Memory and register state is just before the excepting instruction executed, except that non-excepting lanes in the wavefront which generated the exception may have already executed and updated register or memory state. In reasonable time, the kernel agent executing the wavefront must stop initiating new wavefronts for all dispatches executing on the same User Mode queue, and must ensure that all wavefronts currently being executed for those dispatches either complete, or are halted. Any dispatches that complete will have their completion signal updated as normal, however, any dispatch that do not complete the execution of all their associated wavefronts will not have their completion signal updated.

  The kernel agent then signals the runtime, which invokes an exception handler defined in the HSA Runtime. The reported information must include:

  - Exception code
  - PASID
  - Queue id
  - packet ID
  - Precise instruction counter that caused the exception.

Some possible uses for the runtime exception handler are:

- Invoking a debugger which allows the developer to determine which function caused the exception and inspect the surrounding memory and register state.
- Logging the exception to a file.
- Killing the process.

After signaling the BREAK, it is not required that the kernel agent can restart the instruction which generated the exception.

The enabling of the DETECT and BREAK exception policies is controlled through the HSAIL finalization. Both DETECT and BREAK can be enabled for the same arithmetic exception.

The exception handling policy must be specified at the time the HSAIL code is finalized, since implementations may generate different code when exception handling is requested. Enabling either DETECT or BREAK functionality may reduce the performance of the kernel running on the kernel agent. However, it is recommended that performance with DETECT policy be close to native code performance since this is a commonly-used exception policy.

2.14 Requirement: Kernel agent hardware debug infrastructure

The kernel agent shall provide mechanisms to allow system software and some select application software (for example, debuggers and profilers) to set breakpoints and collect throughput information for profiling. At a minimum, instruction breakpoints shall be supported. Memory breakpoints, memory faults, and conditional breakpoints may be supported as an implementation option.

When hitting a breakpoint condition, the hardware shall retire all pending writes to memory, stop the wavefront, and provide status information indicating the breakpoint condition, the affected PASID, user-level queue, instruction(s), and wavefront/group ID.

System software shall provide appropriate resident memory to allow the kernel agent hardware on reaching the breakpoint condition to write out state information that can be parsed by appropriate select application software (for example, debuggers and profilers). The format of the written data shall be system implementation-specific.

2.15 Requirement: HSA platform topology discovery

2.15.1 Introduction

The HSA System Architecture requirements cover a wide range of systems with a variety of elements. In order to leverage the available elements to their fullest, application and system software need a way to discover and characterize the available agents, their topological relation to each other and to other relevant system resources like memory, caches, I/O, and host agents (e.g., CPU).

An HSA-compliant system contains a variety of different agents, interfaces, and other resources such as memory and caches. Application, runtime, and system software need a way to characterize all the available resources in order to use them.

At minimum, an HSA compliant system must provide a set of parameter characterizations that are made available to HSA software. Applications can discover these through various runtime APIs (through the HSA Runtime, with the expectation that other APIs targeting the platform, e.g., OpenCL™ and DirectCompute, report equivalent information) and software may leverage these parameters for optimizing performance on a specific platform.

Unless otherwise provided to the HSA runtime level by system software, each of these properties shall be provided through appropriate parameter storage retrieval; an implementation may use firmware table entries, data retrieved through firmware methods (e.g., via ACPI), kernel image or through equivalent mechanisms. The multitude of kernel agent properties can be enumerated and their relationships to each other established by the system software parsing the data and communicated to the HSA runtime and HSA applications.

Depending on the platform, the data should be grouped hierarchically to correspond to components such as Agent, Memory, Compute Properties, Caches, and I/O.

The data are associated with a node ID, forming a per-node element list which references the elements contained at relative offsets within that list. A node associates with a kernel agent or agent. Node ID's should be 0-based, with the "0" ID representing the primary elements of the system (e.g., "boot cores", memory) if applicable. The enumeration order and—if applicable—values of the ID should match other information reported through mechanisms outside of the scope of the HSA requirements; for example, the data and enumeration order contained in the ACPI SRAT table on some systems should match the memory order and properties reported through HSA. Further detail is out of the scope of the System Architecture and outlined in the Runtime API specification.

Each of these nodes is interconnected with other nodes in more advanced systems to the level necessary to adequately describe the HSA system topology.

Where applicable, the node grouping of physical memory follows NUMA principles to leverage memory locality in software when multiple physical memory blocks are available in the system and agents have a different "access cost" (e.g., bandwidth/latency) to that memory.

Figure 2–2 (on page 38) provides an illustrative example for a grouping characterizing an exemplary multi-socket CPU/APU and multi-board GPU system, whereas Figure 2–1 (on the facing page) is an example for a structurally simpler platform with only one "node," containing agent and memory resources.

Figure 2–1 Example of a simple HSA platform

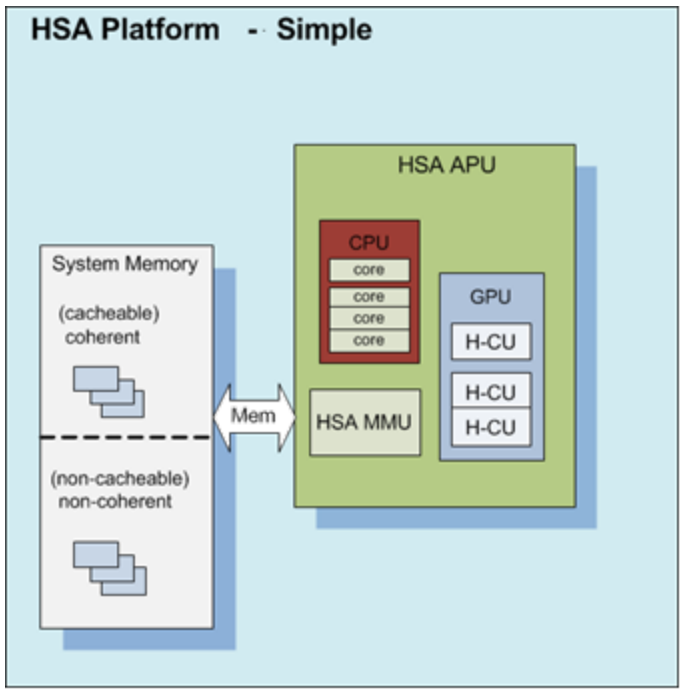
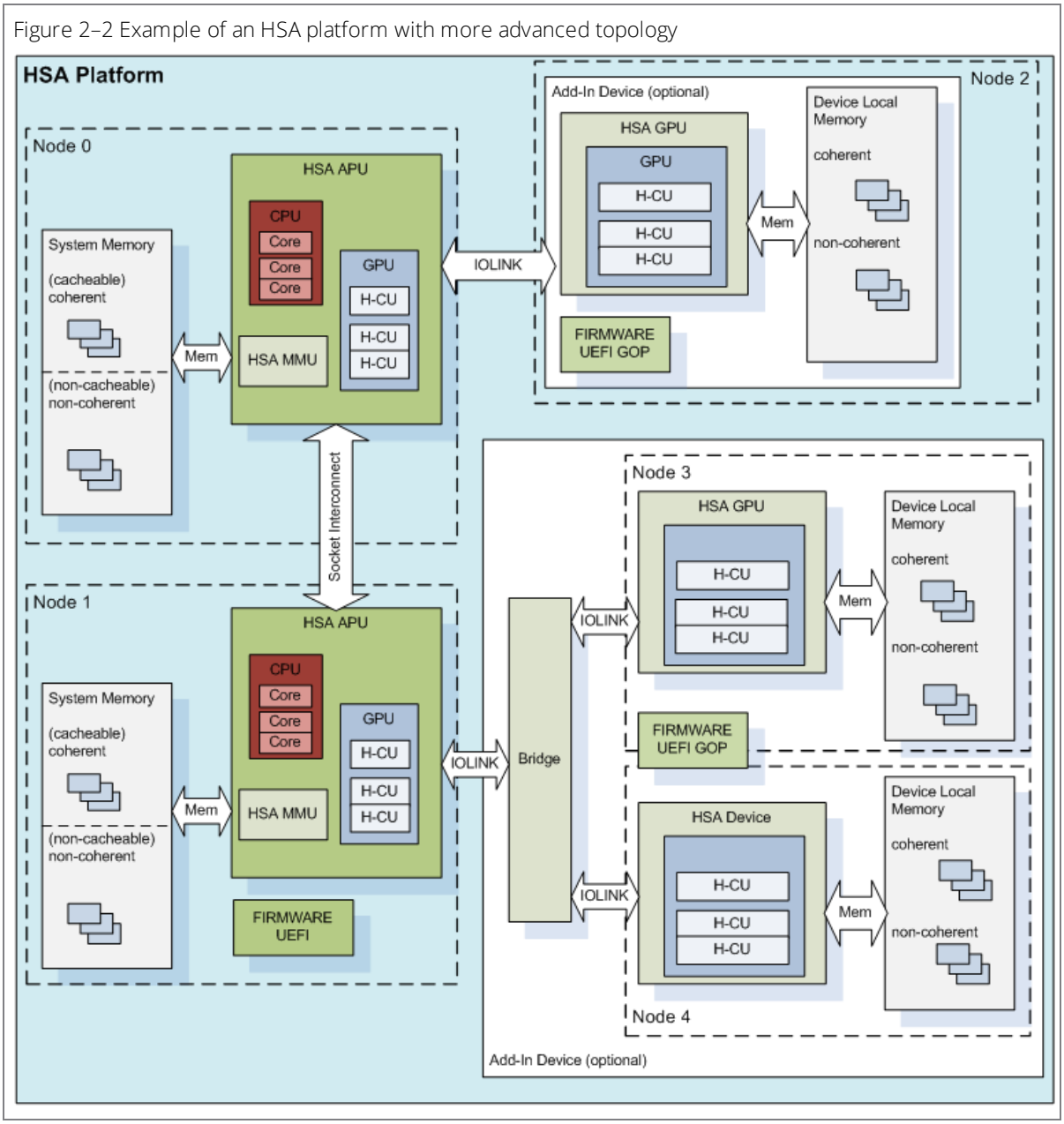


Figure 2–2 (on the next page) shows a more advanced system with a variety of host CPUs and kernel agents and five discoverable HSA Memory Nodes, distributed across multiple system components.

Figure 2–2 Example of an HSA platform with more advanced topology



2.15.2 Topology requirements

The platform topology is an inherent property of the HSA system. From the application point of view, the topology can be fully enumerated and should not change during the application enumeration process itself, but application may receive notification changes after.

For advanced systems that support docking or dynamically attaching or removing components (“docking”), the system must provide properties to detect ephemeral vs persistent components and use notification mechanisms to the HSA runtime and application that allow them to re-enumerate the updated topology. Detection of topology changes during enumeration should be deferred until the enumeration of the previous topology completes to avoid undefined or transitory topology state in application data structures.

Typically, though not in all cases system software may enumerate all HSA system topology information at or close to system startup and makes it available at application startup. System software and runtime may provide mechanisms to the application to notify it about topology changes that may be caused by platform & system events (e.g., docking) that change the previously reported topology.

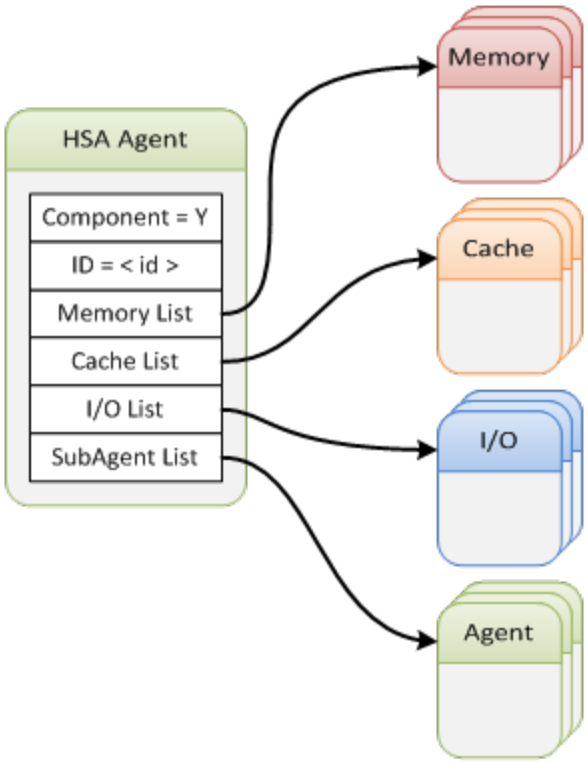
The detail notification mechanisms may be system and vendor specific and are outside of the scope of this specification.

Each of the data parameters characterizes particular kernel agent properties of the system. In addition agent entries may reference elements that characterize the resources the kernel agents are associated with. Each of the topology elements is characterized by a unique element ID and type-specific property data.

The following sections will describe the element requirements in more detail.

2.15.3 Agent & kernel agent entry

Figure 2–3 General structure of an agent entry



This entry characterizes the hardware component participating in the HSA memory model.

The entry should make a distinction between Agents (= can issue AQL packets into a queue) and kernel agents (= can issue and consume AQL packet from a queue), but otherwise provide similar detail of properties for each.

Each such topology entry for an agent/kernel agent is identified by a unique ID on the platform. The ID is used by application and system software to identify agents & kernel agents. Each topology entry also contains an agent/kernel agent type (e.g., throughput/vector, latency/scalar, DSP, ...) characterizing its compute execution properties, work-group and wavefront size, concurrent number of wavefronts, in addition to a nodeID value.

One or more agents/kernel agents may belong to the same nodeID or some agents/kernel agents may be represented in more than one node, depending on the system configuration. This information should be equivalent or complementary to existing information on the platform (e.g., derived from ACPI SRAT or equivalent where applicable).

An agent/kernel agent parameter set may reference other system resources associated with it to express their topological relationship as needed; examples are other agents/kernel agents, memory, cache, and IO resources, or sub-divisible components of the agent/kernel agent itself.

The goal is to allow HSA compliant system software and HSA applications to identify execution blocks in a hierarchy, within an agent/kernel agent, to be individually scheduled and addressed; for example, in a PC platform, an agent (like a CPU) can be expressed in units such as “sockets” and cores, each socket with its memory access path and cache hierarchy representing a “node”, whereas the cores represent individually scheduled execution elements represented as agents.

At minimum, the following agent/kernel agent parameters should be available to system software and to applications via application programming interfaces (APIs) (e.g., the HSA runtime, OpenCL™, or others):

- “Kernel agent” or “Agent” Indicator (see above)
- Type (throughput/vector, latency/scalar, DSP, ...)
- Node ID the element belongs to.
- Element ID. Used to individually identify the Agent on that platform. Existing values inherent in the platform definition may be used for that purpose. There is no requirement for sequentiality.
- Maximum number of Compute Units
- Maximum work-item dimensions
- Maximum work-item sizes
- Machine model
- Endianness
- Maximum and minimum queue sizes, always a power of 2 and a multiple of the AQL packet size

Optional properties of an agent would also surface through the mechanism, as an example:

- Maximum Clock Frequency in MHz (0 if not exposed on a platform)
- Friendly name (equivalent to “device name” in OpenCL™)
- Floating point capabilities

Other optional parameters may be available depending on the platform. Any optional parameter not defined by the platform should be returned as “0”.

2.15.4 Memory entry

The HSA Memory Entry describes the physical properties of a block memory in the system with identifiable characteristics that can be accessed via HSA memory model semantics by the agents.

At minimum, the following HSA memory parameters of each physically distinguishable memory block would be made available to system software and potentially made available to applications via APIs (e.g., the HSA runtime, OpenCL™, or others):

- Node ID the element belongs to.
- Element ID. Used to individually identify the memory block on that platform. Existing values inherent in the platform definition may be used for that purpose. There is no requirement for sequentiality.
- Virtual base address (32-bit or 64-bit value)
- Physical memory size in bytes

Optional HSA memory parameters would also be made available through the mechanism, as an example:

- Memory interleave (1, 2, 4)
- Number of memory channels
- Memory width per memory channel
- Maximum memory throughput in MByte/s
- Minimum memory access latency in picoseconds

Other optional parameters may be available depending on platform. Any optional parameter not defined by the platform should be returned as "0".

2.15.5 Cache entry

The HSA Cache Entry describes the available caches for agents within a system.

At minimum, the following HSA memory parameters would be made available to system software and potentially made available to applications via APIs (e.g., the HSA runtime, OpenCL™, or others):

- Node ID the element belongs to.
- Element ID. Used to individually identify the memory block on that platform. Existing values inherent in the platform definition may be used for that purpose. There is no requirement for sequentiality.
- Cache type, e.g., read-only or read/write cache
- Total size of cache in kilobytes
- Cache line size in bytes
- Cache level (1, 2, 3, ...)
- Associativity (0 = reserved, 1= direct mapped, 2=0xFE associativity, 0xFFh=fully associative)
- Exclusive/inclusive to lower level caches
- Agent association if cache is shared between agents (identified via nodemask)

Optional HSA memory parameters would also be made available through the mechanism, as an example:

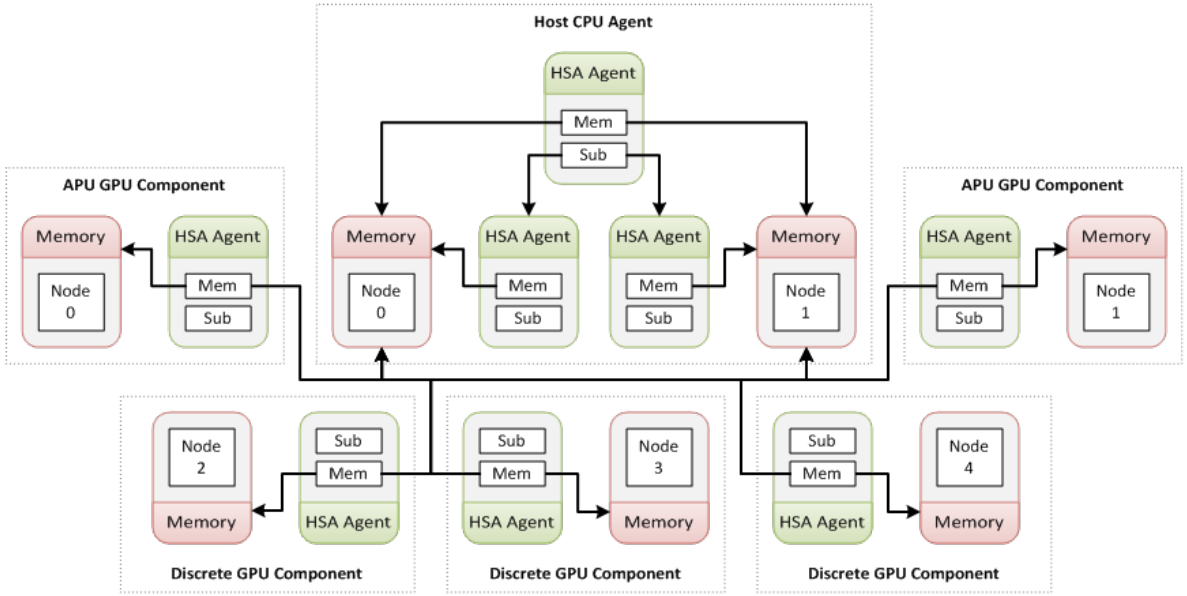
- Maximum latency on a cache hit (in picoseconds)

Other optional parameters may be available depending on platform. Any optional parameter not defined by the platform should be returned as "0".

2.15.6 Topology structure example

Figure 2–4 (on the next page) outlines an illustrative example of how the system information can be used to express specific system topology, using the Figure 2–2 (on page 38) block diagram as the basis.

Figure 2–4 Topology definition structure for previously defined system block diagram



2.16 Requirement: Images

An HSA-compliant platform shall optionally provide for the ability of HSA software to define and use image objects, which are used to store one-, two-, or three-dimensional images.

The elements of an image object are defined from a list of predefined architected or vendor-specific image geometries, formats, and data layouts. Image primitives are accessible from kernel agents and operate on an image value, which is referenced using an opaque 64-bit image handle.

Image handles are created by the HSA runtime. An image object can be read-only, i.e., a kernel agent can only read values from the image, write-only, i.e., a kernel agent can only write values to the image, or read and write, i.e., a kernel agent can write to an image as well as read from it.

A kernel agent may choose to not support image objects.

The image *geometry* refers to the logical organization of elements in the image, e.g., the number of dimensions of the image and whether it is an image array.

The image *size* refers to the size of each image dimension and the number of layers for an image array.

The image *format* refers to the logical organization of each element of the image, e.g., the number, order, and type of channel components for each element in the image.

The image *data layout* refers to how the image is stored in memory.

If images are supported, then there are mandatory requirements on support for image geometries, formats, and data layouts:

- A set of mandatory image geometries and formats must be supported, but the image data layout may be implementation specific (opaque). The mandatory image geometries and formats are further detailed in the *HSA Programmer's Reference Manual*.
- In addition, the “1DB” image format must support the “linear” image data layout. The “1DB” image format and the “linear” data layout are further described in the *HSA Programmer's Reference Manual*.

Images have the following additional properties:

- Image objects are created by the HSA platform through a runtime call with a specified global segment memory buffer for image data storage.
    - An image object may be used only by the agent for which it is created.
    - The same global segment memory buffer may be used to create image objects for multiple agents.
    - Multiple image objects for the same agent may alias to the same image storage, this can be used to alias read-only, write-only and read-write images with each other, if the different access modes are supported by the chosen image data layout.
    - Ownership assignment operations are allowed on the image data storage for images. Note that changes in ownership of the image data storage are considered modifications of the image data and must be performed in accordance with the rules for image data visibility defined below.
    - The data layout of the image data storage for an image object must be one of the architected or vendor-specific image data layouts supported by the agent for which the image object is created.
    - There is an optionally supported architected opaque image data layout. The opaque image data layout has the following properties:
        - The opaque image data layout is not guaranteed to be consistent across different agents. When using the opaque image data layout the image data storage cannot be shared across image objects for different agents.
        - The opaque image data layout is guaranteed to be consistent across all image objects on a single agent that correctly alias to the same image storage. The constraints on image object properties that must be met to correctly alias multiple image objects to the same image storage is described in the *HSA Programmer's Reference Manual.*
- Image operations do not contribute to memory orderings defined by the HSA memory model. The following rules apply for image operation visibility:
    - Modifications to image data through HSA runtime API require the following sequence to occur in order to be visible to subsequent AQL packet dispatches:
        a. HSA runtime operation modifying the image data completes. The HSA runtime operation implies a release operation for the image data.
        b. A packet processor acquire fence applying to image data is executed on the agent reading the image (this may be part of the AQL packet dispatch that reads the modified image data) and synchronizes with the earlier implicit release operation.
    - Modifications to the image data storage of an image by an agent through memory operations require the following sequence to occur in order to be visible to subsequent AQL packet dispatches:
        a. After the image data storage modifications are complete, the modifying agent performs a release operation with the appropriate scope.
        b. A packet processor acquire fence applying to image data is executed on the agent

reading the image (this may be part of the AQL packet dispatch that reads the modified image data) and synchronizes with the earlier release operation.

- Modifications to image data by a kernel agent packet dispatch are visible as follows:
  - Image data modifications made by an AQL packet dispatch through image operations require the following sequence to occur in order to be visible to subsequent AQL packet dispatches:
    a. The active phase of the AQL packet dispatch modifying the image completes.
    b. A packet processor release fence applying to image data is executed on the agent modifying the image (this may be part of the AQL packet dispatch that modified the image).
    c. A packet processor acquire fence applying to image data is executed on the agent reading the image (this may be part of the AQL packet dispatch that reads the modified image data) and synchronizes with the earlier AQL packet release fence.
  - Image data modifications made by an AQL packet dispatch through image operations require the following sequence to occur in order to be visible to HSA runtime operations:
    a. The active phase of the AQL packet dispatch modifying the image completes.
    b. A packet processor release fence applying to image data is executed on the agent for the image (this may be part of the AQL packet dispatch that modified the image).
    c. The HSA runtime operation is performed. The HSA runtime operation implies an acquire operation for the image data and must synchronize with the earlier AQL packet release fence.
  - Image data modifications made by an AQL packet dispatch through image operations require the following sequence to occur in order to be visible to memory operations by an agent:
    a. The active phase of the AQL packet dispatch modifying the image completes.
    b. A packet processor release fence applying to image data is executed on the agent modifying the image (this may be part of the AQL dispatch that modified the image).
    c. An acquire operation is executed on the agent reading the image and synchronizes with the earlier AQL packet release fence.
  - Image store operations by a single work-item are visible to image load operations by the same work-item after execution of a work-item scope image fence operation.
  - Image store operations by a work-item are visible to image load operations by the same work-item and other work-items in the same work-group after both the writing and reading work-item(s) execute work-group execution uniform image acquire/release fences at work-group scope.

        - Image fence operations do not affect read-only images, the only fences that can specify an ordering between image reads from a read-only image and any modification of the read-only image is a packet processor acquire fence on the agent using the read-only image and an appropriate synchronizing release fence.
        - Note that there is no image acquire/release fence at agent scope. Therefore, it is not possible to make image stores performed by a work-item visible to image loads performed by another work-item in a different work-group of the same dispatch.
        - The only mechanism for a kernel agent to specify an ordering between image operations and memory operations is packet processor fences.
        - Image fences and `barrier/wavebarrier` operations cannot be reordered. The ordering between `barrier/wavebarrier` operations and image fences is visible to all work-items that participate in the `barrier/wavebarrier` operations. See the *HSA Programmer's Reference Manual Version 1.2* for more information on the `barrier/wavebarrier` operations.
        - Image accesses and image fences by a single work-item cannot be reordered.
    - Access to modified image data that is not yet visible yields undefined values.
    - Image loads and stores using at least one undefined coordinate cause the program to be undefined.
- Image and sampler object creation must occur through the HSA runtime image/sampler handle creation API before a kernel that uses the image/sampler objects is dispatched. This can be ensured by ordering the API calls to create the image/sampler objects before the release to update the AQL packet format field to a valid packet type. It is the responsibility of the HSA Runtime image/sampler handle creation API to make any implicit memory required for descriptors visible to the agent for which the handle is created. Apart from memory release/acquire operations required to communicate the image/sampler handle to the kernel no other synchronization is required.

As described above, image operations are not included in the HSA Memory Model, as defined in Chapter 3 HSA Memory Consistency Model (on page 47). Interoperation among image operations and the memory model is only possible when the visibility rules defined above are followed.

2.17 Requirement: Profiling

An HSA-compliant platform shall optionally provide for the ability of profiling the performance of HSA software.

2.17.1 Profiling Events extension

An HSA-compliant platform shall optionally provide for the ability of HSA software to define, produce, and read timeline events. Events can be produced by the HSA Runtime API, HSA topology nodes, applications that call into the HSA runtime, and HSAIL kernels. Event types can be registered through the runtime API and triggered from the calling application or an HSAIL kernel with optional metadata. Events have the following additional requirements:

- The HSA platform must be able to provide a single stream of events to an application that requests them.
    - To carry this out, it may need to merge events from multiple sources into a single buffer or

balance the provision of events from different sources.

- There is no requirement that events must be provided to the application in the order in which they are generated.
- There must be a method by which an event produced by HSAIL can by made available to the HSA Runtime API along with any metadata associated with the event.

2.17.2 Read performance counters extension

An HSA-compliant platform shall optionally provide for the ability of HSA software to read implementation-defined performance counters. Performance counter reading has the following additional requirements:

- The HSA platform must be able to communicate which sets of performance counters can be counting at any given time.
- There is no set of required performance counters that must be available; an implementation can expose whatever counters it likes with implementation-defined semantics.

CHAPTER 3.
HSA Memory Consistency Model

3.1 What is a memory consistency model?

A memory consistency model defines how writes by one unit of execution become visible to another unit of execution, whether on the same agent or different agents. The memory consistency model described in this chapter sets a minimum bar. Some HSAIL implementations might provide a stronger guarantee than required.

For many implementations, better performance will result if either the hardware or the finalizer is allowed to relax the perceived order of the execution of memory operations. For example, the finalizer might find it more efficient if a write is moved later in the program; so long as the program semantics do not change, the finalizer is free to do so. Once a store is deferred, other work-items and agents will not see it until the store actually happens. Hardware might provide a cache that also defers writes.

In the same way, both the hardware and the finalizer are sometimes allowed to move writes to memory earlier in the program. In that case, the other work-items and agents may see the write when it occurs, before it appears to be executed in the source.

A memory consistency model provides answers to two kinds of questions. For developers and tools, the question is: "How do I make it always get the right answer?" For implementations, the question is: "If I do this, is it allowed?"

Programs cannot see any global order directly. Implementations cannot know about future loads and stores. Each implementation creates an order such that any possible visibility order requirement created by future loads and stores will not be violated.

Implementations can constrain events more than the model allows. The model is a minimum required ordering of events. The model goes from a description of operations, to a minimal required ordering of operations due to a single compute unit, to an interleaving of the operations in time from multiple compute units with allowed reordering.

Legal results are determined by the interleavings, allowed reorderings, and visibility rules.

3.2 What is an HSA memory consistency model?

The HSA memory consistency model describes how multiple work-items interact with themselves and with agents.

The HSA memory consistency model is a relaxed model based on a data race free framework.

The memory model defines the concept of well-defined and undefined executions, and a program has undefined behavior if it has any execution that is undefined.

The execution of any HSA-race-free program that does not include a relaxed atomic operation appears sequentially consistent.

The execution of any HSA-race-free program that includes one or more relaxed atomic operations may not appear sequentially consistent.

HSA includes concepts of memory segments and scopes.

Each load or store operation occurs to a single memory segment, whereas fences apply across multiple segments. For special memory operations we define a set of scopes it applies to (namely work-item, wavefront, work-group, agent, and system).

The precise rules and definitions for the memory consistency model are described in the rest of the HSA Memory Consistency Model section.

There might be devices in a system that do not subscribe to the HSA memory consistency model, but can move data in and out of HSA memory regions.

Software can only see the results of load and read-modify-write operations. Each load result creates a more constrained partial visibility order. The contract is violated when a load gets a result that does not meet all the required partial visibility orders in the system.

3.3 HSA memory consistency model definitions

The HSA memory consistency model is defined in terms of units of executions and the ordering of visibility of operations.

- A *unit of execution* is a program-ordered sequence of operations through a processing element. A unit of execution can be any thread of execution on an agent, a work-item, or any method of sending operations through a processing element in an HSA-compatible device.
- For primitives of type “store”, visibility to unit of execution A of a store operation X is when the data of store X is available to loads from unit of execution A.
- For primitives of type “load”, visibility is when a load gets the data that the unit of execution will put in the register.
- Primitives of type “fence” can enforce additional constraints on the visibility of load and store operations. A fence is visible at the point where the constraint implied by the fence is met. For example, a release fence is visible when all preceding transactions from units of execution affected by the fence are visible in the required scope instances, and the constraints versus other fences and atomic operations are also met.
- Definition of single-copy atomic property.
- If two single-copy atomic stores overlap, then all the writes of one of the stores will be inserted into the coherent order (see 3.7 Coherent order (on page 55)) of each overlapping byte before any of the writes of the other store are inserted.
- If a single-copy atomic load A overlaps with a single-copy atomic store B and if for any of the overlapping bytes the load returns the data written by the write B inserted into the coherent order of that byte by the single-copy atomic store then the load A must return data from a point in the coherent order of each of the overlapping bytes no earlier than the write to that byte inserted into the coherent order by single-copy atomic store B.
- “Undefined program or execution”. Some incorrect programs can exhibit implementation specific behavior. These programs or the execution of these programs are typically labeled “undefined”. However, possible behaviors of “undefined programs” do not include escaping process isolation or escalation of the privilege level of the process. The behavior potentially exhibited by an undefined program must be consistent with its privilege level. Some examples of permissible behavior for

undefined programs are to complete with an incorrect result, fail with segment violation fault or other fault, and so on.

3.3.1 Operations

- A "memory operation" is any operation that implies executing one or more of the load, store, or fence primitives. The following basic operations are defined:
  - `ld`, load a value, also referred to as "ordinary load".
  - `st`, store a value, also referred to as "ordinary store".
  - `Atomic_op`, perform an atomic operation op returning a value. This may be a load (`ld`) or a read-modify-write operation (`rmw`).
  - `atomicNoRet_op`, perform an atomic operation without returning a value. This may be a store (`st`) or a read-modify-write operation (`rmw`).
  - `fence`, perform a memory fence operation.
- Memory operation order constraints are attached to memory operations. The following ordering constraints are defined:
  - `rlx`, no ordering constraint attached (i.e., relaxed).
  - `scacq`, acquire constraint attached[1].
  - `screl`, release constraint attached[2].
  - `scar`, acquire and release constraints attached[3].
- The only ordering constraint allowed for ordinary loads or stores is rlx.
- The following combinations of `atomic` operation and ordering constraint are allowed:
  - `atomic_ld_rlx`, `atomic_ld_scacq`,
  - `atomic_rmw_rlx`, `atomic_rmw_scacq`, `atomic_rmw_screl`, `atomic_rmw_scar`
- The following combinations of `atomicnoret` operation and ordering constraint are allowed:
  - `atomicnoret_st_rlx`, `atomicnoret_st_screl`,
  - `atomicnoret_rmw_rlx`, `atomicnoret_rmw_scacq`, `atomicnoret_rmw_screl`, `atomicnoret_rmw_scar`
- The following combinations of fence operation and ordering constraint are allowed: `fence_scacq`, `fence_screl`, `fence_scar`.
- Atomic stores are any atomic operations that imply modification of a memory location.
- Atomic loads are any atomic operations that imply reading a memory location.
- Atomic read-modify-writes are any operations that imply both reading and modifying a memory

[1] `scacq` ordering is similar to `seq_cst` ordering for a load in C++11, not acquire ordering.

[2] `screl` ordering is similar to `seq_cst` ordering for a store in C++11, not release ordering.

[3] `scar` ordering is similar to `seq_cst` ordering for a read-modify-write in C++11.

location.

- An ordinary memory operation is an ordinary load or an ordinary store (`ld` or `st`)
- Release operations are all memory operations (atomic or fence) with order `screl` or `scar`.
- Acquire operations are all memory operations (atomic or fence) with order `scacq` or `scar`.
- Synchronizing operations are the union of all release and all acquire operations, i.e., all operations (atomic or fence) with order `scacq`, `screl`, or `scar`.
- Special operations are all atomic operations and all fence operations.

The operations available in HSAIL have different names, and may also have restrictions on the allowable combinations of operation, ordering, segment, and scope. See the *HSA Programmer's Reference Manual Version 1.2* for a full description of HSAIL memory operations.

3.3.2 Atomic operations

The following properties apply to atomic operations:

- Atomic operations must use naturally aligned addresses. If an address that is not naturally aligned is used for an atomic operation then the operation is undefined.
- Atomic loads and stores are single-copy atomic.
- Atomic read-modify-write operations are indivisible in the coherent order of the bytes accessed (see 3.7 Coherent order (on page 55)). The write operation follows the read operation immediately in the coherent order of the bytes accessed. The read and write components of the read-modify-write operation are single-copy-atomic.
- Synchronizing atomics (i.e., paired atomics with release and acquire semantics) must be to the same location and be of the same size.

3.3.3 Segments

The HSA memory consistency model defines the following different segments:

- Global segment, shared between all agents.
- Group segment, shared between work-items in the same work-group in an HSAIL kernel dispatch.
- Private, private to a single work-item in an HSAIL kernel dispatch.
- Kernarg, read-only memory visible to all work-items in an HSAIL kernel dispatch.
- Readonly, read-only memory visible to all agents.
- The flat segment is considered a virtual segment and all operations on the flat segment are considered to be performed on the actual segment a particular flat address maps to.

Read-only segments are not updated after the HSAIL kernel dispatch, therefore only ordinary load operations are relevant for these segments. Programs may assume that loads will read the initial value of a location prior to kernel dispatch.

The different segments are described in more detail in the *HSA Programmer's Reference Manual Version 1.2*.

A particular memory location is always accessed using the same segment.

3.3.3.1 Initialization of the kernarg segment

The HSA memory allocator supports a kernarg region property which is how the application locates a region that must be used for kernarg backing memory allocations.

The kernarg backing memory referenced by a kernel dispatch packet must be released to system scope so it is visible to the packet processor before dispatch of the packet (which can be achieved by the release when updating the AQL packet format field to a valid packet type).

It is the responsibility of the packet processor to make the kernarg segment (which is initialized to have the value of the kernarg backing memory) visible to the work-items of the kernel dispatch packet. Note in particular that no packet acquire fence is required for this.

If there is a race between the kernarg backing memory updates and the release to the packet processor of the dispatch packet, or a further update occurs before the dispatch packet completes then any load from the kernel arguments by the dispatched kernel is undefined.

3.3.3.2 Initialization of the readonly segment

It is the responsibility of HSA Runtime readonly segment update api to make updated memory visible to any future kernel dispatches executed on the agent which is updated. Note in particular that no packet acquire fence is required for this.

Any readonly segment update api call must be ordered outside the boundaries of kernel execution demarcated by:

- the start of kernel execution gated by release to the packet processor of a dispatch packet accessing the readonly segment, and
- the update of the completion signal after completion of the dispatch packet accessing the readonly segment.

If a readonly segment update api call occurs during a kernel execution accessing the readonly segment then all reads from the readonly segment by that kernel are undefined.

3.3.4 Allocating and freeing memory

The HSA Runtime or OS memory allocation APIs can be used to allocate memory. From the perspective of the memory model a memory allocation consists of a set of memory locations, and each call to allocate memory returns distinct memory locations.

Note that an implementation is permitted, but not required, to reuse memory that have been freed, provided it does not change the set of legal candidate executions for a program that does not have undefined behavior.

If there is a call A to an HSA Runtime or OS memory allocation API that returns a memory allocation M; and there is a memory operation B that reads, writes or read-modify-writes a memory location that is a member of M; then it is undefined behavior unless A happens-before B.

If there is a call A to an HSA Runtime or OS memory deallocation API that frees a memory allocation M; and there is a memory operation B that reads, writes or read-modify-writes a memory location that is a member of M; then it is undefined behavior unless B happens-before A.

If there is a call A to an HSA Runtime or OS memory deallocation API that frees memory allocation M; then it is undefined behavior if there is not a call B to an HSA Runtime or OS memory allocation API that returns memory allocation M, such that B happens-before A.

It is undefined behavior if the same memory allocation is deallocated more than once.

It is undefined behavior if a memory allocation created by the HSA Runtime memory allocator API is deallocated by other than the HSA Runtime memory deallocator API.

It is undefined behavior if a memory allocation created by the OS memory allocator API is deallocated by other than the OS memory deallocator API.

3.3.5 Ownership

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

An agent may be given ownership of an address range in the Global segment. The following rules apply to ownership:

Only a single agent may have ownership of an address range at a time. When an agent has ownership of an address range only that agent is allowed to access the address range.

If an agent accesses an address that is owned by another agent then the program is undefined.

There are two types of ownership:

- Read-only ownership, the agent may only read from the address range owned.
- Read-write ownership, the agent may write as well as read the address range owned.

Address range ownership is controlled through HSA runtime calls.

3.3.6 Scopes

The HSA memory consistency model defines the following different scopes:

- Work-item (`wi`)
- Wavefront (`wave`)
- Work-group (`wg`)
- Agent (`agent`)
- System (`system`)

Memory operation scope is used to limit the required visibility of a memory operation to a subset of observers (units of execution) in the system.

A special operation specifies one scope, however more than one scope may be implied through inclusivity (see 3.3.7 Scope instances (on the facing page)).

Ordinary memory operations have an implicit scope of work-item and are only required to be visible to other units of execution in relation to synchronizing operations, and are then required to be visible within the scope instance(s) of the synchronizing operation.

3.3.7 Scope instances

A named scope in a static program corresponds to a specific *scope instance* during execution. For example, a special operation that specifies work-group scope will correspond to a scope instance containing all the units of execution in the same work-group as the unit of execution that performs the special operation.

The actual scope of a special operation is limited by the maximum visibility of segments.

- A special operation performed using a flat address is limited to the maximum scope visibility of the underlying segment mapping for that address.
- A special operation on a byte location in the global segment with system scope performed on an address range owned by the agent behaves as if specified with agent scope.
- A special operation on a byte location in the group segment with system or agent scope behaves as if specified with work-group scope.
- Synchronizing operations with system or agent scope are performed with work-group scope on the group segment.
- Synchronizing operations with system, agent, work-group, or wave scope are performed with work-item scope on the private segment. As a result, atomic operations on private segment locations have no ordering side effects, and are treated similar to ordinary operations.
- Segments other than global and group are either not writable or only visible to a single unit of execution, and thus scopes are not applicable.

When a scope other than system is used then there may be several scope instances. For example, when using work-group scope, unit of executions in different work-groups would specify different instances of the work-group scope.

Scope instances are *inclusive*. When a scope instance is specified it implies all smaller scope instances. For example, if agent scope is specified for an operation then work-group, wave-front and work-item scope instance are also implied.

We define an operator *SI* that returns the set of scope instances specified by a memory operation.

The sets of scope instances S_1 and S_2 specified for two operations executed by two different units of execution match if they contain any common scope instance. This will be the case when:

- the largest scope of S_1 and S_2 are both system, or
- the largest scope of S_1 or S_2 is system, and for the other is agent, and the units of execution belong to the same agent, or
- the largest scope of S_1 or S_2 is system and for the other is work-group, and the units of execution belong to the same work-group, or
- the largest scope of S_1 or S_2 is system and for the other is wavefront, and the units of execution belong to the same wavefront, or
- the largest scope of S_1 and S_2 are both agent, and the units of execution belong to the same agent, or
- the largest scope of S_1 or S_2 is agent and for the other is work-group, and the units of execution belong to the same work-group, or
- the largest scope of S_1 or S_2 is agent and the for other is wavefront, and the units of execution belong to the same wavefront, or

- the largest scope of S_1 and S_2 are both work-group, and the units of execution belong to the same work-group, or
- the largest scope of S_1 or S_2 is work-group and for the other is wavefront, and the units of execution belong to the same wavefront, or
- the largest scope of S_1 and S_2 are both wavefront, and the units of execution belong to the same wavefront, or
- the operations are executed by the same unit of execution.

We define an operator *Match* as true when scope instances match:

$$Match(S_1, S_2) = (\varnothing \neq (S_1 \cap S_2))$$

Note that for operations executed by the same unit of execution (i.e., work-item) there is always a common scope instance and hence *Match* is always true.

For example, if work-items from different work-groups access the global segment using work-group scope they have no matching scope instances and *Match* is false.

3.3.8 Packet processor fences

In addition, the memory model defines packet memory fences that are only used by the packet processor (see 2.9.1 Packet header (on page 25)):

- A packet memory release fence makes any global segment or image data that was stored by any unit of execution that belonged to a dispatch that has completed the active phase on any queue of the same agent visible in all the scopes specified by the packet release fence.
- A packet memory acquire fence ensures any subsequent global segment or image loads by any unit of execution that belongs to a dispatch that has not yet entered the active phase on any queue of the same agent, sees any data previously released at the scopes specified by the packet acquire fence.

3.3.9 Forward progress of special operations

Special operations must make forward progress, and must be visible for the scope instances they specify in finite time. Ordinary stores are not required to be visible in finite time, but if they are ordered before something that is required to be visible in finite time, then it is a consequence of the ordering behavior that those ordinary stores will be visible in finite time.

3.4 Plausible executions

We define a plausible execution as the result of an HSA program in which each load operation observes the value of any store operation that occurs in the same execution. Plausible executions are a superset of the legal executions that could occur when an HSA-race-free program is run in an HSA-compliant system.

3.5 Candidate executions

Candidate executions are a subset of plausible executions in which a requisite set of observable apparent orders of operations (both ordinary and special), defined below, exist and are consistent.

If an HSA program is well-defined (see 3.12 Semantics of race-free programs (on page 58)), then an actual execution of that program will be one of the candidate executions.

3.5.1 Orders

Orders define relations over a set of operations.

$X >_A Y$ means operation X is ordered before Y in order A.

Some orders in a candidate execution must be consistent with others. An order A is consistent with order B if an only if there are no two operations X and Y such that $X >_A Y$ and $Y >_B X$.

3.6 Program order

Within any single unit of execution a, there is a total Program Order, ($\overrightarrow{po_a}$), of transactions consistent with the control flow of the program as specified by the execution model.

We may also refer to a single Program Order ($\overrightarrow{po}$), defined as the simple union of all per-unit of execution Program Orders ($\overrightarrow{po} = \bigcup_{\forall a \in \mathbb{A}} \overrightarrow{po_a}$) for all units of execution $\mathbb{A}$.

3.7 Coherent order

For any given byte location L in any segment, there is an apparent, total Coherent Order ($\overrightarrow{coh_L}$) of all loads and stores in an HSA-race-free program.

We may also refer to a single Coherent Order ($\overrightarrow{coh}$), defined as the simple union of all per-byte-location Coherent Orders ($\overrightarrow{coh} = \bigcup_{\forall L \in \mathbb{L}} \overrightarrow{coh_L}$) for all byte locations $\mathbb{L}$.

The coherent order must be consistent with program order ($\overrightarrow{po}$).

[comment] This order includes both ordinary and atomic operations and is the same in all scope instances.

[comment] Each byte of a load will observe the value produced by the most recent store to the same byte location in the coherent order.

3.8 Global dependence order

The execution model for single unit of execution *a* must define a notion of Local Dependence Order ($\overrightarrow{ldo_a}$) that defines dependent operations.

[comment] Informally, an operation Y is said to depend on an operation X from the same single unit of execution if X produces a result used by Y in an address or data computation, or if X affects control flow leading to Y.

We define a Global Dependence Order ($\overrightarrow{gdo}$) that captures the dependencies that exist among agents in the system. An operation X is ordered before an operation Y in the global dependence order if and only if the following conditions hold (formally, equivalent to the transitive irreflexive closure of all $\overrightarrow{ldo_a}$ and all $\overrightarrow{coh_L}$):

- $X >_{ldo_a} Y$ for some unit of execution a.
- X and Y are memory transactions that access byte location L and $X >_{coh_L} Y$.
- There is an operation A such that $X >_{gdo} A$ and $A >_{gdo} Y$.

By rule, there cannot be a cycle in $\overrightarrow{gdo}$.

[comment] The acyclic requirement prevents a situation in which an operation Y appears to complete before an operation X that affects the outcome of Y.

[comment] The acyclic requirement also prohibits out-of-thin-air values. Note that this does not prevent a cycle in the apparent order of all memory operations (that is, non-sequentially consistent executions are possible). See 3.13.3 Non-sequentially consistent execution (on page 71).

3.9 Scoped synchronization order

In the execution of an HSA-race-free program, acquire operations are ordered after any prior release operation that specifies (directly or indirectly through inclusion) the same scope. Formally, we call this the Scoped Synchronization Order ($\overrightarrow{sso_S}$) for scope instance S.

3.9.1 SC atomic release synchronization with an SC atomic acquire

An atomic operation X is ordered before an atomic operation Y in the scoped synchronization order of scope instance S if and only if all of the following conditions hold:

- X is an atomic store or read-modify-write with release or acquire-release semantics
- Y is an atomic load or read-modify-write with acquire or acquire-release semantics
- X and Y reference the same location L and are the same size (e.g., 64-bit)
- X and Y both specify (directly or indirectly through inclusion) scope instance S
- $X >_{coh_L} Y$ in the coherent order of operations for byte location L

3.9.2 Release fence synchronization with an SC atomic acquire

A fence X is ordered before an atomic operation Y in the scoped synchronization order of scope instance S if and only if all of the following conditions hold:

- X is a fence with release or acquire-release semantics
- Y is an atomic operation on location L with acquire or acquire-release semantics
- X and Y both specify S (directly or indirectly through inclusion) scope instance S
- There is a store or read-modify-write operation A to location L such that $X >_{po} A$
- A and Y are the same size (e.g., 64-bit)
- $A >_{coh_L} Y$ in the total coherent order $\overrightarrow{coh_L}$ for all byte locations common to A and Y.

3.9.3 Release SC atomic synchronization with an acquire fence

An atomic operation X is ordered before a fence Y in the scoped synchronization order of scope instance S if and only if all of the following conditions hold:

- X is an atomic store or read-modify-write on location L with release or acquire-release semantics
- Y is a fence with acquire or acquire-release semantics
- X and Y both specify (directly or indirectly through inclusion) scope instance S
- There is a load or read-modify-write operation A from location L such that $A >_{po} Y$

- X and A are the same size (e.g., 64-bit)
- $X >_{coh_L} A$ in the total coherent order $\overrightarrow{coh_L}$ for all byte locations common to X and A

3.9.4 Synchronization of two fences

A fence X is ordered before a fence Y in the scoped synchronization order of scope instance S if and only if all of the following conditions hold:

- X is a fence with release or acquire-release semantics
- Y is a fence with acquire or acquire-release semantics
- X and Y both specify S (directly or indirectly through inclusion) scope instance S
- There is a store or read-modify-write operation A to location L such that $X >_{po} A$
- There is a load or read-modify-write operation B from location L such that $B >_{po} Y$
- A and B are the same size (e.g., 64-bit)
- $A >_{coh_L} B$ in the total coherent order $\overrightarrow{coh_L}$ for all byte locations common to A and B

3.9.5 Notes

[comment] Note that there may be multiple scope instances for a particular scope. For example, work-items from two different work-groups that access the global segment using work-group scope will see two different instances of the work-group scope instance, and potentially two different Scoped Synchronization Orders.

[comment] In an execution, the completion order of fences, which have no associated value, can only be determined by observing the order of loads and stores surrounding the fences. Given a release fence R and an acquire fence A, R and A are only guaranteed to be synchronized once an ordering has been observed by a load and store such that $R >_{po} S >_{coh} L >_{po} A$. As will be apparent in 3.12 Semantics of race-free programs (on the next page), the store and load must be atomic to avoid introducing a race, and the store and load will likely be relaxed (otherwise the fences would be redundant).

[comment] If there also exists an ordinary store S' and ordinary load L' such that $S' >_{po} R$ and $A >_{po} L'$, then S' must be visible to L' (formally S' "HSA-happens-before" L', see 3.11 HSA-happens-before order (on the next page)). Practically, that means that most implementations will not reorder a relaxed store with an earlier release fence or a relaxed load with a later acquire fence, as such a reordering may cause L' to complete before S'.

3.10 Sequentially consistent synchronization order

In a candidate execution, there is a total apparent order of all synchronization operations with release, acquire, or acquire-release semantics in a single scope instance. This order is called Sequentially Consistent Synchronization Order ($\overrightarrow{sc_S}$).

Given synchronization operations X and Y, if $X >_{po} Y$ and X and Y specify the same scope instance S (directly or indirectly through inclusivity), then $X >_{sc_S} Y$.

In an HSA-race-free program, $\overrightarrow{sc_S}$ must be consistent with:

- Program order, $\overrightarrow{po}$
- Coherent order, $\overrightarrow{coh_L}$, for every byte location L
- All other sequentially consistent synchronizations orders $\overrightarrow{sc_S}$where $S' \neq S$

3.11 HSA-happens-before order

We define the order HSA-happens-before ($\overrightarrow{hhb}$) that captures the causal relationship among synchronized memory operations.

Given two memory operations X and Y, $X >_{hhb} Y$ if and only if:

- There exists operations A and B such that $A >_{sso_S} B$ for some scope instance S, and
- $X >_{po} A$ or X is A itself, and
- $B >_{po} Y$ or Y is B itself, or
- There exists another operation Z such that $X >_{hhb} Z$ and $Z >_{hhb} Y$.

Formally, HSA-happens-before is the irreflexive transitive closure of scope synchronization order and program order ($\overrightarrow{hhb} = (\overrightarrow{po} \cup \overrightarrow{sso})^+$).

Heterogeneous-happens-before is consistent with:

- Coherent Order, $\overrightarrow{coh_L}$, of every byte location L, and
- All Sequentially Consistent Synchronization Orders, $\overrightarrow{SC_S}$, for all scope instances S.

There can be no cycle in happens-before.

[comment] If a release operation X is followed by an acquire operation Y in a scope synchronization order of scope instance S, then the two operations X and Y are said to *synchronize with* respect to scope instance S.

[comment] A synchronization between X and Y orders X and all transactions preceding X in program order before Y and all transactions following Y in a single, globally visible apparent order called HSA-happens-Before.

[comment] Synchronization is transitive, such that two units of execution, A and C, can synchronize indirectly by each synchronizing with a unit of execution B. Transitivity also follows when different scopes are involved as long as each release/acquire pair, respectively, use matching scopes. For example, if A synchronizes with B through work-group scope and B synchronizes with C through device scope, A and C are considered synchronized.

3.12 Semantics of race-free programs

Given a program P, the candidate executions are all plausible executions in which the following orders and rules exist and are consistent as specified in the following sections.

3.12.1 Definitions for a valid candidate execution

- **Program order ($\overrightarrow{po_a}$)**: There is a total sequential order of operations by a single unit of execution *a* that respects the semantics of the execution model. See 3.6 Program order (on page 55).
- **Sequentially consistent synchronization order ($\overrightarrow{scs}$)**: There is a total apparent order of all synchronization operations within a scope instance *S* that is consistent with each program order in the execution. See 3.10 Sequentially consistent synchronization order (on page 57).
- **Coherent order ($\overrightarrow{coh_L}$)**: There is a total apparent order of all memory transactions to a single byte location L that is consistent with each sequentially consistent synchronization order and each program order in the candidate execution. See 3.9 Scoped synchronization order (on page 56).
- **Scoped synchronization order ($\overrightarrow{ssos}$)**: There is an apparent order of special operations for each scope instance S that defines how release operations synchronize with acquire operations. See 3.9 Scoped synchronization order (on page 56).
- **HSA-happens-before ($\overrightarrow{hhb}$):** There is a happens-before relation that captures the causal relationship of memory operations among synchronized units of execution.

  HSA-happens-before is the irreflexive transitive closure of the union of all program orders and all scoped synchronization orders:

  $$\left(\left(\bigcup_{\forall a \in A} \overrightarrow{po_a}\right) \cup \left(\bigcup_{\forall S \in \mathbb{S}} \overrightarrow{sso_S}\right)\right)^+$$

  where *A* is the set of all units of execution and $\mathbb{S}$ is the set of all scope instances in the candidate execution.

  $\overrightarrow{hhb}$ order must be consistent with each coherent order in the candidate execution.

  $\overrightarrow{hhb}$ order must be consistent with all Sequentially Consistent Synchronization Orders $\overrightarrow{scs}$ for all S in $\mathbb{S}$.

  There can be no cycle in $\overrightarrow{hhb}$.

- **Value of a load**: In a well-defined execution, a load (ordinary or atomic) from byte location L will always observe the most recent store in the coherent order of byte location L.
- **Forward progress**: All stores will occur in the coherent order in a reasonable, finite amount of time.

Given the set of candidate executions, we can determine if the actual execution of program P is well-defined or undefined by checking for races.

3.12.2 Conflict definitions

- **Ordinary conflict**: Two operations X and Y conflict iff they access one or more common byte locations, at least one is a write, and at least one is an ordinary data operation.
- **Special conflict**: Two special operations X and Y conflict iff X and Y access the same byte location and:

- X and Y are different sizes (e.g., 32-bit vs. 64-bit), or
- At least one is a write (or a read-modify-write), and ⌐ *Match*(*Sl*(X), *Sl*(Y)).

3.12.3 Undefined operations

- **Undefined operation**: Any operation X is undefined if:
  - There is an undefined operation Y such that $Y >_{ido} X$. Note that any operation depending on an undefined value is itself undefined.
- **Undefined store**: A store S (atomic or ordinary) is undefined if:
  - The general rule on undefined operations defined above applies, or
  - S and another store S' are either an ordinary or special conflict, and S and S' are unordered in $\overrightarrow{hhb}$
- **Undefined load**: A load L (ordinary or atomic) is undefined if:
  - The general rule on undefined operations defined above applies, or
  - The store S, which is most recent in coherent order with respect to L, is undefined (a load is undefined if it reads a value written by an undefined store), or
  - L and a store S are either an ordinary or special conflict, and L and S are unordered in $\overrightarrow{hhb}$.

    Note that a load is undefined if the load is ordinary and subject to a race with an ordinary or atomic store, if the load is atomic and subject to a race with an ordinary store, or there is a scope mismatch.

3.12.4 Races

- **HSA-race**: An HSA-race occurs iff either an ordinary or atomic load is undefined.
- **HSA-race-free execution**: An execution is HSA-race-free iff there are no HSA-races.
- **HSA-race-free program**: A program is HSA-race-free iff all valid candidate executions of the program are HSA-race-free.

3.12.5 Program semantics

- **Well-defined and undefined executions**: An execution is either well-defined or undefined. An execution is undefined if there is any operation X such that:
  - X is a load or store operation and the address of X is the result of an undefined operation, or
  - X is an image load or store operation and any of the coordinates of the image operation X is the result of an undefined operation, or
  - X is a conditional operation and the condition of X is the result of an undefined operation.
- **Well-defined and undefined HSA programs**: An HSA program is well-defined iff all valid candidate executions of the program are well-defined executions; otherwise, it is undefined.
- **Semantics of a well-defined HSA program**: The actual execution of a well-defined HSA program will be one of the valid candidate executions.
- **Sequential consistency**: The execution of an HSA-race-free program *that does not use relaxed*

atomics will appear sequentially consistent. The value of a load is the value produced by the most recent store to the same location that immediately precedes the load in $\overrightarrow{hhb}$.

3.12.6 Corollaries

- In a well-defined program, the subset of the execution containing only defined operations and *that does not include relaxed atomics* will appear sequentially consistent.
- In an HSA-race-free program in which each location is accessed exclusively by either only ordinary or only atomic operations:
  - The value observed by an ordinary load will be the unique most recent ordinary store to the same location in HSA-happens-before.
  - The value observed by an atomic load will come from a store, X, that is the most recent store that precedes the load in the $\overrightarrow{coh_L}$ order. The store X is either ordered with respect to the load in HSA-happens-before or not. If the store X is ordered with the load in HSA-happens-before, then the store X must precede the load in HSA-happens-before and there will be no other store, X', that appears between X and the load in HSA-happens-before (i.e., X will be one of potentially several most recent stores in the HSA-happens-before partial order).
- In an HSA-race-free program in which at least one location is accessed by both ordinary and atomic operations:
  - The value observed by an ordinary load will come from one of the most recent stores to the same location in HSA-happens-before. There may be more than one such store.
  - The value observed by an atomic load is the same as above.

3.13 Examples

For clarity, all examples show pseudo code followed by valid HSAIL with the following exceptions:

- In HSAIL, group variables cannot be initialized. However, for clarity and brevity, this is ignored in the examples, and group variable declarations in the examples typically specify an initial value.
- Forward progress of work-items is complex, and some aspects of the examples, such as spin-locks, require particular conditions to be met to ensure live-lock cannot occur. To more clearly illustrate the memory model forward progress considerations for work-items are ignored in the examples.
- Flat addresses cannot be expressed as constant expressions in HSAIL; however, this is ignored for clarity and brevity.

3.13.1 Sequentially consistent execution

3.13.1.1 Synchronizing operations are sequentially consistent by definition

Consider four units of execution, A, B, C, and D, in the same work-group, and two memory locations, X and Y:

```
group int X = 0;
group int Y = 0;
```

A:

```
   atomic_store(X, 1, screl, wg);
```

B:

```
atomic_store(Y, 1, screl, wg);
```

C:

```
s1 = atomic_load(X, scacq, wg);
s2 = atomic_load(Y, scacq, wg);
```

D:

```
s3 = atomic_load(Y, scacq, wg);
s4 = atomic_load(X, scacq, wg);
```

```
group_s32 &X=0;
group_s32 &Y=0;
```

A:

```
atomicnoret_st_screl_wg_s32 [&X], 1;
```

B:

```
atomicnoret_st_screl_wg_s32 [&Y], 1;
```

C:

```
atomic_ld_scacq_wg_s32 $s1, [&X];
atomic_ld_scacq_wg_s32 $s2, [&Y];
```

D:

```
atomic_ld_scacq_wg_s32 $s3, [&Y];
atomic_ld_scacq_wg_s32 $s4, [&X];
```

Synchronizing operations are sequentially consistent by definition, and thus outcomes that violate sequential consistency are not allowed. In this example the following outcome is not allowed:

$s1=1, $s2=0 & $s3=1, $s4=0

Sequential consistency implies that units of execution C and D must observe X and Y changing from 0 to 1 in the same order.

3.13.1.2 Successful synchronization between units of execution

Consider two units of execution, A and B, in the same work-group, and two memory locations, X and Y:

```
group int X = 0;
group int Y = 0;
```

A:

```
X = 53;
atomic_store(Y, 1, screl, wg);
```

B:

```
while (atomic_load(Y, scacq, wg) != 1) {}
s2 = X;
```

```
group_s32 &X=0;
group_s32 &Y=0;
```

A:

```
   st_s32 53, [&X];
   atomicnoret_st_screl_wg_s32 [&Y], 1;
```

B:

```
@b_start:
   atomic_ld_scacq_wg_s32 $s1, [&Y];
   cmp_ne_s32_b1, $s1, $c1, 1;
   cbr_b1 $c1, @b_start;
   ld_s32 $s2, [&X];
```

If B loads $s1 with the value '1', then B must also load the value 53 in $s2. This is an example of a successful handover of data from unit of execution A to B.

3.13.1.3 Correct synchronization, safe transitivity with a single scope

Consider three units of execution, A, B, and C, in the same work-group, and three memory locations X, Y, and Z:

```
global int X = 0;
global int Y = 0;
global int Z = 0;
```

A:

```
   X = 53;
   atomic_store(Y, 1, screl, system);
```

B:

```
   while (atomic_load(Y, scacq, system) != 1) {}
   atomic_store(Z, 1, screl, system);
```

C:

```
   while (atomic_load(Z, scacq, system) != 1) {}
   s3 = X;  // s3 must get '53'
```

```
   global_s32 &X=0;
   global_s32 &Y=0;
   global_s32 &Z=0;
```

A:

```
   st_s32 53, [&X];
   atomicnoret_st_screl_system_s32 [&Y],1;
```

B:

```
@b_start:
   atomic_ld_scacq_system_s32 $s1, [&Y];
   cmp_ne_s32_b1, $s1, $c1, 1;
   cbr $c1, @b_start;
   atomicnoret_st_screl_system_s32 [&Z],1;
```

C:

```
@c_start:
   atomic_ld_scacq_system_s32 $s2, [&Z];
   cmp_ne_s32_b1, $s2, $c2, 1;
   cbr $c2, @c_start;
   ld_s32 $s3, [&X]; // $s3 must get '53'
```

At the conclusion of this example, if $s1 = $s2 = 1, then $s3 must observe 53. This is an example of a successful handover of data from unit of execution A to C with transitivity implied through multiple synchronizations.

3.13.1.4 Race-free transitive synchronization through multiple scopes

Consider three units of execution, A, B, and C, three memory locations, X, Y, and Z. A and B are in the same work-group. C is in a different work-group:

```
global int X = 0;
global int Y = 0;
global int Z = 0;
```

A:

```
   X = 53;
   atomic_store(Y, 1, screl, wg);
```

B:

```
   while (atomic_load(Y, scacq, wg) != 1) {}
   s2 = X; // s2 must get '53'
   atomic_store(Z, 1, screl, system);
```

C:

```
   while (atomic_load(Z, scacq, system) != 1) {}
   s3 = X; // s3 must get '53'
```

```
   global_s32 &X=0;
   global_s32 &Y=0;
   global_s32 &Z=0;
```

A:

```
   st_s32 53, [&X];
   atomicnoret_st_screl_wg_s32 [&Y],1;
```

B:

```
   atomic_ld_scacq_wg_s32 $s1, [&Y];
   ld_s32 $s2, [&X];
   atomicnoret_st_screl_system_s32 [&Z], 1;
```

C:

```
@c_start:
   atomic_ld_scacq_system_s32 $s3, [&Z];
   cmp_ne_s32_b1 $s3, $c1, 1;
   cbr_b1 $c1, @c_start
   ld_s32 $s4, [&X];
```

At the conclusion of this example, if B loads the value 1 into $s1, then $s2 must be 53. If C loads the value 2 into $s3, then $s4 must also be 53. This is an example of race-free transitive synchronization involving multiple scopes.

3.13.1.5 Successful synchronization through scope inclusion

Consider two units of execution, A and B, in the same work-group, and two memory locations, X and Y:

```
global int X = 0;
global int Y = 0;
```

A:

```
   X = 53;
   atomic_store(Y, 1, screl, wg);
```

B:

```
   while (atomic_load(Y, scacq, agent) != 1) {}
   s2 = X; // s2 must get '53'
```

```
   global int X=0;
   global int Y=0;
```

A:

```
   st_s32 53, [&X]
   atomicnoret_st_screl_wg_s32 [&Y], 1
```

B:

```
@b_start:
   atomic_ld_scacq_agent_s32 $s1, [&Y];
   cmp_ne_s32_b1 $s1, $c1, 1;
   cbr_b1 $c1, @b_start;
   ld_s32 $s2, [&X]
```

At the conclusion of this example, $s2 must get the value 53. This is an example of race-free synchronization through different but matching (through inclusion) scopes.

3.13.1.6 Successful synchronization through scope inclusion and scope transitivity

Consider three units of execution, A, B, and C, and three memory locations, X, Y, and Z. A and B are in the same work-group. C is in a different work-group.

```
global int X = 0;
global int Y = 0;
global int Z = 0;
```

A:

```
   X = 53;
   atomic_store(Y, 1, screl, wg);
```

B:

```
   while (atomic_load(Y, scacq, agent) != 1) {}
   s2 = X; // s2 must get 53
   atomic_store(Z, 1, screl, system);
```

C:

```
   while (atomic_load(Z, scacq, system) != 1) {}
   s4 = X; // s4 must get 53
```

```
   global_s32 &X=0;
   global_s32 &Y=0;
   global_s32 &Z=0;
```

A:

```
   st_s32 53, [&X];
   atomicnoret_st_screl_wg_s32 [&Y],1;
```

B:

```
@b_start:
   atomic_ld_scacq_agent_s32 $s1, [&Y];
   cmp_ne_s32_b1 $s1, $c1, 1;
   cbr_b1, $c1, @b_start;
   ld_s32 $s2, [&X];
   atomicnoret_st_screl_system_s32 [&Z],1;
```

C:

```
@c_start:
   atomic_ld_scacq_system_s32 $s3, [&Z];
   cmp_ne_s32_b1 $s3, $c2, 1;
   cbr_b1, $c2, @c_start;
   ld_s32 $s4, [&X];
```

At the conclusion of this example, $s2 and $s4 must both get the value 53.

3.13.1.7 Coh and hhb must be consistent

Consider two units of execution, A and B, and two memory locations X and Y:

```
global int X = 0;
global int Y = 0;
```

A:

```
   X = 52;
   X = 53;
   atomic_store(Y, 1, screl, system);
```

B:

```
   while (atomic_load(Y, scacq, system) != 1) {}
   s2 = X; // s2 must get 53
```

```
   global_s32 &X=0;
   global_s32 &Y=0;
```

A:

```
   st_s32 52, [&X];
   st_s32 53, [&X];
   atomicnoret_st_screl_system_s32 [&Y], 1;
```

B:

```
@b_start:
   atomic_ld_scacq_system_s32 $s1, [&Y];
   cmp_ne_s32_b1 $s1, $c1, 1;
   cbr_b1, $c1, @b_start;
   ld_s32 $s2, [&X];
```

At the conclusion of this example, $s2 must get the value 53. This outcome is enforced by the requirement that the coherent orders $\overrightarrow{coh}$ must be consistent with the happens-before order $\overrightarrow{hhb}$, which prevents the coherent order for location X from being store '52' -> load -> store '53'.

3.13.1.8 Separate segment synchronization

Consider two units of execution, A and B, in the same work-group, and two memory locations, X and Y:

```
global int X = 0;
group int Y = 0;
```

A:

```
   X = 53;
   atomic_store(Y, 1, screl, wg);
```

B:

```
   while (atomic_load(Y, scacq, wg) != 1) {}
   s2 = X; // s2 must get 53
```

```
   global_s32 &X=0;
   group_s32 &Y;
```

A:

```
   st_s32 1, [&X];
   atomicnoret_st_screl_wg [&Y],1;
```

B:

```
@b_start:
   atomic_ld_scacq_wg_s32 $s1, [&Y];
   cmp_ne_s32_b1 $s1, $c1, 1;
   cbr_b1 $c1, @b_start;
   ld_s32 $s2, [&X]; // s2 must get 53
```

This example shows that synchronization can cross segments. Even though the atomic store and load specify a location in global memory, they still synchronize the group location Y.

3.13.2 Sequentially consistent with relaxed operations

The following examples show the limits of relaxed (ordinary and atomic) operation reorderings. All examples in this subsection are HSA-race-free and will appear sequentially consistent.

3.13.2.1 Successful synchronization between units of execution using relaxed atomics

Consider two units of execution, A and B, in the same work-group, and two memory locations, X and Y:

```
group int X = 0;
group int Y = 0;
```

A:

```
   X = 53;
   atomic_thread_fence(screl, wg);
   atomic_store(Y, 1, rlx, wg);
```

B:

```
   while (atomic_load(Y, rlx, wg) != 1);
   atomic_thread_fence(scacq, wg);
   s2 = X;
```

```
   group_s32 &X=0;
   group_s32 &Y=0;
```

A:

```
   st_global_s32 53, [&X];
   memfence_screl_wg;
   atomicnoret_st_global_rlx_wg_s32 [&Y], 1;
```

B:

```
@b_start:
   atomic_ld_global_rlx_wg_s32 $s1, [&Y];
   cmp_ne_b1_s32 $c1, $s1, 1;
   cbr_b1 $c1, @b_start;
   memfence_scacq_wg;
   ld_global_s32 $s2, [&X];
```

If B loads $s1 with the value '1', then B must also load the value 53 in $s2. This is an example of a successful handover of data from unit of execution A to B.

3.13.2.2 Correct synchronization between a store-release and a fence-acquire

```
global int X = 0;
global int R = 0;
```

A:

```
   X = 53;
   atomic_store(R, 1, screl, system);
```

B:

```
   while (atomic_load(R, relaxed, system) != 1) {}
   atomic_thread_fence(scacq, system);
   s2 = X   // s2 must observe '53';

-----------------------------------------------------------------

   global_s32 &X=0;
   global_s32 &R=0;
```

A:

```
   st_global_s32 53, [&X];
   atomicnoret_st_global_screl_system_s32 [&R], 1;
```

B:

```
@b_start:
   atomic_ld_global_rlx_system_s32 $s1, [&R];
   cmp_ne_b1_s32 $c1, $s1, 1;
   cbr_b1 $c1, @b_start;
   memfence_scacq_system;
   ld_global_s32 $s2, [&X];
```

3.13.2.3 Correct synchronization between a fence-release and a load-acquire

```
global int X = 0;
global int R = 0;
```

A:

```
   X = 53;
   atomic_thread_fence(screl, system);
   atomic_store(R, 1, relaxed, system);
```

B:

```
   while (atomic_load(R, scacq, system) != 1) {}
   s2 = 53;

-----------------------------------------------------------------

   global_s32 &X=0;
   global_s32 &R=0;
```

A:

```
   st_global_s32 53, [&X];
   memfence_screl_system;
   atomicnoret_st_global_rlx_system_s32 [&R], 1;
```

B:

```
@b_start:
   atomic_ld_global_scacq_system_s32 $s1, [&R];
   cmp_ne_b1_s32 $c1, $s1, 1;
   cbr_b1 $c1, @b_start;
   ld_global_s32 $s2, [&X];
```

3.13.2.4 Incorrect synchronization involving an SC atomic and a fence

```
global int X = 0;
global int P = 0;
global int R = 0;
```

A:

```
   X = 53;
   atomic_store(P, 1, screl, system);
   atomic_store(R, 1, relaxed, system);
```

B:

```
   while (atomic_load(R, relaxed, system) != 1) {}
   atomic_thread_fence(scacq, system);
   R1 = X;

-----------------------------------------------------------------------

   global_s32 &X=0;
   global_s32 &Y=0;
```

A:

```
   st_global_s32 53, [&X];
   memfence_screl_wg;
   atomicnoret_st_global_screl_system_s32 [&P], 1;
   atomicnoret_st_global_rlx_system_s32 [&R], 1;
```

B:

```
@b_start:
   atomic_ld_global_rlx_system_s32 $s1, [&R];
   cmp_ne_b1_s32 $c1, $s1, 1;
   cbr_b1 $c1, @b_start;
   memfence_scacq_system;
   ld_global_s32 $s2, [&X];
```

In the above example, there is no synchronization between A and B because the atomic store release (to P) is not to the same location (R) that could make the fence observed. As a result, this program has a race on location X.

3.13.2.5 Store speculation is not observable

Consider two units of execution, A and B, and two memory locations, X and Y.

```
global int X = 0;
global int Y = 0;
```

A:

```
   if (X == 1)
         Y = 1; // will never execute
```

B:

```
   if (Y == 1)
         X = 1; // will never execute
```

```
   global_s32 &X=0;
   global_s32 &Y=0;
```

A:

```
   ld_global_s32 $s1, [&X];
   cmp_ne_b1_s32 $c1, $s1, 1;
   cbr_b1 $c1, @a_done;
   st_global_s32 1, [&Y]; // will never execute
@a_done:
```

B:

```
   ld_global_s32 $s2, [&Y];
   cmp_ne_b1_s32 $c2, $s2, 1;
   cbr_b1 $c2, @b_done;
   st_global_s32 1, [&X]; // will never execute
@b_done:
```

In all valid executions of this example, the final value of both $s1 and $s2 will be '0'.

This example shows that HSA implementations are prohibited from making speculative stores observable.

3.13.2.6 No out-of-thin-air values

Consider two units of execution, A and B, and two memory locations, X and Y.

```
global int X = 0;
global int Y = 0;
```

A:

```
   s1 = atomic_load(X, rlx, system); // s1 gets 0
   atomic_store(Y, s1, rlx, system);
```

B:

```
   s2 = atomic_load(Y, rlx, system); // s2 gets 0
   atomic_store(Y, s2, rlx, system);
```

```
   global_s32 &X=0;
   global_s32 &Y=0;
```

A:

```
   atomic_ld_global_rlx_system_s32 $s1, [&X];  // $s1 gets '0'
   atomicnoret_st_global_rlx_system_s32 [&Y], $s1;
```

B:

```
   atomic_ld_global_rlx_system_s32 $s2, [&Y];  // $s2 gets '0'
   atomicnoret_st_global_rlx_system_s32 [&X], $s2;
```

The only valid outcome of this execution is $s1=$s2=0.

In an HSA-race-free execution, all byte values observed by a load must be produced by a non-speculative store in the execution. Loads will not observe so-called out-of-thin-air values. In the formal model, this is enforced by the requirement that the Global Dependence Order ($\overrightarrow{gdo}$) is acyclic.

3.13.3 Non-sequentially consistent execution

The following examples use relaxed atomics, are HSA-race-free, and can result in non-sequentially consistent executions.

3.13.3.1 Dekker's Algorithm

Consider two units of execution, A and B, in the same work-group, and two memory locations A and B:

```
global int X = 0;
global int Y = 0;
```

A:

```
atomic_store(X, 1, rlx, system);
s1 = atomic_load(Y, rlx, system);
```

B:

```
atomic_store(Y, 1, rlx, system);
s2 = atomic_load(X, rlx, system);
```

```
global_s32 &X=0;
global_s32 &Y=0;
```

A:

```
atomicnoret_st_global_rlx_system_s32 [&X], 1;
atomic_ld_global_rlx_system_s32 $s1, [&Y];
```

B:

```
atomicnoret_st_global_rlx_system_s32 [&Y], 1;
atomic_ld_global_rlx_system_s32 $s2, [&X];
```

In the above example, a valid outcome (though not the only valid outcome) at the end of execution is $s1 = $s2 = 0. Note that in that execution, there is no total, globally visible order (i.e., sequentially consistent order) of memory operations.

3.13.4 Races

The following examples contain an HSA-race.

3.13.4.1 Conflict without synchronization

Consider two units of execution, A and B, and a memory location X:

A:

```
X = 1;
```

B:

```
s1 = X;
```

A:

```
st_global_s32 1, [&X];
```

B:

```
ld_global_s32 $s1, [&X];
```

The write to X by A and the read from X by B form an ordinary conflict. There is no synchronization, which means that the operations from A and B are unordered in HSA-happens-before ($\overrightarrow{hhb}$). Thus, this example contains a race.

3.13.4.2 Insufficient scope

Consider two units of execution, A and B, in different work-groups, and two memory locations, X and Y:

```
global int X = 0;
global int Y = 0;
```

A:

```
   X = 1;
   atomic_store(Y, 1, screl, wg);
```

B:

```
   while (atomic_load(Y, scacq, wg) != 1) {}
   s2 = X; // races with X = 1
```

```
   global_s32 &X=0;
   global_s32 &Y=0;
```

A:

```
   st_global_s32, 1, [&X];
   atomicnoret_st_global_screl_wg_s32 [&Y], 1;
```

B:

```
@b_start:
   atomic_ld_global_scacq_wg_s32 $s1, [&Y];
   cmp_ne_b1_s32 $c1, $s1, 1;
   cbr_b1 $c1, @b_done;
   ld_global_s32 $2, [&Y];
```

There are two races in this example. First, the atomic load by B races with the atomic store by A because the work-group scopes are not inclusive. Second, because the scopes used for synchronization similarly do not match, the write and read of ordinary variable X form a race.

APPENDIX A.
Limits

This appendix lists the maximum or minimum values that HSA implementations must support:

- **Wavefront size**: Every call convention of every kernel agent of an implementation must have a wavefront size that is a power of 2 in the range from 1 to 256 inclusive.
- **Work-group dimensions and size**: Each work-group dimension is limited to 2^{16} - 1. The work-group size is the product of the three individual work-group dimensions and is limited to 2^{32} - 1. Every kernel agent of an implementation must support individual work-group dimensions and work-group sizes of at least 256. Every permutation of supported individual work-group dimensions that do not exceed the supported work-group size must be supported.
- **Grid dimensions and size**: Each grid dimension is limited to 2^{32} - 1. The grid size is the product of the three individual grid dimensions and is limited to 2^{64} - 1. Every kernel agent of an implementation must support individual grid dimensions and grid sizes of at least the corresponding supported work-group sizes. Every permutation of supported individual grid dimensions that do not exceed the supported grid size must be supported.
- **Size of group segment memory**: Every kernel agent of an implementation must support at least 32K bytes of group segment memory per compute unit for group segment variables. This amount might be reduced if an implementation uses group memory for the implementation of other HSAIL features such as fine-grained barriers and the exception detection.
- **Size of private segment memory**: Every kernel agent of an implementation must support at least the smaller of the following number of bytes of private segment memory per work-item:
  - 256 bytes
  - 64K divided by the work-group size specified by the associated kernel dispatch packet

  Since every kernel agent of an implementation is required to support work-groups up to 256 work-items, and is required to execute at least one work-group at a time, this implies a kernel agent must be capable of supporting at least 64K bytes of private segment memory.
- **Size of kernarg segment memory**: Every implementation must support at least 1K bytes of kernarg segment memory per dispatch.
- **Image data type support**: Every agent that supports images must support images with the following per agent limits:
  - **1D images**: Must support 1D, 1DA image sizes of at least 16384 image elements for width.
  - **2D images**: Must support 2D, 2DA, 2DDEPTH, 2DADEPTH image sizes of at least (16384 x 16384) image elements for (width, height) respectively.
  - **3D images**: Must support 3D image sizes of at least (2048 x 2048 x 2048) image elements for (width, height, depth) respectively.
  - **Image arrays**: Must support 1DA, 2DA, 2DADEPTH image arrays of at least 2048 image layers for array size.

- **1DB images**: Must support 1DB image sizes of at least 65536 image elements for width.
- **Read-only image handles**: Must support having at least 128 read-only image handles created at any one time.
- **Write-only and read-write image handles**: Must support having a combined total of at least 64 write-only and read-write image handles created at any one time.
- **Sampler handles**: Must support having at least 16 sampler handles created at any one time.

HSA runtime queries are available to determine the actual supported limits for a specific agent. Note that the creation of an image or sampler may fail due to insufficient resources. For example, an agent may not be able to create an image with an image size it supports due to insufficient memory.

APPENDIX B.
Glossary

acquire synchronizing operation
: A memory operation that specifies an acquire memory ordering.

agent
: A hardware or software component that participates in the HSA memory model. An agent can submit AQL packets for execution. An agent may also, but is not required, to be a kernel agent. It is possible for a system to include agents that are neither kernel agents nor host CPUs.

application global memory
: Memory that is to be shared between all agents and the host CPUs for processing using the HSA. This corresponds to the global segment.

Architected Queuing Language (AQL)
: An AQL packet is an HSA-standard packet format. AQL kernel dispatch packets are used to dispatch kernels on the kernel agent and specify the launch dimensions, kernel code handle, kernel arguments, completion detection, and more. Other AQL packets control aspects of a kernel agent such as when to execute AQL packets and making the results of memory operations visible. AQL packets are queued on user mode queues. See 2.9 Requirement: Architected Queuing Language (AQL) (on page 24).

arg segment
: A memory segment used to pass arguments into and out of functions.

compute unit
: A piece of virtual hardware capable of executing the HSAIL instruction set. The work-items of a work-group are executed on the same compute unit. A kernel agent is composed of one or more compute units.

global segment
: A memory segment in which memory is visible to all units of execution in all agents.

grid
: A multidimensional, rectangular structure containing work-groups. A grid is formed when a program launches a kernel.

group segment
: A memory segment in which memory is visible to a single work-group.

host CPU
: An agent that also supports the native CPU instruction set and runs the host operating system and the HSA runtime. As an agent, the host CPU can dispatch commands to a kernel agent using memory operations to construct and enqueue AQL packets. In some systems, a host CPU can also act as a kernel agent (with appropriate HSAIL finalizer and AQL mechanisms).

HSA application
: A program written in the host CPU instruction set. In addition to the host CPU code, it may include zero or more HSAIL programs.

HSA implementation
: A combination of one or more host CPU agents able to execute the HSA runtime, one or more kernel agents able to execute HSAIL programs, and zero or more other agents that participate in the HSA memory model.

HSA memory management unit (HSA MMU)
: A memory management unit used by kernel agents and other agents, designed to support page-granular virtual memory access with the same attributes provided by a host CPU MMU.

HSA Memory Node (HMN)
: A node that delineates a set of system components (host CPUs and kernel agents) with “local” access to a set of memory resources attached to the node's memory controller and appropriate HSA-compliant access attributes.

HSAIL
: Heterogeneous System Architecture Intermediate Language. A virtual machine and a language. The instruction set of the HSA virtual machine that preserves virtual machine abstractions and allows for inexpensive translation to machine code.

invalid address
: An invalid address is a location in application global memory where an access from a kernel agent or other agent is violating system software policy established by the setup of the system page tables attributes. If a kernel agent accesses an invalid address, system software shall be notified. See 2.1 Requirement: Shared virtual memory (on page 11) and 2.9.3 Error handling (on page 27).

kernarg segment
: A memory segment used to pass arguments into a kernel.

kernel
: A section of code executed in a data-parallel way by a kernel agent. Kernels are written in HSAIL and are translated by a finalizer to machine code.

kernel agent
: An agent that supports the HSAIL instruction set and supports execution of AQL kernel dispatch packets. As an agent, a kernel agent can dispatch commands to any kernel agent (including itself) using memory operations to construct and enqueue AQL packets. A kernel agent is composed of one or more compute units.

memory type
: A set of attributes defined by the translation tables, covering at least the following properties:

  - Cacheability
  - Data Coherency requirements
  - Requirements for endpoint ordering, being the order of arrival of memory accesses at the endpoint
  - Requirements for observation ordering, being restrictions on the observable ordering of

memory accesses to different locations

- Requirements for multi-copy atomicity
- Permissions for speculative access

natural alignment
: Alignment in which a memory operation of size *n* bytes has an address that is an integer multiple of *n*. For example, naturally aligned 8-byte stores can only be to addresses 0, 8, 16, 24, 32, 40, and so forth.

packet ID
: Each AQL packet has a 64-bit packet ID unique to the user mode queue on which it is enqueued. The packet ID is assigned as a monotonically increasing sequential number of the logical packet slot allocated in the user mode queue. The combination of the packet ID and the queue ID is unique for a process.

packet processor
: Packet processors are tightly bound to one or more agents, and provide the functionality to process AQL packets enqueued on user mode queues of those agents. The packet processor function may be performed by the same or by a different agent to the one with which the user mode queue is associated that will execute the kernel dispatch packet or agent dispatch packet function.

primary memory type
: The memory type used by default for user processes. Agents shall have a common interpretation of the data coherency (excluding accesses to read-only image data) and *cacheability* attributes for this type.

private segment
: A memory segment in which memory is visible only to a single work-item. Used for read-write memory.

process address space ID (PASID)
: An ID used to identify the application address space within a host CPU or guest virtual machine. It is used on a device to isolate concurrent contexts residing in shared local memory

queue ID
: An identifier for a user mode queue in a process. Each queue ID is unique in the process. The combination of the queue ID and the packet ID is unique for a process.

readonly segment
: A memory segment for read-only memory.

release synchronizing operation
: A memory operation that specifies a release memory ordering.

segment
: A contiguous addressable block of memory. Segments have size, addressability, access speed, access rights, and level of sharing between work-items. Also called memory segment.

signal handle
: An opaque handle to a signal which can be used for notification between threads and work-items belonging to a single process potentially executing on different agents in the HSA system.

spill segment
: A memory segment used to load or store register spills.

unit of execution
: A unit of execution is a program-ordered sequence of operations through a processing element. A unit of execution can be any thread of execution on an agent, a work-item, or any method of sending operations through a processing element in an HSA-compatible device.

user mode queue
: A user mode queue is a memory data structure created by the HSA runtime on which AQL packets can be enqueued. The packets are processed by the packet processor associated with the user mode queue. For example, a user mode queue associated with the packet processor of a kernel agent can be used to execute kernels on that kernel agent. See 2.8 Requirement: User mode queuing (on page 17).

wavefront
: A group of work-items executing on a single program counter.

work-group
: A work-group is a partitioning of the grid of work-items formed by a kernel dispatch. It is an instance of execution in a compute unit.

work-item
: A single unit of execution of the grid formed by a kernel dispatch.

Index

T

U

V

W