This presentation covers Gen-Z Read and Write operations. Read and write operations are the basis for exchanging data between any component type.
Disclaimer

This document is provided ‘as is’ with no warranties whatsoever, including any warranty of merchantability, noninfringement, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. Gen-Z Consortium disclaims all liability for infringement of proprietary rights, relating to use of information in this document. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted herein.

Gen-Z is a trademark or registered trademark of the Gen-Z Consortium.

All other product names are trademarks, registered trademarks, or servicemarks of their respective owners.

All material is subject to change at any time at the discretion of the Gen-Z Consortium

http://genzconsortium.org/
Gen-Z supports multiple OpClasses that support read operations.
- P2P 64 is optimized for point-to-point and meshed topologies. It supports cache-line sized coherent reads (exclusive, shared, etc.) and variable sized non-coherent reads with byte granularity addressing.
- Core 64 supports variable-sized reads with byte granularity addressing. Core 64 also supports multi-op reads. Multi-op reads enable components to perform 2-4 read operations within a single request packet. In addition to improving protocol efficiency, multi-op reads provide scatter and gather data movement.
- The LDM OpClass supports large reads up to 4 GiB. Due to their variable execution time, large reads should be off-loaded to data mover logic to enable processor cores to perform other tasks or reduce power consumption.

Gen-Z supports multiple OpClass that support multiple write operations.
- P2P 64 supports cache-line sized coherent writes and variable sized non-coherent writes with byte granularity addressing.
- Core 64 supports variable-sized writes with byte granularity addressing. Core 64 also support multi-op writes. Multi-op writes enable components to perform 2-4 write operations or write combo operations (write + interrupt, write + signal, write + read, write + wake, etc.). Multi-op writes are self-contained, hence, all writes or write combinations can be atomically executed to ensure ordering is maintained.
• There are multiple write operation types. Delineation simplifies implementation, improves performance, etc.
To perform a read operation, a Requester generates a Read request packet, upon receipt and execution, the Responder generates a Read Response packet.

- Requesters that support P2P 64 Read requests use the Tag to correlate Read Responses with Read requests.
- Requesters that support non-P2P-* OpClass read requests use the [SCID / SSID, DCID / DSSID, Tag] tuple to correlate Read Responses with Read requests.
- The Requester is responsible for ensuring end-to-end reliability. The Responder only validates and executes the request packet, and does not maintain any other record of the operation.
A LDM Read operation consists of a single LDM Read request packet and multiple LDM Read Response packets.

- Each LDM Read Response contains an additional Offset field that indicates where the response data is to be placed relative to the Requester’s implementation-specific buffer.
Multi-op Read requests support 2-4 read operations within a single request packet.

- Reads can be any supported size.
- Multi-op Read is a form of “gather-scatter”
  - Requester can gather (contiguous) or scatter (discontiguous) across buffers.
  - At least one Read Response per embedded Read Request.
  - Unique request Tags enable Requester to delineate Read Responses.

Multi-op Read requests support 2-4 read operations within a single request packet.
- Reads can be any supported size. If the Read length is less than or equal to 256 bytes, then a single Read Response is returned. If the Read length is larger than 256 bytes, then multiple LDM Read Responses are returned.
- Multi-Op Reads are a form of gather-scatter. A Requester can issue multiple read operations to different Responder addresses, and then gather the data into a single Requester buffer. Similarly, a Requester can issue multiple read operations to a single Responder buffer (just different addresses within the buffer), and then scatter the responses across multiple Requester buffers.
- Multi-Op Reads can be used to perform native Gen-Z reads or to perform LPD (Logical PCI Device) read operations.
P2P 64 Read Request and Response Packet Formats

- Byte 0: PCRC (4:3), OpCode (4:3), Address (0:3), OpCode (0:3), LEN (4:3), Tag (8:5), LEN (2:0), Tag (4:0)
- Byte 4: Address (63:32)
- Byte 8: Address (31:0)
- Byte 12: RD See

- Byte 0: PCRC (4:3), OpCode (4:3), Address (0:3), OpCode (0:3), LEN (4:3), Tag (8:5), LEN (2:0), Tag (4:0)
- Byte 4: Meta (if present)
- Byte 8: Payload

- Byte YY: CRC (4:3), Pad CNT, M3, RREP Reason
Core 64 Read Request and Response Packet Formats

```
<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 4</th>
<th>Byte 8</th>
<th>Byte 12</th>
<th>Byte 16</th>
<th>Byte 20</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCRC</td>
<td>Tag</td>
<td>OCL</td>
<td>OSCLCRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KHID28</td>
<td>KHID16</td>
<td>KHID14</td>
<td>KHID12</td>
<td>KHID10</td>
<td>KHID8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address [63:32]</td>
<td>Access Key</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address [31:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECRC</td>
<td>FPS</td>
<td>N0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 4</th>
<th>Byte 8</th>
<th>Byte 12</th>
<th>Byte 16</th>
<th>Byte 20</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCRC</td>
<td>Tag</td>
<td>OCL</td>
<td>OSCLCRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KHID28</td>
<td>KHID16</td>
<td>KHID14</td>
<td>KHID12</td>
<td>KHID10</td>
<td>KHID8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address [63:32]</td>
<td>Access Key</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address [31:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECRC</td>
<td>FPS</td>
<td>N0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 4</th>
<th>Byte 8</th>
<th>Byte 12</th>
<th>Byte 16</th>
<th>Byte 20</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCRC</td>
<td>Tag</td>
<td>OCL</td>
<td>OSCLCRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KHID28</td>
<td>KHID16</td>
<td>KHID14</td>
<td>KHID12</td>
<td>KHID10</td>
<td>KHID8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address [63:32]</td>
<td>Access Key</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address [31:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECRC</td>
<td>FPS</td>
<td>N0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 4</th>
<th>Byte 8</th>
<th>Byte 12</th>
<th>Byte 16</th>
<th>Byte 20</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCRC</td>
<td>Tag</td>
<td>OCL</td>
<td>OSCLCRC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KHID28</td>
<td>KHID16</td>
<td>KHID14</td>
<td>KHID12</td>
<td>KHID10</td>
<td>KHID8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address [63:32]</td>
<td>Access Key</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address [31:0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECRC</td>
<td>FPS</td>
<td>N0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
# LDM 1 Read Request and Response Packet Formats

| Byte 0 | Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 | Byte 6 | Byte 7 | Byte 8 | Byte 9 | Byte 10 | Byte 11 | Byte 12 | Byte 13 | Byte 14 | Byte 15 | Byte 16 | Byte 17 | Byte 18 | Byte 19 | Byte 20 | Byte 21 | Byte 22 | Byte 23 | Byte 24 | Byte 25 | Byte 26 | Byte 27 | Byte 28 | Byte 29 | Byte 30 | Byte 31 |
|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
Write request packets contain essentially the same information as Read request packets, with the addition of a data payload field that contains 0-256 bytes of data.

If the Requester requires reliable delivery, then a Standalone Acknowledgment packet is returned per Write request packet.
Meta Read Response and Meta Write

- Meta Read and Meta Write request packets enable a Requester to transmit a quantity of Meta data in conjunction with the respective read or write operation.
- Meta data communicates additional information to a Responder to facilitate solution operation.
  - For example, a Requester moving a 4 KiB block of data wants to send an additional 64-bit block-level data integrity value could transmit 15 256-byte Write request packets and 1 256-byte Meta Write request packet to complete the data transfer and communicate the data integrity value. Upon receipt of the Meta Write request packet, the Responder uses the 64 bits of Meta data to perform block-level data integrity. Similarly, a solution could use a mix of Read Response / LDM 1 Read Response packets and Meta Read Response packets / LDM 1 Read Response packets with the Meta field present to efficiently communicate additional Responder information.
  - Use of Meta data is not constrained to block-level data integrity; there are many possible use cases, e.g., the Meta field can communicate control information to a memory module that contains a DRAM cache and Persistent Memory (PM) or Flash media.
  - The Meta field in a Meta Write request packet could guide the media controller on what data to cache in the DRAM or guide cache prefetch algorithms to reduce subsequent read or write latency and associated jitter.
  - Similarly, the Meta field in a Meta Read Response packet could provide media status such as read and write NVM endurance levels, remaining time at current workload level until garbage collection is required, etc.
If a Responder supports persistent media, then a Requester can use a Write P (Persistent) request to ensure that data is persistent before an acknowledgment is returned. Each Write P request packet is independently acknowledged. Acknowledgment at the hardware level is more efficient than using higher-level logic or additional software to ascertain persistence.

- Used to ensure that a write requests successfully updates non-volatile media
- Each Write P request is independently acknowledged
  - Acknowledgment packet is delayed until the Responder can guarantee that the media will be updated
If the Persistent Memory media is relatively slow, then a Requester can use a Persistent Flush operation to ascertain when the data for all outstanding write operations have reached a persistent state. The outstanding write operations can be of any type, and could have been issued by multiple Requesters. The Persistent Flush indicates that at the time it was acknowledged, that all outstanding write operations to persistent media have been successfully executed and the data is persistent.
If a problem has been encountered with the underlying memory media, the Requester can issue a Write Poison request to ensure that all subsequent accesses do not act upon the media’s content. The actual mechanism to poison the underlying memory media data is outside of the specification’s scope (as a reminder, Gen-Z abstracts the memory media), and will depend upon the media type, the media controller’s access resolution, etc.

Note: A Write Poison could also be used as an indirect mechanism to impact subsequent Requesters that access the poisoned address range.
Write Multi-Op requests operate similar to multi-op read requests, except that all data to be written is contained within the request packet. Further, a single acknowledgment packet is transmitted to indicate success or failure.
## Write Multi-Op Variants

- Gen-Z supports multiple write multi-op variants
  - To simplify programming model
  - To reduce communication overheads
  - To improve application performance
- Write with Interrupt—write a single payload and then trigger a component-local interrupt
  - For example, write to a completion queue and then signal an interrupt
- Write-Wake—write a single payload and then wake up a thread or process
- Write-Read—write a single payload and then read a single buffer
  - For example, partial cache line write followed by full cache line read
- Signaled Write—write a single payload and then atomically increment the data at a separate address
  - Signaled writes can provide 2-5x performance improvements in distributed applications (e.g., PGAS) by eliminating the need to rendezvous distributed threads
  - Instead, threads periodically poll the increment location (not payload), then take application-specific actions

Multi-Op write requests support multiple variants. This simplifies communication and ensures operations are executed using strong ordering. The following multi-op Write variants are supported:

- **Write with interrupt** performs a write operation followed by an interrupt operation. The interrupt is either a native Gen-Z interrupt or a LPD interrupt.
- **Write-Wake** is a write followed by execute of a wake operation using the supplied process or thread identifier.
- **Write-Read** is a write followed by a read operation. This enables the address range to be updated (e.g., a partial cache line), followed by a read of the entire cache line or page.
- **Signaled Write** is a write followed by the atomic increment of the data at the indicated address. Signaled writes eliminate complex coordination techniques, and have been shown to improve application performance by 2x-5x.
Write MSG operations enable a Requester to send multi-packet messages to a Responder without targeting a specific addressed buffer, i.e., the packets target anonymous buffers. When using datagram communications, this provides multiple advantages:

- Multiple Requesters can independently communicate with a Responder without requiring explicit buffer management, i.e., the Requesters share the Responder’s pool of receive buffers
- Responders transparently manage buffers to meet workload needs
- Requesters and Responders use Context Identifiers to simplify packet processing and enable greater parallelism (e.g., multiple pools of receive buffers).
- Write MSG operations support unicast and multicast communications
- Write MSG operations can support unreliable and reliable message delivery

Write MSG operations supports up to 512 KiB message sizes (covers 99% of the application needs)

Write MSG operations support an optional 96-bit receive tag used by the Responder to filter messages based on application needs rather than on a FIFO basis. This enables more responsive solutions.

Write MSG supports an embedded read option. This enables a Requester to use a single
Write MSG packet to communicate to a Responder where the source buffer is located. The Responder can allocate a local buffer (can be larger than what is supported by Write MSG itself), and then pull the data using Reads or LDM Reads via a data mover into the local buffer. Responders can directly place the data into an application buffer without using the processor to copy the data, the data buffer can be dynamically allocated eliminating the need to pre-allocate and unnecessarily consume memory, and the Responder can pull the data at its own rate reducing the potential for congestion.

Since Write MSG operations target anonymous buffers, they are ideal for bootstrapping services, supporting traditional management applications that are datagram based, supporting traditional network stacks including support for large send offload and receive-side scaling without the constraints or complexity found in traditional network implementations, etc.
Each Write MSG packet is self-describing. Hence, the packets can take advantage of multipath topologies to increase aggregate bandwidth, enable natural resiliency, etc.

A Write MSG packet targets a RSPCTXID. A Responder can support up to $2^{24}$ context identifiers to enable scalability and parallelism.

Each Write MSG packet contains an offset field that indicates where the payload is to be placed within the message. If using a reliable Write MSG, the corresponding Standalone Acknowledgment packet indicates which payload was successfully placed. This enables the Requester to determine success and whether to retransmit any packets.
This illustrates the Write MSG packet format. If RT == 1b, then the RCV-Tag field is present in the first Write MSG packet within a given messaging. If ER == 1b, then this indicates if the payload contains an embedded read. Embedded Reads can reduce the probability of Responder oversubscription / congestion.
Write MSG with Embedded Read Operation

Sequence 1:
1. Identify Buffer to be written by Requester
2. Transact Write/Flush MSG with Embedded Read
3. Standalone Acknowledgment (Tag = 226)
4. Write/Flush MSG, Tag = 226
5. Standalone Acknowledgment (Tag = 226)
6. Generate Read Response
7. Read/Load Read (Address, Buffer)
8. Generate Read Response
9. Atomic Add/Or Complete Address, Tag = 1023
10. Read/Load Address (Address, Buffer)
11. Atomic Response (Tag = 1022)
12. Transact NVRAM (Tag = 1022)
13. Transact NVRAM completes Atomic

Sequence 2:
1. Transact NVRAM completes Atomic
Receive Tags are used to filter messages based on application-specific needs instead of requiring messages to be handled in FIFO order.

There are multiple implementation options that can be implemented in hardware or a combination of hardware and software depending upon whether OS or middleware software needs to take action to complete the operation, e.g., allocate a large buffer for an embedded read.
Thank you

This concludes this presentation. Thank you.