SiFive TileLink Specification

Version 1.8.1

Table of Contents

Version	Date	Note
1.7-draft	July 30, 2018	Pre-Release version
1.7.1	December 3, 2018	Release version.
1.8-draft	May 3, 2019	Pre-Release Version. Corrupt/Denied.
1.8.0	August 9, 2019	TL-C Opcodes and Permissions.
1.8.1	January 27, 2020	Textual corrections and improved Deadlock Freedom chapter.

1. Introduction

TileLink is a chip-scale interconnect standard providing multiple masters with coherent memory-mapped access to memory and other slave devices. TileLink is designed for use in a System-on-Chip (SoC) to connect general-purpose multiprocessors, co-processors, accelerators, DMA engines, and simple or complex devices, using a fast scalable interconnect providing both low-latency and high-throughput transfers. TileLink:

- is a free and open standard for tightly coupled, low-latency SoC buses
- was designed for RISC-V but supports other ISAs
- provides a physically addressed, shared-memory system
- can be implemented over scalable, hierarchically composable, point-to-point networks
- provides coherent access for an arbitrary mix of caching or non-caching masters
- can scale down to simple slave devices or scale up to high-throughput slaves

Some of the important features of TileLink include:

- cache-coherent shared memory, supporting a MESI-equivalent protocol
- verifiable deadlock freedom for any conforming SoC
- out-of-order completion to improve throughput for concurrent operations
- decoupled interfaces, easing register-stage insertion
- stateless bus-width adaptation and burst fragmentation

1.1. Protocol Conformance Levels

A TileLink network may support a mix of communicating agents, each supporting different subsets of the protocol. The TileLink specification includes three conformance levels for attached agents, which indicates which subset of the protocol they must support as shown in Table 1. The simplest is TileLink Uncached Lightweight (TL-UL), which supports only simple memory read and write (Get/Put) operations of single words. The next most complex is TileLink Uncached Heavyweight (TL-UH), which adds various hints, atomic operations, and burst accesses but without support for coherent caches. Finally, TileLink Cached (TL-C) is the complete protocol, which supports use of coherent caches.

Table 1. TileLink conformance levels

	TL-UL	TL-UH	TL-C
Read/Write operations	y	y	y
Multibeat messages	.	y	y
Atomic operations	.	y	y
Hint operations	.	y	y
Cache block transfers	.	.	y
Channels B+C+E	.	.	y

When a TL-C processor agent communicates with a TL-UL device agent, either the processor agent should refrain from using the more advanced features or there must be a TL-C-to-TL-UL adapter in the network between the two. Agents could support other combinations of features but only the three listed conformance levels are covered by this specification.

1.2. Document Overview

The remainder of this specification is broken up into the following sections:

- Section 2 gives an overview of the TileLink architecture and its common abstractions.
- Section 3 defines the specific signals required by each TileLink channel.
- Section 4 defines how those signals are use to exchange TileLink messages.
- Section 5 describes Tilelink's deadlock-free design, and provides rules and explanations to ensure conformance.
- Section 6 gives an overview of the operations available to TileLink agents, and provides guidance on their ordering, use of address spaces, and transaction identifiers.
- Section 7 details the messages used to perform basic get/put operations on TileLink.
- Section 8 extends TileLink with burst transfers, atomic operations, and hints.
- Section 9 outlines how cached data blocks are managed in the complete TileLink protocol.

2. Architecture

The TileLink protocol is defined in terms of a graph of connected *agents* that send and receive *messages* over point-to-point *channels* within a *link* to perform *operations* on a shared address space.

operation

A change to an address range's data values, permissions or location in the memory hierarchy.

agent

An active participant in the protocol that sends and receives messages in order to complete operations (a more precise definition will be given in Section 5.1).

channel

A one-way communication connection between a master interface and a slave interface carrying messages of homogeneous priority.

message

A set of control and data values sent over a particular channel.

link

The set of channels required to complete operations between two agents.

2.1. Network Topology

Pairs of agents are connected by links. One end of each link connects to a master interface in one agent, and the other end connects to a slave interface in the other agent. The agent with the master interface can request the agent with the slave interface to perform memory operations, or request permission to transfer and cache copies of data. The agent with the slave interface manages permissions and access to a range of addresses, wherein it performs memory operations on behalf of requests arriving from the master interface.

Figure 1 shows a TileLink network consisting of a single link between a master interface and a slave interface, with two channels. To perform an operation on shared memory, the master sends a request message on the request channel to the slave and awaits an acknowledgement message on the response channel.

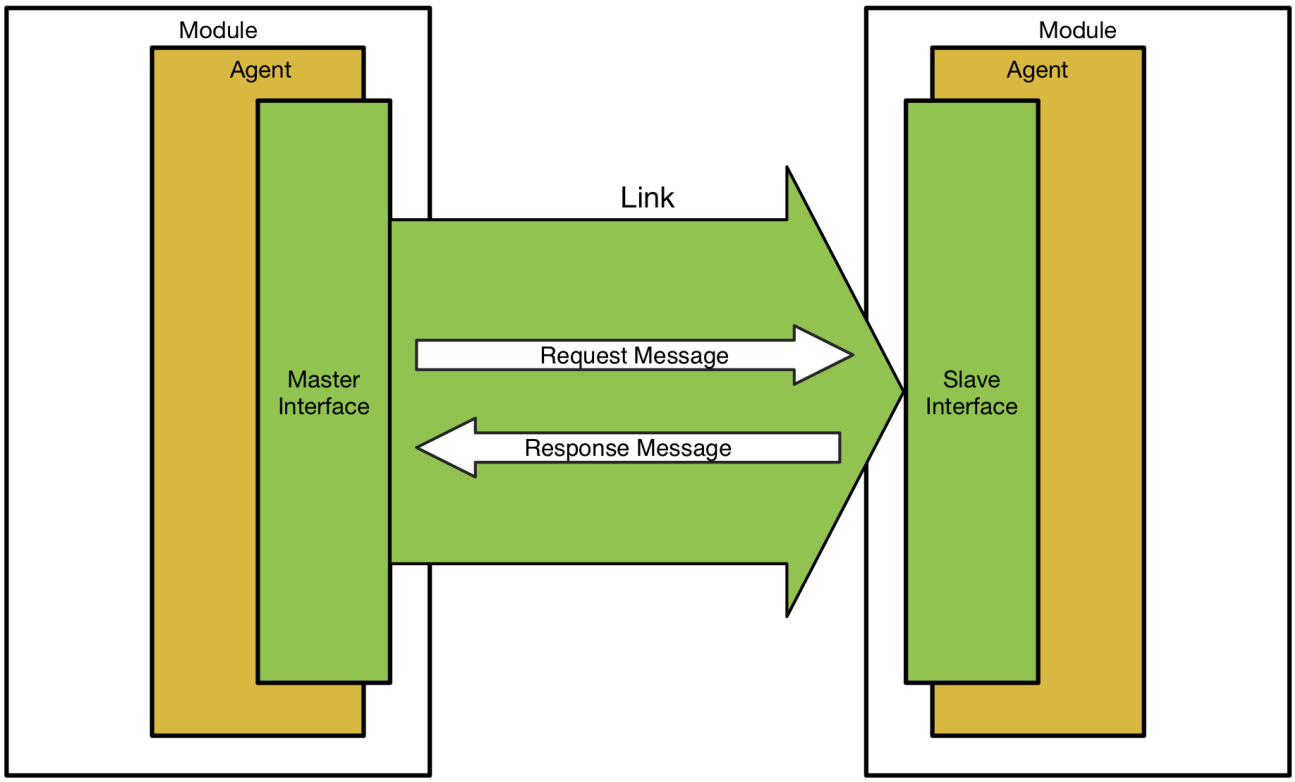

Figure 1. Overview of the most basic TileLink network operation.

Two modules are connected by a link, with one module containing an agent with a master interface and the other module containing an agent with a slave interface. The agent with a master interface sends a request to an agent with a slave interface. The agent with the slave interface communicates with backing memory if required. Having obtained the required data or permissions, the slave responds to the original requestor.

TileLink supports a wide variety of network topologies, subject to the restrictions specified in Section 5.3. Figure 2 illustrates an example of such a topology, wherein two of the modules (the crossbar and the cache) have agents that have a master interface on their right-side and a slave interface on their left-side.

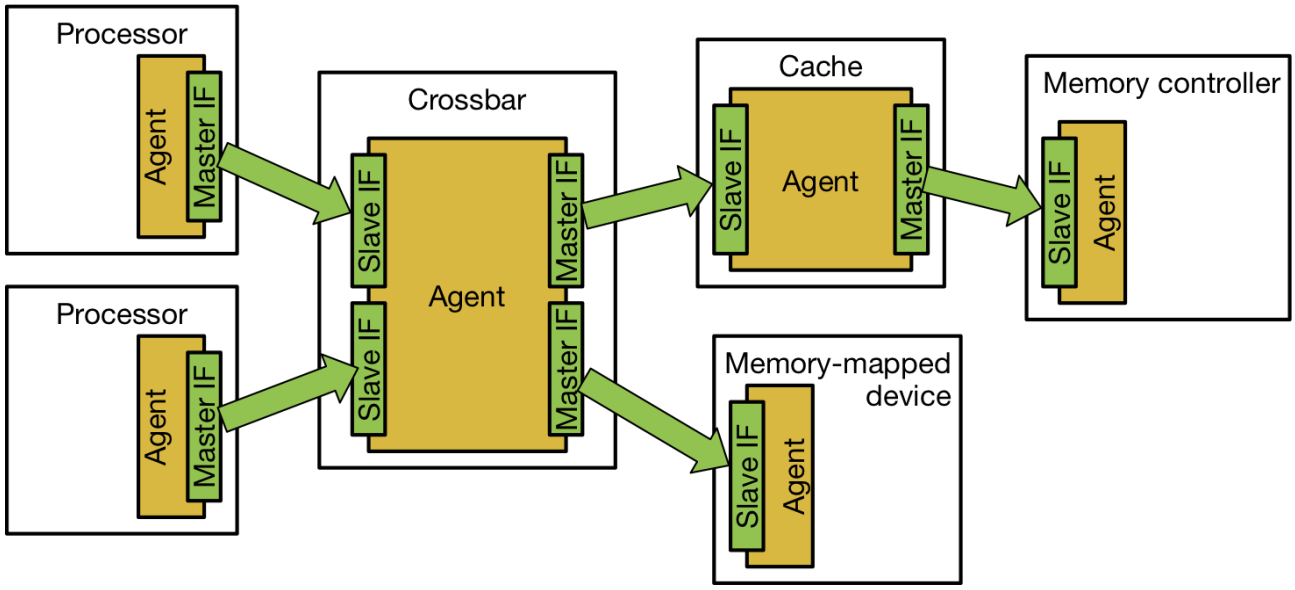

Figure 2. Example of a more complicated TileLink network topology.

It is important to note that a single hardware module can contain multiple independent TileLink agents. An example is shown in Figure 3, where the crossbar has one agent that routes data between links and a second agent that allows configuration state to be accessed.

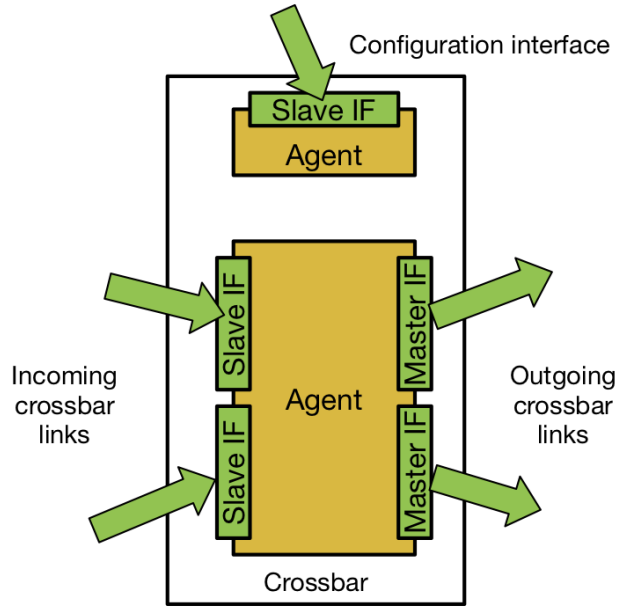

Figure 3. Example module with two TileLink agents.

This is an example of a more complicated crossbar module that contains two agents. One agent has multiple interfaces and is used to route data in normal operation, while the other agent has a single slave interface to access configuration data for the crossbar.

2.2. Channel Priorities

Within each network link, the TileLink protocol defines five logically independent channels over which messages can be sent by agents. To avoid deadlock, TileLink specifies a priority amongst the channels' messages that must be strictly enforced. Most channels contain both transaction control signals as well as a bus to exchange data. Channels are directional, in that each passes messages either from master to slave interface or from slave to master interface. Figure 4 illustrates the directionality of the five channels.

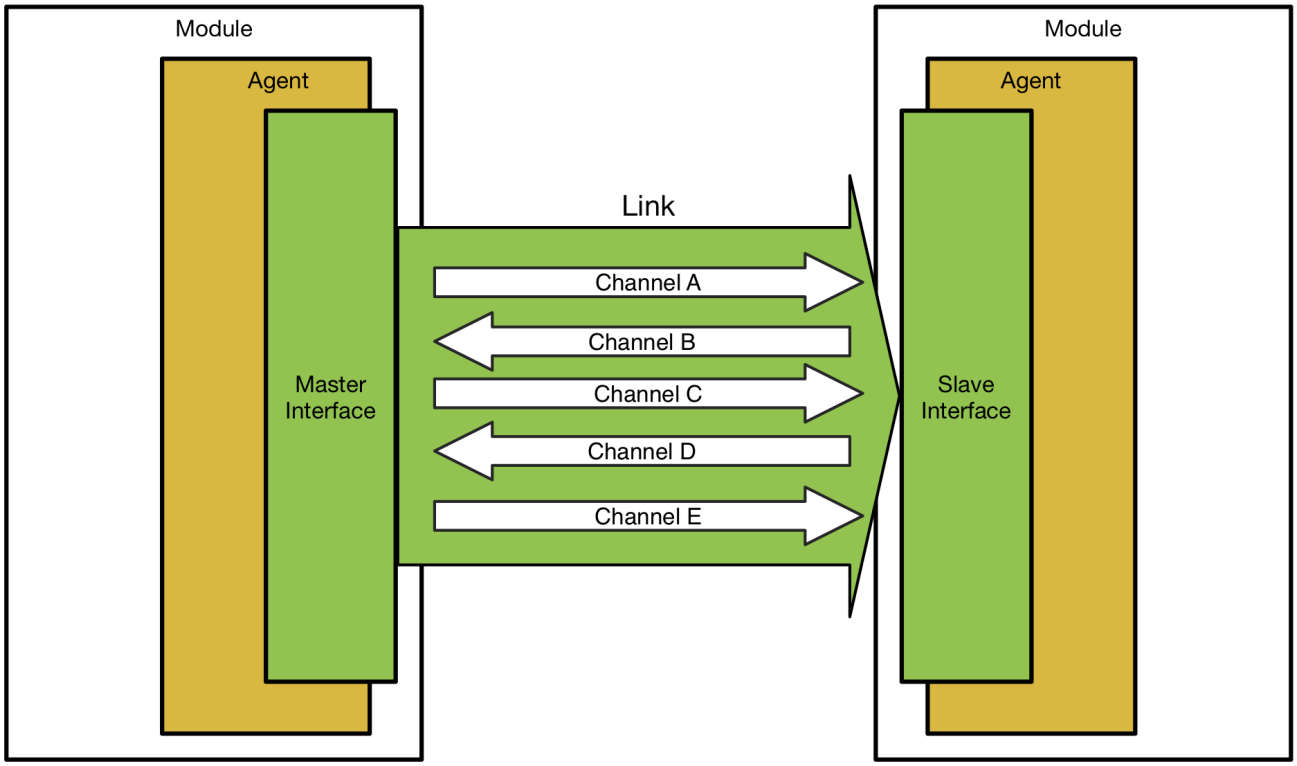

Figure 4. The five channels that comprise a TileLink link between any pair of agents.

The two basic channels required to perform memory access operations are:

Channel A

Transmits a request that an operation be performed on a specified address range, accessing or caching the data.

Channel D

Transmits a data response or acknowledgement message to the original requestor.

The highest protocol conformance level (TL-C) adds three additional channels that provide the capability to manage permissions on cached blocks of data:

Channel B

Transmits a request that an operation be performed at an address cached by a master agent, accessing or writing back that cached data.

Channel C

Transmits a data or acknowledgment message in response to a request.

Channel E

Transmits a final acknowledgment of a cache block transfer from the original requestor, used for serialization.

The prioritization of messages across channels is A < B < C < D < E, in order of increasing priority. Priorities ensure that messages flowing through the TileLink network never enter a routing or hold-and-wait loop. In other words, the message flow through all channels between all agents remains acyclic. This is a necessary property for TileLink to remain deadlock free; see Section 5 for details.

2.3. Address Space Properties

Properties limit what messages are allowed to be injected into a TileLink network, based on the range of addresses that the operation is targeting. Properties that might be ascribed to an address space include its: TileLink conformance level, memory consistency model, cacheability, FIFO ordering requirements, executeability, privilege level, and any Quality-of-Service guarantees.

Relying on properties, TileLink separates the concerns of determining what operations are possible on a particular address from the contents of the messages themselves. By front-loading the effort of determining whether a operation is legal onto the agent sending the initiatory request message, TileLink is able to eschew a variety of signals from its channel contents.

Specific mechanisms for describing which address ranges have which properties and how those properties in turn govern message injection are beyond the scope of this document.

3. Signal Descriptions

This chapter tabulates all signals used by TileLink's five channels, which are summarized in Table 2. When combined with each channel's direction, the signal type in Table 3 determines signal direction. The widths of these signals are parameterized by values described in Table 4.

Table 2. Overview of TileLink Channels

Channel	Direction	Purpose
A	Master to Slave	Request messages sent to an address.
B	Slave to Master	Request messages sent to a cached block (TL-C only).
C	Master to Slave	Response messages from a cached block (TL-C only).
D	Slave to Master	Response messages from an address.
E	Master to Slave	Final handshake for cache block transfer (TL-C only).

Table 3. TileLink signal types. Channel direction is as indicated in Table 2

Type	Direction	Description
X	Input	Clock or reset signal, an input to both TileLink agents.
C	Channel direction	Control signals, unchanging between beats of a burst.
D	Channel direction	Data signals, changing on every beat.
V	Channel direction	Valid signal, indicates **C**/**D** contain valid data.
R	Reverse direction	Ready signal, indicating that **V** was accepted.

Table 4. TileLink per-link parameters.

Parameter	Description
w	Width of the data bus in bytes. Must be a power of two.
a	Width of each address field in bits.
z	Width of each size field in bits.
o	Number of bits needed to disambiguate per-link master sources.
i	Number of bits needed to disambiguate per-link slave sinks.

3.1. Signal Naming Conventions

Other than the `clock` and `reset` signals, TileLink signal names consist of the channel identifier (a–e) followed by an underscore, followed by the name of the signal (enumerated in the following subsections).

For devices with multiple TileLink interfaces, it is recommended to prefix all TileLink signal names with some descriptive token and an underscore. For example, `a_opcode` becomes `gpio_a_opcode`.

3.2. Clocking, Reset, and Power

TileLink is a synchronous bus protocol. Both master interface and slave interface on a TileLink link must share the same clock, reset, and power. However, different links within the topology may have

different clocks, resets, and power.

Table 5. TileLink Clock and Reset Signals common to all channels

Signal	Type	Width	Description
`clock`	X	1	Bus clock. Inputs are sampled on the rising edge.
`reset`	X	1	Bus reset. Active HIGH. May be asserted asynchronously, but must be deasserted synchronous with a rising edge of clock.

3.2.1. Clock

Every channel samples its signals on the rising edge of the clock. Output signals may only change after the rising edge of the clock.

3.2.2. Reset

Before deasserting reset, `a_valid`, `c_valid`, and `e_valid` must be driven LOW by the master, while `b_valid` and `d_valid` must be driven LOW by the slave. The `valid` signals may be driven HIGH after the first rising edge of clock where reset is LOW. The `valid` signals must be driven LOW for at least 100 cycles while reset is asserted.

Ready, control, and data signals are free to take any value during reset.

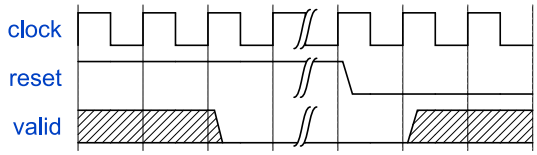

Figure 5. Valid must be driven LOW for at least 100 cycles during reset.

3.2.3. Power or Clock Crossing

It is forbidden for one side of a TileLink link to power down while its opposite is powered on.

If TileLink must cross between power or clock domains, a TileLink-to-TileLink adapter is needed which acts as a slave in one domain and a master in the other domain. The two interfaces of this adapter can then be safely powered, clocked, and reset separately from the other.

It is highly recommended that a crossing carefully coordinates with the rest of the SoC to ensure that there are no unanswered TileLink requests when one half of the crossing is reset or depowered. If a TileLink message is ever lost or repeated, it could cause the entire TileLink bus to deadlock.

3.3. Channel A (Mandatory)

Channel A flows from master interface to slave interface, carrying request messages sent to a particular address. This channel is used by all TileLink conformance levels and is mandatory.

Table 6. Channel A signals

Signal	Type	Width	Description
a_code	C	3	Operation code. Identifies the type of message carried by the channel. (Table 12)
a_param	C	3	Parameter code. Meaning depends on a_opcode; specifies a transfer of caching permissions or a sub-opcode. (Section 7.2, Section 8.2, Section 9.3)
a_size	C	***z***	Logarithm of the operation size: 2^n bytes (Section 4.5)
a_source	C	***o***	Per-link master source identifier. (Section 6.4)
a_address	C	***a***	Target byte address of the operation. Must be aligned to a_size. (Section 4.5)
a_mask	D	***w***	Byte lane select for messages with data. (Section 4.5)
a_data	D	***8w***	Data payload for messages with data. (Section 4.5)
a_corrupt	D	1	The data in this beat is corrupt. (Section 4.4)
a_valid	V	1	The sender is offering progress on an operation. (Section 4.1)
a_ready	R	1	The receiver accepted the offered progress. (Section 4.1)

3.4. Channel B (TL-C only)

Channel B flows from slave interface to master interface, carrying request messages sent to a particular cached data block held by a particular master. This channel is used by the TL-C conformance level and is optional in lower levels.

Table 7. Channel B signals

Signal	Type	Width	Description
b_opcode	C	3	Operation code. Identifies the type of message carried by the channel. (Table 12)
b_param	C	3	Parameter code. Meaning depends on ; specifies a transfer of caching permissions or a sub-opcode. (Section 9.3)
b_size	C	***z***	Logarithm of the operation size: 2^n bytes. (Section 4.5)
b_source	C	***o***	Per-link master source identifier. (Section 6.4)
b_address	C	***a***	Target byte address of the operation. Must be aligned to b_size. (Section 4.5)
b_mask	D	***w***	Byte lane select for messages with data. (Section 4.5)
b_data	D	***8w***	Data payload for messages with data. (Section 4.5)
b_corrupt	D	1	Corruption was detected in data payload. (Section 4.4)
b_valid	V	1	The sender is offering progress on an operation. (Section 4.1)
b_ready	R	1	The receiver accepted the offered progress. (Section 4.1)

3.5. Channel C (TL-C only)

Channel C flows from master interface to slave interface. It can carry response messages to Channel B requests sent to a particular cached data block. It is also used to voluntarily write back dirtied cached data. This channel is used at the TL-C conformance level and is optional in lower levels.

Table 8. Channel C signals

Signal	Type	Width	Description
c_opcode	C	3	Operation code. Identifies the type of message carried by the channel. (Table 12)
c_param	C	3	Parameter code. Meaning depends on ; specifies a transfer of caching permissions. (Section 9.3)
c_size	C	***z***	Logarithm of the operation size: 2^n bytes. (Section 4.5)
c_source	C	***o***	Per-link master source identifier. (Section 6.4)
c_address	C	***a***	Target byte address of the operation. Must be aligned to c_size. (Section 4.5)
c_data	D	***8w***	Data payload for messages with data. (Section 4.5)
c_corrupt	D	1	Corruption was detected in data payload. (Section 4.4)
c_valid	V	1	The sender is offering progress on an operation. (Section 4.1)
c_ready	R	1	The receiver accepted the offered progress. (Section 4.1)

3.6. Channel D (Mandatory)

Channel D flows from slave interface to master interface. It carries response messages for Channel A requests sent to a particular address. It also carries acknowledgements for Channel C voluntary writebacks. This channel is used by all TileLink conformance levels and is non-optional.

Table 9. Channel D signals

Signal	Type	Width	Description
d_opcode	C	3	Operation code. Identifies the type of message carried by the channel. (Table 12)
d_param	C	2	Parameter code. Meaning depends on d_opcode; specifies permissions to transfer or a sub-opcode. (Section 7.2, Section 8.2, Section 9.3)
d_size	C	***z***	Logarithm of the operation size: 2^n bytes. (Section 4.5)
d_source	C	***o***	Per-link master source identifier. (Section 6.4)
d_sink	C	***i***	Per-link slave sink identifier. (Section 6.4)
d_denied	C	1	The slave was unable to service the request. (Section 4.4)
d_data	D	***8w***	Data payload for messages with data. (Section 4.5)
d_corrupt	D	1	Corruption was detected in the data payload. (Section 4.4)
d_valid	V	1	The sender is offering progress on an operation. (Section 4.1)
d_ready	R	1	The receiver accepted the offered progress. (Section 4.1)

3.7. Channel E (TL-C only)

Channel E flows from master interface to slave interface. It carries acknowledgements of receipt of Channel D response messages, which are used for operation serialization. This channel is used at the TL-C conformance level and is optional in lower levels.

Table 10. Channel E signals

Signal	Type	Width	Description
`e_sink`	C	*i*	Per-link slave sink identifier. (Section 6.4)
`e_valid`	V	1	The sender is offering progress on an operation. (Section 4.1)
`e_ready`	R	1	The receiver accepted the offered progress. (Section 4.1)

4. Serialization

The five channels in TileLink are implemented as five physically distinct unidirectional parallel buses. Each channel has a sender and a receiver. For the A, C, and E channels, the agent with the master interface is the sender and the agent with the slave interface is the receiver. For the B and D channels, the agent with the slave interface is the sender and the agent with the master interface is the receiver.

Many TileLink messages contain a data payload, which, depending on the size of the message and data bus, may need to be spread out across multiple clock cycles (or *beats*). A multi-beat message is often called a *burst*. TileLink messages without a data payload are always exchanged in a single beat. It is forbidden in TileLink to interleave the beats of different messages on a channel. Once a burst has begun, the sender must not send beats for any other message until the last beat of the burst has been accepted by the receiver. The duration of a burst is determined by the channel's `size` field.

Progress on an operation is regulated by the exchange of beats between sending and receiving agents on a particular channel. The sender of a beat raises the channel `valid` signal to offer the availability of a beat on the channel. Receivers raise the `ready` channel signal to indicate their ability to accept a beat. The receiver lowers the `ready` signal to indicate that they are busy and are not accepting a beat. Only when both `ready` and `valid` are raised concurrently is the content of the beat considered exchanged.

The rest of this chapter lays out the flow control rules that govern when `ready` and `valid` may be toggled to exchange a beat of a message, and also defines rules for how request/response message pairs can be ordered. We finally discuss interfacing with legacy bus standards, error handling, and how bursted data is mapped onto a physical data bus of a particular width.

4.1. Flow Control Rules

In order to implement correct ready-valid handshaking, these rules must be followed:

- If `ready` is LOW, the receiver must not process the beat and the sender must not consider the beat processed.
- If `valid` is LOW, the receiver must not expect the control or data signals to be a syntactically correct TileLink beat.
- `valid` must never depend on `ready`. If a sender wishes to send a beat, it must assert `valid` independently of whether the receiver signals that it is `ready`.
- As a consequence, there must be no combinational path from `ready` to `valid` or any of the control and data signals.
- A low priority `valid` may not combinationally depend on a high priority `valid`. In other words, the decision to send a request may not be based on receiving a response in the same cycle.
- A high priority `ready` may not combinationally depend on a low priority `ready`. In other words, acceptance of a response may not be made contingent upon a request being accepted the same cycle.

Anything not forbidden is allowed. In particular:

- It is acceptable for a receiver to drive `ready` in response to `valid` or any of the control and data signals. For example, an arbiter may lower `ready` if a `valid` request is made for an address which is busy. However, whenever possible, it is recommended that `ready` be driven independently so as to reduce the handshaking circuit depth.

- A channel may change `valid` and all control and data signals based on the value of `ready` in the prior cycle. For example, after a request has been accepted (`ready` HIGH), a new request may be presented. Only a same-cycle dependency of `valid` on `ready` is forbidden.

- A device may legally drive `valid` for a response based on `valid` of a request in the same cycle. For example, a combinational ROM which answers immediately. In this case, presumably `ready` for the request will likewise be driven by `ready` for the response. The converse relationship is forbidden.

Note that a sender may raise `valid` and then lower it on the following cycle, even if the message was *not* accepted on the previous cycle. For example, the sender might have some other higher priority task to perform on the following cycle, instead of trying to send the rejected message again. Furthermore, the sender may change the contents of the control and data signals when a message was not accepted.

On TileLink channels which can carry bursts, there are additional restrictions. A burst is said to be *in progress* after the first beat has been accepted and until the last beat has been accepted. When a burst is in progress, if `valid` is HIGH, the sender must additionally present:

- Only a beat from the same message burst.

- Control signals identical to those of the first beat.

- Data signals corresponding to the previous beat's address plus the data bus width in bytes.

If the first beat of data in a burst is rejected, the sender may choose to attempt a different message on the following cycle (including initiating a different burst). If a beat of data of a burst that is already in progress is rejected, the sender may choose to change the values of the data the next time the beat is presented. However, whatever value of data is presented on the next valid beat will continue to correspond to the same address, until some offered beat is accepted by the receiver. Only accepted beats cause the burst to make progress through the address range associated with the beats of the data payload. Control signals for a burst that is in progress must remain constant for all valid beats until the burst is complete.

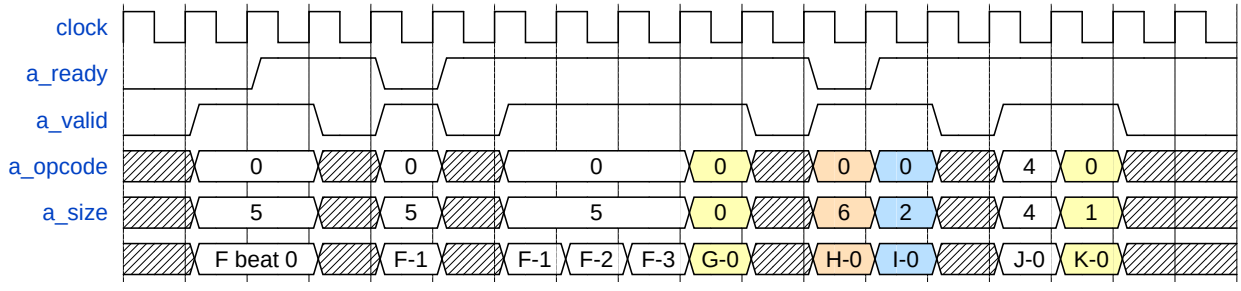

Figure 6. Ready-Valid signalling for 6 messages in a 64-bit Channel A).

One waveform which obeys these rules is illustrated for an 8-byte-wide channel in Figure 6. There are 6 messages sent in this figure: `F`, `G`, `H`, `I`, `J`, `K`. Notice that the validity of all control and data signals are predicated on `valid` HIGH. A beat is exchanged only when both `ready` and `valid` are HIGH.

The first message, F, has size 5, which indicates the operation accesses 2^n = 32 bytes. Opcode 0 is a `PutFullData` message, so F carries data. Because this particular Channel A is wide enough to transmit 8-byte beats, there are 4 beats of data to exchange. These are indicated as `F-0`, `F-1`, `F-2`, and `F-3`. The first cycle on which `F-0` is presented, the slave does not accept it. The master chooses to repeat `F-0` and it is then accepted. After `F-0` is accepted, burst F is considered in progress. Therefore, the master has no choice but to repeat `F-1` until it is accepted. However, the master is still free to lower `valid` during the burst. The master then continues to present beats of F in order, as it must, until the last beat `F-3` is accepted.

The second message, G, has size 0, indicating a 1 byte message. This size can be sent in a single beat and is exchanged immediately. Message H (an 8 beat burst) was presented by the master, but rejected. As the first beat of the burst was not accepted, the burst is not in progress and the master chooses to present a different message, `I`, on the following cycle instead. Message H need never be sent.

Message J has opcode 4, which on Channel A indicates a `Get`. Even though the Get operates on 16 bytes as indicated by `a_size`, message J itself carries no data, and thus fits in a single beat, which is accepted immediately. Message K can then be issued and accepted the following cycle.

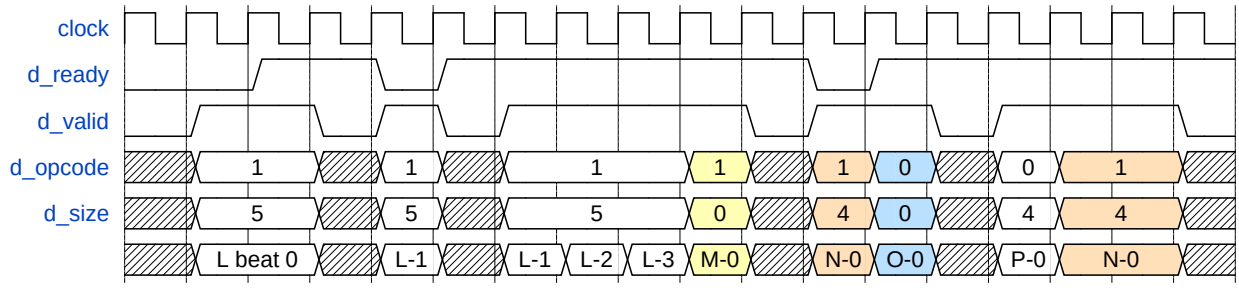

Figure 7. Ready-Valid signalling for 6 messages in a 64-bit Channel D).

Another waveform which obeys these rules is illustrated for an 8-byte-wide D channel in Figure 7. There are 5 messages sent in this figure: `L`, `M`, `N`, `O`, P. These are repsonse messages sent on Channel D. They present an example unrelated to the sequence of example request messages used in Figure 6, but they do illustrate that the same set of ready-valid-based flow control rules apply to both request and response channels.

The first message, L, has size 5, which indicates the operation accessed 2^5 = 32 bytes. Opcode 1 is a `AccessAckData` message, so L carries data. Because this particular Channel D` carries 8-byte beats, there are 4 beats of data to exchange. These are indicated as `L-0`, `L-1`, `L-2`, and `L-3`. The first cycle on which `L-0` is presented, the master does not accept it. The slave chooses to repeat `L-0` and it is then accepted. After `L-0` is accepted, the burst L is considered in progress. Therefore, the slave

has no choice but to repeat L-1 until it is accepted. However, the slave is still free to lower valid during the burst. The slave then continues to present beats of L in order, as it must, until the last beat L-3 is accepted.

The second message, M, has size 0, indicating a 1 byte message. This size fits into a single beat and is exchanged immediately.

Message N was presented by the slave, but rejected. As the first beat of the burst was not accepted, the burst is not in progress and the master chooses to present a different message, 0, on the following cycle instead. To complete operations in TileLink all response messages must eventually be sent, because all requests must eventually receive a response. Unlike with the rejected messages in the previous example, Message N must eventually be offered again for transaction completion reasons.

Message 0 has opcode 0, which on D indicates a AccessAck. Even though the AccessAck is in response to an operation performed on 16 bytes as indicated by d_size, message 0 itself carries no data, and thus fits in a single beat, which is accepted immediately.

Message N is then issued and finally accepted on the following cycles.

4.2. Request-Response Message Ordering

We now define the rules governing when response messages can be sent, with a particular emphasis on bursts that contain multiple beats.

The first beat of the response message is allowed to be valid:

- on the same cycle that the first beat of the request is valid, but not before.
- after an arbitrary amount of time following the first beat of the request message, but only if ready and valid were high on the request channel during the first beat of the request message.

This means that on the same cycle for the first beat of both the request message and the response message, ready high on the response channel implies ready high on the request channel. In particular, valid high and ready low on the request channel and valid high and ready low on the response channel are legal. However, valid high and ready low on the request channel is illegal if valid high and ready high are on the response channel.

Beats following the first beat of a burst response message may also be presented after an arbitrarily long delay, but no beats from other messages may be interleaved between.

The fact that a response message can be received concurrently and combinationally with the first beat of the request interacts with the forward progress rules in Section 5. Those rules govern when an agent receiving a response may present e.g. d_ready LOW while d_valid is HIGH.

For example, a designer might be tempted to implement a master interface which holds d_ready LOW while a_valid is HIGH in order to delay a concurrent response message until the following cycle. However, this represents an indefinite delay on Channel D that is not allowed by any of the forward progress rules. Indeed, a TL-UL–conforming slave interface may have connected d_valid and d_ready to a_valid and a_ready respectively. Thus, the non-conforming master interface has introduced a deadlock.

If a master interface cannot deal with receiving a response message on the same cycle as its request message, then it can instead put a register stage after its Channel D input. The registers absorb a concurrent Channel D response message and present `d_ready` HIGH until it has been filled. This response handling logic satisfies the forward progress rules while allowing the slave to respond as quickly as possible. All agents must follow the rules: either proactively deal with the possibility of a concurrent response, or place a buffer on the receiving input port to absorb it.

The following subsections elaborate on the interaction of request and response messages of different burst sizes.

4.2.1. Burst Responses

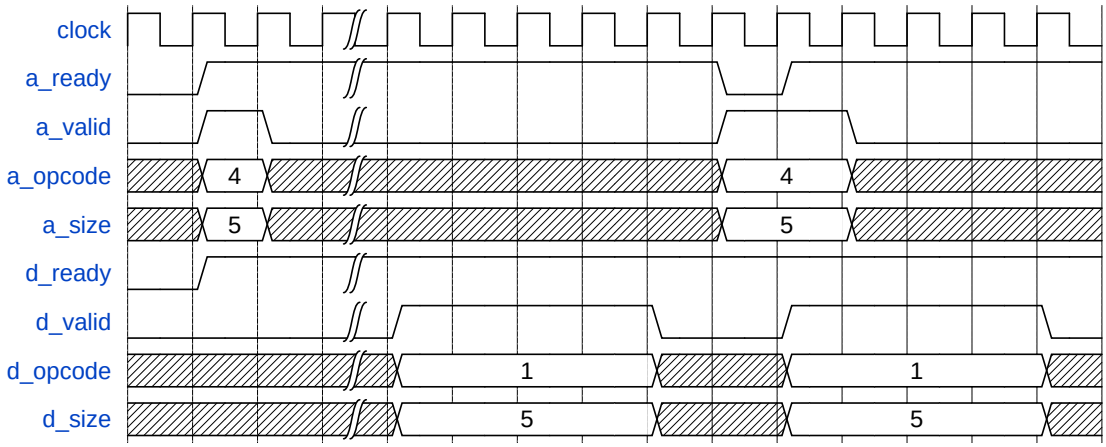

Figure 8. The max and min delay between a Get (4) and an AccessAckData (1) on and 8-byte bus.

Figure 8 illustrates two Get operations. The `Get` request messages (opcode 4) are sent out Channel A. They are both accessing $2^n = 32$ bytes, which takes 4 beats of data on an 8-byte bus. We see their 4-beat `AccessAckData` (opcode 1) arriving on `Channel D`. The first response message arrives after an arbitrary delay. The master interface must be willing to wait indefinitely for this reponse message, as timeouts within the TileLink network are forbidden. Eventually, the response message arrives, which is guaranteed by TileLink's deadlock freedom.

The second `Get` is responded to within the same cycle as the request message itself is accepted. This overlap is allowed as soon as the first beat of the `Get` is accepted. The response message was presented no earlier: as `a_ready` was LOW when the second `Get` was first presented, the request was rejected, and so `d_valid` also had to be LOW or the first rule would have been violated.

4.2.2. Burst Requests

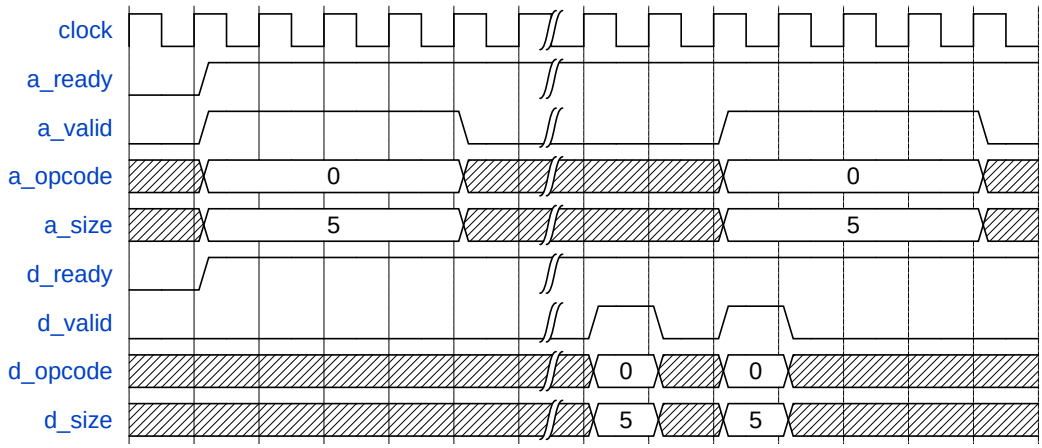

Figure 9. The max and min delay between an PutFullData (0) and an AccessAck (0) on and 8-byte bus.

Figure 9 illustrates two Put operations. The PutFullData request messages (opcode 0) are sent out Channel A and their AccessAck response messages (opcode 0) come back on Channel D. Again, the size is 2^5 = 32 bytes = 4 beats. However, this time it is the Channel A request message that is a burst. As their names indicate, a PutFullData message carries a data payload, whereas an AccessAck does not.

The first AccessAck message is delayed for an arbitrary amount of time, but the requestor continues to send the rest of the burst request message.

The second AccessAck message is presented on the same cycle as the first of the PutFullData message. This is the earliest response allowed by the rules. If either a_ready or a_valid had been LOW on that cycle, then d_valid would have also been LOW. The previously discussed ready caveat for the master interfaces applies here: the master interface must accept a concurrent AccessAck, even before it has finished sending the PutFullData message. It may, however, buffer the AccessAck message and leave it pending there until it completes sending the request.

4.2.3. Burst Requests and Responses

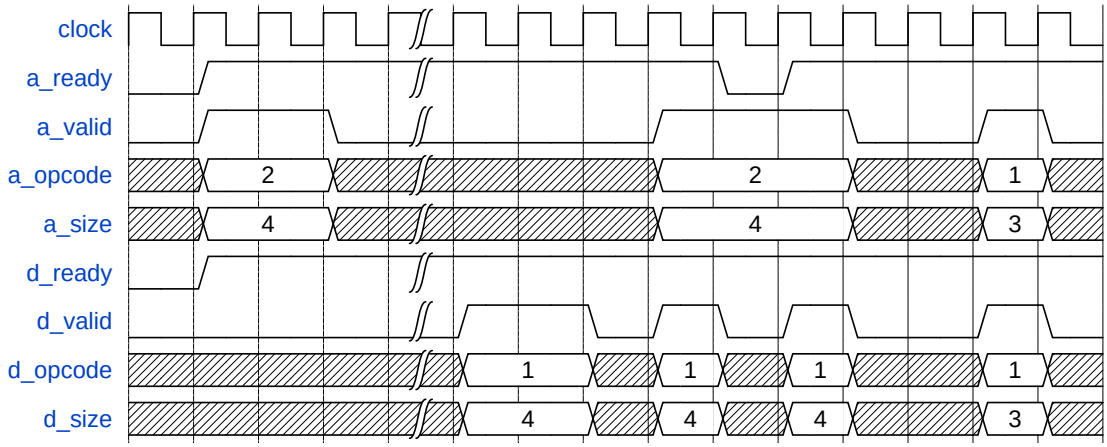

Figure 10. The legal delay between an `ArithmeticData` (2) and an `AccessAckData` (1) on an 8-byte bus

The situation for request messages and response messages that both carry a data payload follows the same rules. The first of the response messages must be presented no earlier than the first beat of the request message, but may be delayed arbitrarily. Additional response beats may be delayed arbitrarily, and for most operations of this sort will be delayed for at least as long as it takes to accept the corresponding beats of the request message.

Figure 10 illustrates Atomic operations that consist of a request message and response message that both carry data. For the 2^4 = 16 byte = 2 beat operations, there can be either a long delay between request message and response message, or the beats of both may overlap. Response beats may be delayed if they require data from the corresponding request beats. If the entirety of each message fits within a single beat (2^3 = 8 byte = 1 beat), the messages may overlap completely.

4.3. Interfacing with Legacy Buses

Unfortunately, older buses do not guarantee forward progress. When controlling these buses, it would violate TileLink's `ready` rules if the bridge were to block TileLink traffic indefinitely while waiting for the legacy bus to accept a message. Therefore, bridges to buses like AXI must include a timeout, to fit within the auspices of the forward progress rules. If the legacy bus does not accept a request within this timeout, the request must be discarded and a TileLink error response inserted.

If a legacy bus sends response messages, a bridge must also put a limit on how long it will wait for those responses, unless the legacy bus can be verified to be deadlock free. If an unverified legacy bus exceeds the time limit, the bridge must cancel the outstanding request, inject a TileLink error response, and if the original legacy response ever arrives, discard it. To put a limit on the memory required to track discarded responses, it is acceptable for a bridge to completely disable a deadlocked legacy bus.

Inside the TileLink network itself, timeouts that cause alternative messages to be generated are expressly forbidden. TileLink agents waiting on other TileLink agents must be infinitely patient. However, this does not preclude TileLink watchdogs which trigger reset. TileLink is only deadlock free when all agents conform to this specification. If one is not confident in the quality of all TileLink agent implementations included in a given network, a watchdog can help.

4.4. Errors

There are two types of errors that may be communicated across TileLink networks: corrupt data errors and access denied errors.

Data corruption is signaled alongside the data on a channel. Any individual beat of data in a burst may be marked as corrupt by an agent producing the message. A typical instance of corruption occurs when an unrecoverable error is detected by ECC protection in an agent that was storing a copy of the data.

The `corrupt` signal is present on all channels that carry data. However, only message types that carry data may have beats marked as corrupt. Certain beats may be marked as corrupt while others are not.

The ones that are not marked corrupt still contain valid data. Every TileLink request message requires a mandatory response message of a mandatory size, and all beats of the message must be sent, even if every beat is marked as corrupt.

Access `denied` is a single bit control signal that indicates whether an attempted access or permissions transfer operation was processed by the recipient or not. When an operation is denied, it must have no effect on the permissions of the data block, nor change its contents, nor trigger any side effects related to accessing the data.

This `denied` control signal is only present on the D channel, which is the only channel for responses to A-channel requests, which are the only type of requests which require increasing permissions. All other requests are guaranteed sufficient permissions and so may not be denied. For example, master agents that cache blocks of data must always restore permissions to their slave upon request. Thus, they are not permitted to deny permissions transfers related to the copy.

Denial of access may be a permanent or a transient condition.

When a response message that carries data is denied, it must mark all beats of the message as corrupt.

How errors that have been signaled via either corrupt or denied fields are reported to the broader system is beyond the scope of this document. For example, interrupts, traps, or exceptions may be used to notify software that an error has occurred.

4.5. Byte Lanes

TileLink channels which carry a data field always carry payload data little-endian naturally aligned. In other words, the zeroth byte lane carries the data found at the address specified by the operation. Furthermore, if the data bus is 16-bytes wide, then the byte lanes of the bus always carry data for the same lowest nibble of the address; see Figure 11. Not all byte lanes are used if the `size` is smaller than the data bus width. Multi-beat burst operations increment their data addresses between beats, while their control signals remain constant.

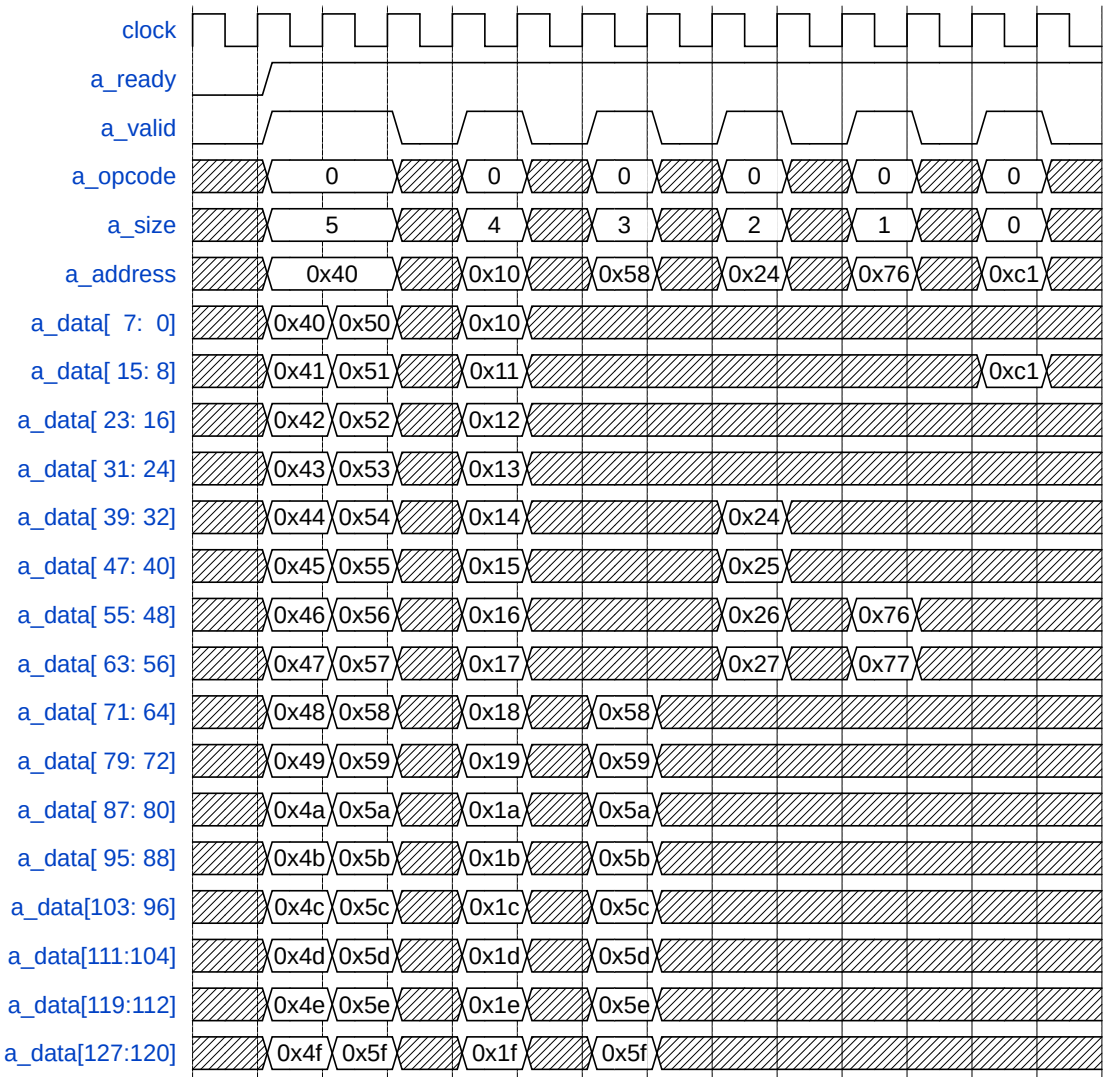

Figure 11. An example of the addresses of data carried in byte lanes on a 16-byte data bus.

TileLink operations always describe power-of-two-sized byte ranges with an aligned address. Mathematically, $(address \ \& \ (2^{size} - 1) = 0)$ always holds. Therefore, either an operation uses all of the data byte lanes or it uses a power-of-two-sized slice of them. The byte lanes used by an operation are called the *active* byte lanes. In Figure 11, the inactive byte lanes are crossed out.

On channels A and B, which carry a `mask` field, the `mask` must always be LOW for all inactive byte lanes. Furthermore, for all messages other than `PutPartialData`, the bits of `mask` for all active byte lanes must be HIGH. `PutPartialData` may lower individual bits of the mask and these bits do not have to be contiguous.

The `mask` is also used for messages without a data payload. When the operation `size` is smaller than the data bus, the `mask` should be generated identically to an operation which does carry a data payload. For data-less operations which are larger than the data bus, all bits of the `mask` should be HIGH, although the message remains a single-beat. See, for example, Figure 12.

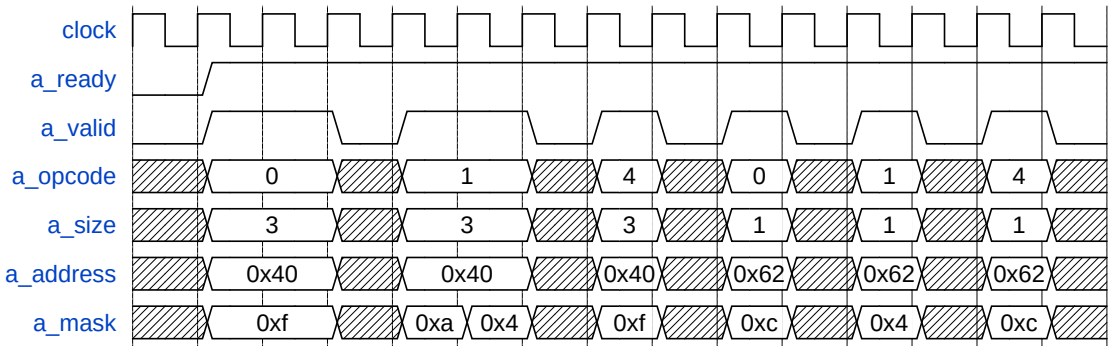

Figure 12. An example of the mask bits carried by byte lanes on a 4-byte data bus.

In Figure 12, `PutFullData` (0) must drive all active lanes of `mask` HIGH. Thus, the first message has all beats HIGH over multiple beats. In comparison, `PutPartialData` (1) may drive active lanes of `mask` HIGH or LOW for all beats. `Get` (4) messages are never multi-beat, but must still drive `mask` HIGH on active byte lanes. For messages smaller than a beat, all inactive byte lanes of `mask` must be driven LOW (bits 0 and 1 in the operations addressing 0x62).

5. Deadlock Freedom

TileLink is designed to produce deadlock-free systems by construction. To guarantee that a TileLink network will never deadlock, we specify a set of rules to which conforming agents and systems must adhere. Should a system deadlock in practice, the rules make it clear which component is faulty. This chapter first outlines the rules in human understandable form, followed by careful definition of the terms involved. After presenting conforming and non-conforming examples, a sketch of the proof will conclude this chapter.

Forward progress in a quiescent TileLink network requires adherence to these rules:

1. The agent graph (Section 5.3) contains no cycles
2. Agents must eventually present all beats of a received message
3. Unless they have a higher priority message in flight or unanswered
    i. Agents must eventually accept a presented beat
    ii. Agents must eventually answer a received request message

To meet these rules, agents may make these technical assumptions:

- To satisfy rule X, an agent may assume all channels obey rules < X.
- To satisfy rule 2, an agent may assume all lower priority channels obey rule 2.
- To satisfy rule 3, an agent may assume all higher priority channels unconditionally obey rule 3i and rule 3ii.

5.1. Definition of Terms

To correctly interpret the forward progress rules, the terms involved require precise definitions:

accept a beat

When a sender drives a channel with `valid` HIGH, the receiver **accepts** that beat if it drives `ready` HIGH.

reject a beat

When a sender drives a channel with `valid` HIGH, the receiver **rejects** that beat if it drives `ready` LOW.

retract a beat

When a receiver rejects a beat, the sender may **retract** that beat by lowering `valid` or modifying the control or data signals on the next cycle. As per Section 4.1, these changes are illegal to

perform on the same cycle that `ready` is lowered.

present a beat

A beat is **presented** if `valid` is held HIGH and all control and data signals are held constant forever or until `ready` is also HIGH. This definition essentially describes sender behavior. If the sender ever retracts a beat in the future, then that beat was not presented. Conversely, if a sender commits to sending a beat until it is accepted, then that beat was presented. TileLink only guarantees that presented beats are eventually accepted.

in progress (a message)

A message is **in progress** from the cycle when the first beat is accepted up to and including the cycle when the last beat is accepted.

in flight (a message)

A message is **in flight** from the cycle when the first beat is presented up to and including the cycle when the last beat is accepted. If the first beat is rejected, that message will be in flight earlier than when it is in progress.

request message

A **request message** requires a response message. All operations in TileLink begin with a request message on some channel.

response message

A **response message** is the mandatory follow-up message paired to a request. Notice that Grants are both request and response messages.

received message

A message is **received** when the first beat of the burst has been accepted.

answered message

An **answered** message is a received request with a received response. An unanswered message is a received request without a received response. As per Section 4.1, it is illegal to receive a response without receiving the request.

agent

An **agent** is a participant in the TileLink system that possesses one or more TileLink links.

> Agents may only leverage the rule 3 exemption for their own links.

For example, consider a device which copies memory from one address to another using a stream of Gets and PutFullDatas on a DMA link. To control this device, suppose there is another memory-mapped interface connected to a control link.

If the device withholds answers to control requests until the copy operation has completed, then both the DMA and control link must be considered as connected to the same agent.

If the master answers all control requests immediately while separately running a decoupled state machine to drive the DMA link, these two links can be considered as connected to two distinct agents.

This distinction is very important, because of rule 1. The decoupled device can be safely used in systems where the coupled device would have introduced a cycle in the graph.

It is also important to note that a single agent might be very large, composed out of many modules all over a chip. A mesh interconnect that spans the entire chip can be a single agent. The fact that the mesh internally includes cyclic message forwarding paths is irrelevant to TileLink. The mesh need only adhere to the TileLink rules at the boundaries where it has links.

priority

In TileLink, there is **priority** ordering of an agent's links' channels. A message has a priority which corresponds to the channel on which it is sent. On every link, an agent is either the master or the slave. In increasing priority order:

- channel A on links where the agent is the slave/receiver
- channel A on links where the agent is the master/sender
- channel B on links where the agent is the master/receiver
- channel B on links where the agent is the slave/sender
- channel C on links where the agent is the slave/receiver
- channel C on links where the agent is the master/sender
- channel D on links where the agent is the master/receiver
- channel D on links where the agent is the slave/sender
- channel E on links where the agent is the slave/receiver
- channel E on links where the agent is the master/sender

Notice that response messages are always carried by channels with a higher priority than their requests. Also note that agents which need to forward messages always use a channel with higher priority than the channel from which they receive. These two properties make it fairly easy to follow the forward progress rules.

eventual event

An event **eventually** occurs after an arbitrarily long, but not infinite amount of time. There are essentially only two ways to demonstrate this. Either there is a proven upper-bound on the number of cycles until the event occurs, or the event occurs after another eventual event occurs. Combinations of these two scenarios are possible.

forward progress

The system has **forward progress** when:

- All presented beats in the system are eventually accepted.
- All received requests are eventually answered.

quiescent

A system is **quiescent** when no new messages are added. In TileLink, while deadlock is impossible, starvation (also called livelock) is possible. Concretely, it is possible for one agent to completely monopolize a resource by continuously requesting it. In this case, a different agent may be unable to complete its competing operation. By assuming that eventually no new messages are added, livelock scenarios become impossible, and we can prove forward progress.

To formulate this restriction formally, agents are only allowed to transmit $(n + rf)$ messages each:

- n is the maximum number of *new* messages an agent is allowed to send.
- r is the number of messages an agent receives via its links.
- f is the maximum number of *follow-up* messages an agent is allowed to send per received message.

5.2. Examples of agent conformance

While the forward progress rules are quite pithy, examples of specific agent behaviors can help to explore the rule's practical implications a bit more concretely.

Periodic forwarding master (non-conforming)

Consider an agent with two links, one on which it is a master and one a slave. Suppose it receives an A-channel request message, but when forwarding the message, it only raises `valid` HIGH on every even cycle. This agent violates the specification, because it fails to present the forwarded beat, and so cannot be guaranteed to eventually answer a received request.

If it were not alternatingly `valid`, the following argument would apply:

- The forwarded request was presented (this is the step that fails).
- The forwarded request is on a higher priority channel (master A vs. slave A).
- By the technical assumptions, we know that it will be accepted (ie: received).
- By the technical assumptions, we then also know it will be eventually answered.
- We can then forward the response to the originator, meeting our requirements.

Waiting for a refresh (conforming)

Consider a DDR controller which is periodically unable to service requests. During these periods, it unconditionally lowers all `ready` signals. Unlike periodic forwarding, periodic readiness is legal, because the agent will still eventually accept presented beats.

Withholding beats of a received message (non-conforming)

Consider an agent which has two links it uses to send messages. Suppose it starts burst A on link 1, and the first of several beats is accepted, meaning the message has been received, but is still in flight. Then it starts a burst B on link 2, leaving burst A incomplete. It would be non-conforming to then wait for beats of B to be accepted before resuming burst A.

The problem is that this agent has violated rule 2 for link 1. Normally, it would be true that link 2 eventually accepts all beats of burst B (rule 3i), by technical assumption. However, this is not a legal assumption to use in proving rule 2! To see why this could scenario could produce deadlock, consider how this agent interacts with the next, conforming, agent.

Very slow arbiter (conforming)

Consider a TL-UH agent with three links. It arbitrates A-channel requests from two links and forwards them out the A channel of the third link. However, this arbiter has very low throughput.

When the agent is idle, it will select a `valid` request from either incoming A channel. Once a request has been selected, it lowers `ready` on the other incoming A channel and connects `ready` and `valid` between the selected link and the outgoing link (for both channels A and D). Once the request has been received, the agent is busy and the selection is fixed. As soon as all beats of the request have been accepted by the outgoing channel, the agent lowers `ready` on both incoming A channels. It then waits until the last beat of the response message has been accepted by the selected channel, whereupon it transitions back to the idle state.

This agent meets rule 2, because it can assume that both incoming A channels obey rule 2 and that therefore its outgoing A channel request will present all beats of a received request. A similar argument applies for the D-channel.

Suppose the agent is either busy or has a presented incoming message. Then it has respectively either an unanswered or in-flight message on the forwarding link. As that link is higher priority than the incoming links, rule 3 applies and it is legal for the agent to reject further presented A-channel requests indefinitely.

Suppose the agent is not busy and there are no presented incoming messages. In that case, rule 3i is upheld because there is no presented message to accept. Furthermore, rule 3ii is upheld because we would only be idle again if the received request has received its answer.

5.3. The Agent Graph

Every TileLink network is composed out of TileLink links and the agents those links connect. This network can be represented as an agent graph, where every agent is a vertex and every link is a directed edge pointing from master to slave. Technically, the agent graph is a multigraph, since it is legal to connect two agents with multiple links, but we will continue to call it simply a graph in this specification. Figure 13 illustrates an example of a TileLink agent graph. Blue boxes indicate RTL modules, while yellow circles indicate agents.

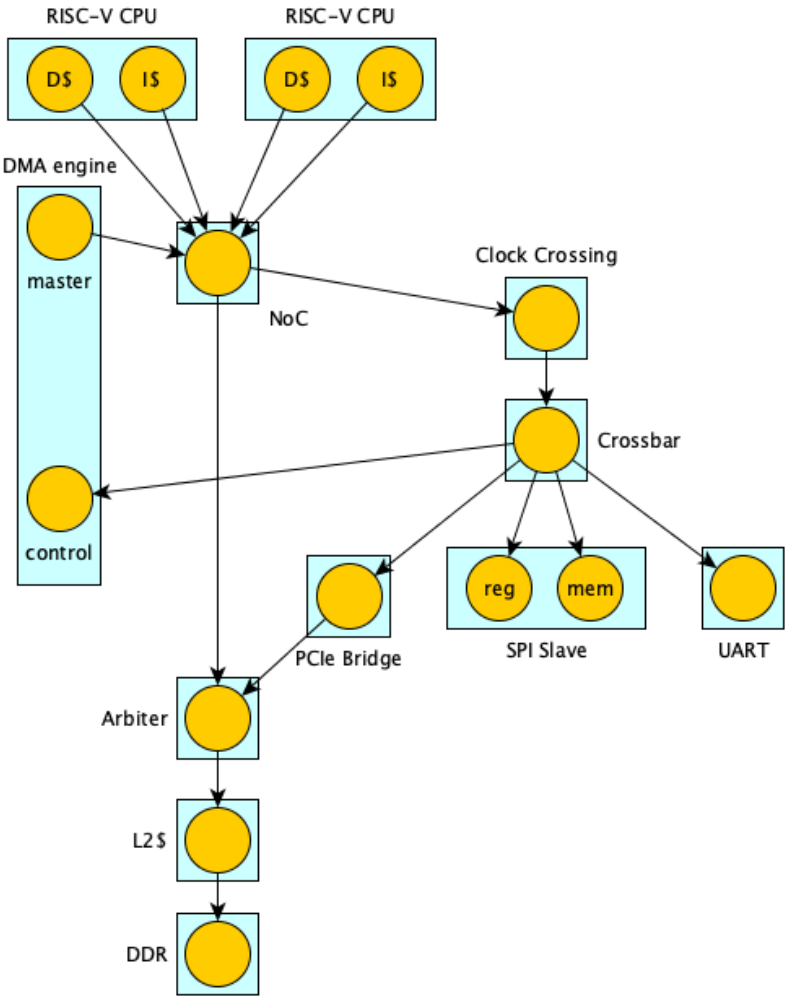

Figure 13. An agent graph example of a small RISC-V system.

Recall that rule 1 demands that the agent graph contain no cycles. A nice convention when drawing agent graphs is to arrange the agents such that all the edges point down. This makes it visually obvious that there are no cycles in the graph.

The example graph includes two RISC-V cores, each with an instruction and data cache. Even though each core is a single module, the caches are two distinct agents. Typically, caches operate autonomously from each other; there is no message coupling (ala rule 3) between them. Thus, the links can be connected to two independent agents in a shared module.

Slightly less obvious is the DMA engine. In order to allow the master and control links to be connected as they are, without a cycle, the engine needed two independent agents. This means that the control interface must be capable of answering requests without waiting for DMA mastering operations to complete.

Somewhat innocuously drawn as a single agent is the network on chip (NoC). This might be implemented as a hypothetical token ring network, which circles around the chip, dropping off messages at the connected TileLink links. The agent graph often wildly misrepresents the physical

layout and size/complexity of both the physical modules and logical agents. Note that the presence of a cycle inside the token ring agent microarchitecture does not violate the forward progress guarantees of TileLink, so long as the forward progress rules are upheld by the TileLink links.

One often-problematic type of device in systems is PCI express (PCIe) bridges. Ordering rules within PCIe have the effect of coupling forward progress of requests sent *to* a PCIe bridge to forward progress of requests sent *from* the PCIe bridge. Due to this coupling, it is necessary (by rule 3) to represent the PCIe bridge using a single agent. In this agent graph, that forced us to restrict PCIe DMA to have visibility of just main memory, and not to the peripheral crossbar (which includes PCIe).

5.4. Forward progress proof sketch

To better understand the forward progress rules, a bird's-eye view of the proof can help. This section outlines the basic argument, which proceeds in four parts. First, the agent graph is rewritten into a channel graph. Then, a suitably distant point in time is selected. A global version rule 2 of is established. Finally, a global version of unconditional rule 3i and rule 3ii are established, satisfying the definition of forward progress, and completing the proof.

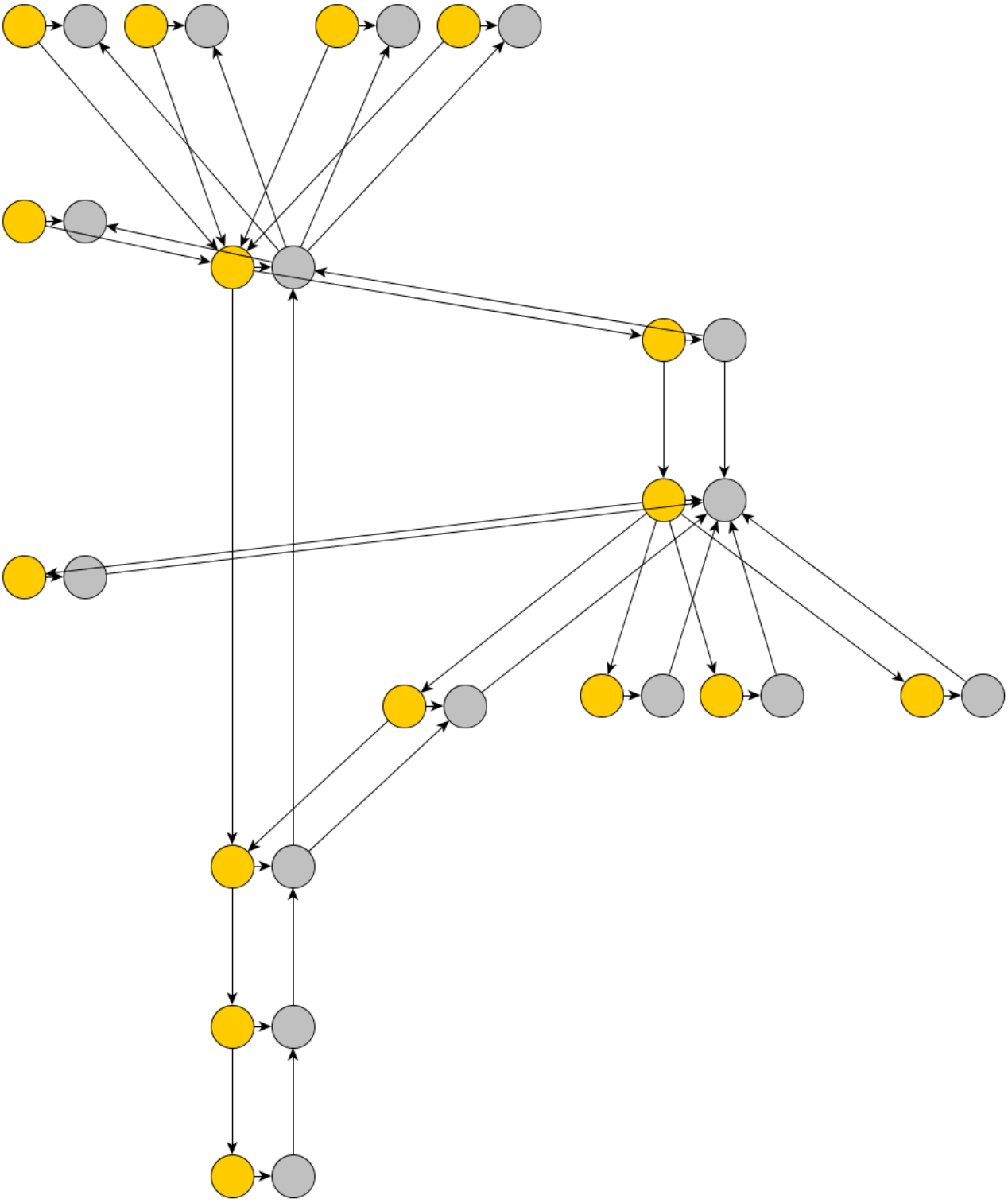

Figure 14. An A+D channel graph built from Figure 13. Yellow node are priority A. Grey nodes are priority D.

Given a directed acyclic agent graph, we must transform it into a directed acyclic channel graph. To do this, replicate every agent into five copies and label them A-E, one for each channel. Then, reverse all

of the arrows which connect nodes of type B and D. Finally, connect the node replicas such that A points to B points to ... E. An example of this for two channels (A+D) can be seen in Figure 14.

Now we prove that the channel graph is directed acyclic graph (DAG). Reversing all the arrows in a DAG is still a DAG. Each priority level is thus still a DAG. Imagine the five DAGs stacked vertically. Clearly, the arrows between DAGs all ascend, so adding these arrows cannot create a cycle. Therefore, the channel graph is a DAG.

A topological sorting assigns a number to every node in a graph. Edges in a sorted graph always point from a lower number to a high number. Topological sortings always exist for DAGs. Label the channel graph nodes with such a numbering. For any given agent, sort its channels/edges by the labels of the connected agents. Notice that this sort order puts the channels in the same priority order as defined in Section 5.1.

The rest of the proof will assume that we are inspecting the state of the system after a *very long time*. Roughly, a point in time beyond which:

- the system is quiescent, meaning no new messages are being created.
- every eventual event has occured.

We now establish rule 2 for all channels in the system. Label all edges in the system by the sender agent that it connects. We will run strong induction over edges in increasing label order, proving rule 2 for each of them. Consider the sender agent for the current edge. We know that rule 1 is upheld. All channels of lower priority (in the sense of the forward progress rules) have lower labels. Therefore, by the induction hypothesis, they all uphold rule 2. In conclusion, we have met the sender agent's assumptions and have thus proven rule 2 for this edge.

Similarly, we will now establish rule 3i, rule 3ii, and the absence of in-flight and unanswered message by strong induction. In this case, label the edges in the system by the receiver agent they connect. We will run strong induction in decreasing label order. We have already established that both rule 1 and rule 2 are upheld by all edges in the system. All channels of higher priority have higher labels, and by the induction hypothesis they all carry no in-flight message or unanswered messages and they all satisfy rule 3i and rule 3ii unconditionally. Therefore, there are no higher priority message in-flight or unanswered. Also, all the requisite receiver agent assumptions have been met, so we may conclude that rule 3 holds for this edge and, furthermore, rule 3i and rule 3ii hold unconditionally. Finally, we establish that there are no in-flight or unanswered messages. We know that there eventually cannot be, since the receiver upholds rule 3i and rule 3ii. By earlier assumption, we have waited long enough such that all the eventually clauses have expired. Therefore, there can be no such messages, and the proof is complete.

6. Operations and Messages

TileLink agents with master interfaces interact with the shared memory system by executing *operations*. An operation effects a desired change to an address range's data value, permissions or location in the memory hierarchy. Operations are executed by the exchange of concrete messages which flow over the five TileLink channels. To support an operation, all of its constituent messages must be supported. This Chapter lists all of the Tilelink operations and the messages exchanged to implement them. We then detail the specific message exchange flow for each operation in the Chapters detailing the three TileLink conformance levels: in TL-UL (Section 7), in TL-UH (Section 8), and in TL-C (Section 9).

6.1. Operation Taxonomy

TileLink operation can be categorized into three groups:

- **Accesses** (A) read and/or write the data at a specified address.
- **Hints** (H) are informational only and have no direct effects.
- **Transfers** (T) move permissions or cached copies of data through the network.

Not every TileLink agent needs to support every operation. Depending on its TileLink conformance level, an agent only needs to support the matching operations listed in Table 11.

Table 11. Summary of TileLink Operations

Operation	Type	TL-UL	TL-UH	TL-C	Purpose
Get	A	y	y	y	read from an address range
Put	A	y	y	y	write to an address range
Atomic	A	.	y	y	read-modify-write an address range
Intent	H	.	y	y	advance notification of likely future operations
Acquire	T	.	.	y	cache a copy of an address range or increase the permissions of that copy
Release	T	.	.	y	write-back a cached copy of an address range or relinquish permissions to a cached copy

6.2. Message Taxonomy

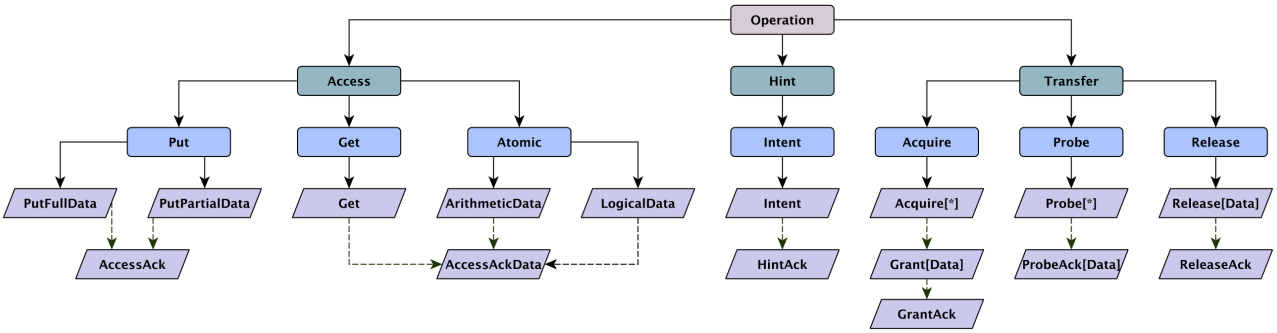

Figure 15. Taxonomy of operations.

Operations are executed by exchanging messages over the five TileLink channels. Some messages carry a data payload, while others do not. The name of a TileLink message always ends with `Data` if it carries a data payload. Not every channel supports every type of message. Receipt of request messages *must* result in the eventual receipt of a response message by the requestor.

A graphical representation of the operation and message taxonomy with responses shown can be seen in Figure 15. Operations(blue boxes) comprise their constituent messages (purple parallelograms). Dotted arrows indicate request-response message pairs. TL-UL conformance only requires supporting Get and Put Access operations. TL-UH conformance requires all Hint and Access operations. TL-C conformance requires all operations. Table 12 lists all messages used in TileLink, grouped by conformance level and operation. Table 13 presents the same information but ordered by channel and opcode.

Notice that multiple message types have the same opcode. Different channels have different namespaces for opcode numbering. Within any given channel, each possible message type has a unique opcode. Furthermore, the same message type has the same opcode regardless of the channel over which it is exchanged. Opcode space has been allocated for efficient decoding of message properties. Future editions of the spec reserve the right to add further opcodes.

The responses listed in the rightmost column of both tables are the only allowed responses to each message. Some messages allow for multiple response types depending on whether or not a copy of the data needs to be returned. Other messages have no expected/allowed response.

Table 12. Summary of TileLink messages, grouped by conformance level and operation.

Message	Operation	Opcode	A	B	C	D	E	Response
TL-UL								
Get	Get	4	y	y	.	.	.	AccessAckData
AccessAckData	Get or Atomic	1	.	.	y	y	.	
PutFullData	Put	0	y	y	.	.	.	AccessAck
PutPartialData	Put	1	y	y	.	.	.	AccessAck
AccessAck	Put	0	.	.	y	y	.	
TL-UH								
ArithmeticData	Atomic	2	y	y	.	.	.	AccessAckData
LogicalData	Atomic	3	y	y	.	.	.	AccessAckData
Intent	Intent	5	y	y	.	.	.	HintAck
HintAck	Intent	2	.	.	y	y	.	
TL-C								
AcquireBlock	Acquire	6	y	.	.	.	.	Grant or GrantData
AcquirePerm	Acquire	7	y	.	.	.	.	Grant
Grant	Acquire	4	.	.	.	y	.	GrantAck
GrantData	Acquire	5	.	.	.	y	.	GrantAck
GrantAck	Acquire	-	.	.	.	.	y	
ProbeBlock	Probe	6	.	y	.	.	.	ProbeAck or ProbeAckData
ProbePerm	Probe	7	.	y	.	.	.	ProbeAck
ProbeAck	Probe	4	.	.	y	.	.	
ProbeAckData	Probe	5	.	.	y	.	.	
Release	Release	6	.	.	y	.	.	ReleaseAck
ReleaseData	Release	7	.	.	y	.	.	ReleaseAck
ReleaseAck	Release	6	.	.	.	y	.	

Table 13. Summary of TileLink messages, ordered by channel and opcode.

Channel	Opcode	Message	Operation	Response
A	0	PutFullData	Put	AccessAck
	1	PutPartialData	Put	AccessAck
	2	ArithmeticLogic	Atomic	AccessAckData
	3	LogicalData	Atomic	AccessAckData
	4	Get	Get	AccessAckData
	5	Intent	Intent	HintAck
	6	AcquireBlock	Acquire	Grant or GrantData
	7	AcquirePerm	Acquire	Grant
B	0	PutFullData	Put	AccessAck
	1	PutPartialData	Put	AccessAck
	2	ArithmeticData	Atomic	AccessAckData
	3	LogicalData	Atomic	AccessAckData
	4	Get	Get	AccessAckData
	5	Intent	Intent	HintAck
	6	ProbeBlock	Probe	ProbeAck or ProbeAckData
	7	ProbePerm	Probe	ProbeAck
C	0	AccessAck	Put	
	1	AccessAckData	Get or Atomic	
	2	HintAck	Intent	
	4	ProbeAck	Probe	
	5	ProbeAckData	Probe	
	6	Release	Release	ReleaseAck
	7	ReleaseData	Release	ReleaseAck
D	0	AccessAck	Put	
	1	AccessAckData	Get or Atomic	
	2	HintAck	Intent	
	4	Grant	Acquire	GrantAck
	5	GrantData	Acquire	GrantAck
	6	ReleaseAck	Release	
E	-	GrantAck	Acquire	

6.3. Addressing

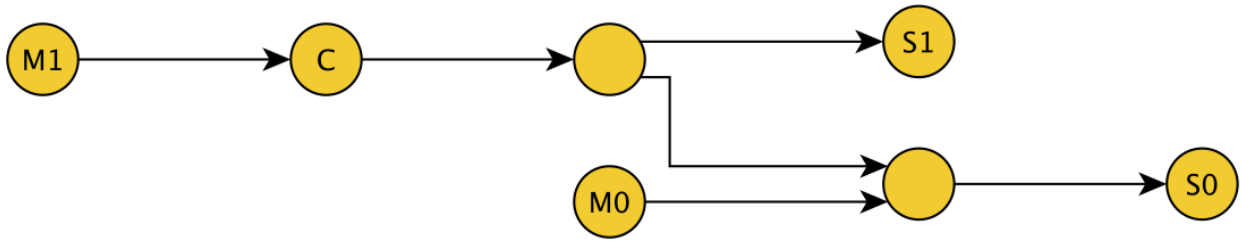

Figure 16. Two masters, M0 and M1, can both access slave S0, while only M1 can access S1, via a cache C.

All addresses carried by TileLink channels are physical addresses. From any node in the TileLink DAG, every valid address must route over a single path to exactly one slave. In TileLink, the address determines which operations are supported, which effects are generated, and which ordering restrictions are imposed. Properties that might be ascribed to an address space include its: TileLink conformance level, memory consistency model, cacheability, FIFO ordering requirements, executeability, privilege level, and any Quality-of-Service guarantees.

For example, when the master executes an operation on a particular address, it has no control over whether or not that request is cached; the network decides. If a particular slave has side effects on Get operations, then a cache placed between a master and that slave must not cache Get operations sent to that slave's addresses. Similarly, if a slave has side effects on Put operations, a cache must at least write-through Put operations sent to that slave's addresses. The specific mechanism by which these requirements are enforced is outside the scope of the TileLink specification.

We recommend that a System-on-Chip implementation create a local address map which describes which regions of memory have side effects. This mapping can then be used by a cache to determine if it is safe to cache a particular Get operation. Similarly, a crossbar can use the address map to determine down which port to route an operation.

If using an address map, we further advise that the address map not be a single global map. As one moves through the TileLink network, some properties of the address map can change. For example, consider Figure 16. Master M1 can access both slaves S0 and S1, while master M0 can only access slave S0. Beyond mere reachability, some TileLink agents may change the properties of slaves behind them. For example, the cache C in Figure 16 may cause the address range of slave S1 to support atomic operations, which the original slave did not support.

When it is not possible to know a-priori what sort of slave devices will be attached to a given address range, the rest of the TileLink network must define what it expects. For example, one can be conservative and suppose that all operations to the external address range have both Get and Put effects, or one can be optimistic and require that only side-effect free devices will be attached. When exposing a blind TileLink slave port, the port should be accompanied with documentation describing the properties of the addresses behind the port. Similarly, when exposing a blind TileLink master port, the port should be accompanied with documentation describing what assumptions the master has made about the addresses behind the port.

We strongly recommend that if an address region has any Get or Put side effects that the address region be rounded up and down to the next nearest multiple of 4kB. This makes it much easier for a processor with a TLB to deal with the address map. The same reasoning applies to any other address-range modifiers that might be defined in the future.

For obvious reasons, burst operations must not under- or over-run the boundaries of the slave which manages the addresses in the operation. Slaves must therefore not declare support for bursts larger than their minimum required address alignment (which we recommend be at least 4kB). Masters, on the other hand, must not generate operations larger than slaves support. However, one might have intermediate TileLink adapters which fragment operations into smaller operations that fit within the target devices. How this information is made available to masters is out-of-scope for this document, although a local address map scheme may again be used.

For the purposes of optimizing throughput, it is also helpful to track which address ranges respond to independent requests in FIFO order. Generally, TileLink responses are completely out-of-order. However, if one knows that a given address range responds in FIFO order, it becomes possible to statelessly transform TL-UH into TL-UL. For these reasons, we recommend that the address map also include an optional FIFO domain. All address ranges which share a common FIFO domain identifier are known to mutually respond in the order of the requests they receive.

Future versions of this specification may define further requirements on the behavior of operations targeting address ranges with certain properties.

6.4. Source and Sink Identifiers

Table 14. Summary of TileLink routing fields

Channel	Dest.	Sequence	Routed By	Provides	For Use As
A	slave	request	`a_address`	`a_source`	
B	master	request	`b_source`	`b_address`	`c_address`
C	slave	response	`c_address`	.	
C	slave	request	`c_address`	`c_source`	`d_source`
D	master	response	`d_source`	.	
D	master	request	`d_source`	`d_sink`	`e_sink`
E	slave	response	`e_source`	.	

Not all routing in TileLink is performed by address. In particular, response messages must be returned to the correct requestor. To make this possible, TileLink channels include one or more link-local transaction identifier fields. These fields are used in combination with the `address` field both to route messages and ensure that every unanswered message can be uniquely identified with a specific ongoing operation. Table 14 provides a summary of the fields used for routing request and response messages on each channel.

At least one signal in every type of request message *must be duplicated* into its corresponding response message. These signals are identified in the **Provides** column in Table 14. For example, if a `Get` request had `a_source` = 4, then the response must have `d_source` = 4 as well. The paired response message will then be routed based on the corresponding signal, shown in column

For Use As. Other signals may also be required to be copied across individual message pairs, as identified below and in the following chapters.

In addition to being used for routing responses, transaction identifiers help to uniquely associate each message with an ongoing operation. Identifiers carry no inherent semantic meaning. Therefore, they can be used by agents to tag a message so as to recognize the message's response, which can be useful when writing stateless forwarding agents, as well as non-blocking masters and slaves. Identifiers are also useful for creating monitors of network behavior.

In order to enable agents to put an upper-bound on the amount of state needed to track ongoing operations, we impose per-link and per-channel uniqueness constraints on identifiers for unaswered request messages. An identifier is said to be outstanding if a request using the identifier has not yet been answered, including the cycle on which the the response is received by the original sender of the request. Each outstanding request identifier of a channel in a particular link must be unique, as defined below.

Channels A and C route their requests based solely on `address`, but both provide `source` signals for use by their responses on Channel D. Because their Channel D responses can be differentiated based on `d_opcode` as well as `d_source`, we allow them to create their `source` identifiers from separate namespaces. In other words, an unanswered Channel A and unanswered Channel C request can each use the same value for `a_source` and `c_source`, but that value cannot be reused within each channel while the request on that channel is unanswered.

Because Channel C responses to Channel B requests are routed to a single slave and uniquely identified to an ongoing operation based on `c_address`, we can further relax the uniqueness restriction on Channel B requests, requiring only that the combination of `b_source` and `b_address` be unique. This relaxation is necessary in order to simultaneously probe multiple masters on the same address, while also probing the same master on multiple addresses. Channel C responses may use any `b_source` associated with the sender; this signal is ignored for those message types.

Channel D must provide unique `d_sink` transaction identifiers for unanswered `Grant` requests. Channel D responses may use any `d_sink` associated with the sender; this signal is ignored for those message types.

The range of possible identifiers is local to a particular TileLink link. Thus, the width of the `source` or `sink` signal in channels can vary wildly between links. A crossbar, for example, might be connected to two masters, M and N. Master M might declare that it uses sources 0-2 while master N uses sources 0-1. The crossbar has two different links to these two masters, so the link-local source identifiers are unrelated. In order for the crossbar to route messages from these masters to slaves, the crossbar must somehow combine the source identifiers into a common namespace for the messages it sends to slaves. One method might be to leave the M sources as 0-2 and remap the N sources to 4-5. Then the crossbar would be able to determine which responses go to which master. The mapping performed by an agent on transaction identifiers is completely implementation defined. Note, for example, that our example crossbar choose to leave source 3 unused in order to optimize its decoding logic. The width of `source` is common to all channels within a particular link.

6.5. Operation Ordering

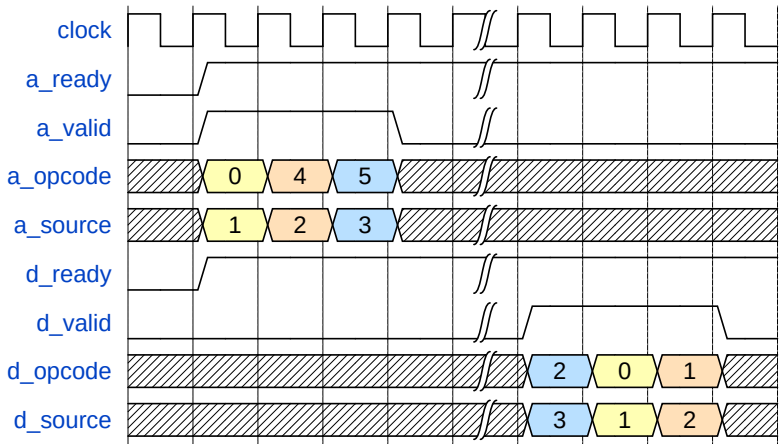

Figure 17. Operations do not necessarily receive responses in order.

Within a TileLink network, there may be multiple outstanding operations at any given time. These operations may be completed in any order. To make it possible for masters to execute one operation after another, TileLink requires that slaves only send a response message once the effect of the operation is completed. Therefore, if a processor needs to ensure that two writes, X and Y, become visible to all other agents in that order, the processor should send a `PutFullData` for X, wait for the `AccessAck` response, and only then send the `PutFullData` for Y.

TileLink slaves, including caches, need not actually write-back the Put operations before they are acknowledged. The only restriction is that the entire TileLink network cannot observe the old state once the acknowledgement has been sent. This implies that all current cached copies of the data are up-to-date before the acknowledgement is sent. For example, in the case of a `Put` operation, an outer-level cache must either `Probe` inner caches with current copies or forward the `PutFullData` message to those inner caches, and collect the appropriate response message(s) before acknowledging the original request.

Response-issuing agents are responsible for ensuring that there is a valid serialization of the operations they received. For example, suppose an agent receives two Puts, X and Y, which it has not yet acknowledged. It must select some ordering, say X before Y. If this ordering is selected, it must ensure that there are only three visible states: the state before X and Y, the state after X and before Y, and the state after both X and Y. The agent need not issue responses to X and Y in this order. However, once the agent has issued a response, say for Y, if it receives a new operation Z, then Z must be ordered after Y.

These rules ensure that the globally visible total order of operations at each agent is consistent with the Ack-induced partial orderings of the masters. A processor can implement fence instructions by waiting for outstanding Acks to return before executing new operations on the other side of the fence. This capability makes it possible multiple processors to safely synchronize their operations via the TileLink shared memory system.

7. TileLink Uncached Lightweight (TL-UL)

TileLink Uncached Lightweight (TL-UL) is the minimal TileLink conformance level. It is intended to be used to save area in low-performance peripherals. There are two types of operations available to agents in TL-UL. Both are memory access operations:

get operation

Read some amount of data from backing memory.

put operation

Write some amount of data to backing memory. The write can have a partial write mask at byte granularity.

These operations are all completed using the two-stage request/response transaction structure laid out in Section 4.2. However, in TL-UL, every message fits within a single beat; there are no bursts. In total there are three request message types and two response message types related to memory access operations in TL-UL. Table 15 enumerates these messages. The listed responses are the only ones possible.

Table 15. Summary of TL-UL messages.

Message	Opcode	Operation	A	D	Response
`Get`	4	Get	y	.	`AccessAckData`
`AccessAckData`	1	Get	.	y	
`PutFullData`	0	Put	y	.	`AccessAck`
`PutPartialData`	1	Put	y	.	`AccessAck`
`AccessAck`	0	Put	.	y	

7.1. Flows and Waves

The figures in this section provide waveforms and message sequence charts for the TL-UH operations. Figure 18 shows a waveform containing both Get and Put operations between a single pair of agents.

Message sequence charts display the ordering and dependencies of the messages sent between agents and the actions they take in response over time. Time flows from the top of the sequence chart to the bottom. Figure 19 shows the message flow employed by Get operations between a single pair of agents. Figure 20 shows the message flow employed by Put operations between a single pair of agents. Figure 21 shows the message flow employed by either operation through two levels of master-slave agent pairs.

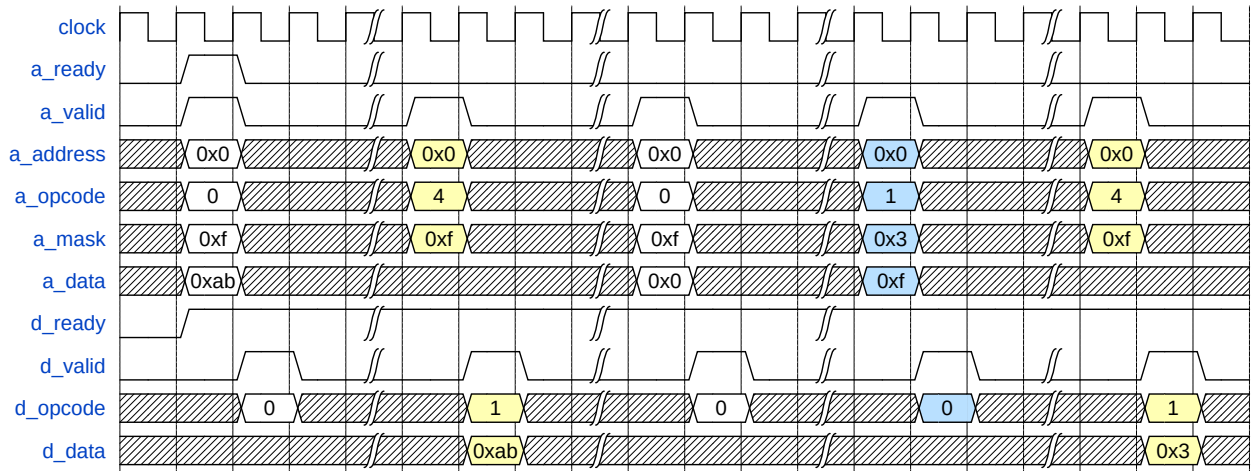

Figure 18. Waveform containing Get and Put operations.

`PutFullData` writes `0xabcd`; `Get` reads `0xabcd`; `PutFullData` writes `0x0000`; `PutPartialData` writes part of `0xffff`; `Get` reads `0x00ff`.

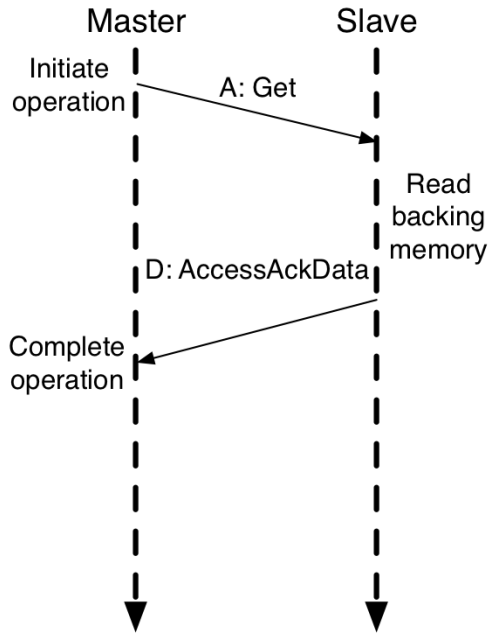

Figure 19. An overview of the Get message flow.

A master sends a `Get` to a slave. Having read the required data, the slave responds to the master an `AcessAckData`.

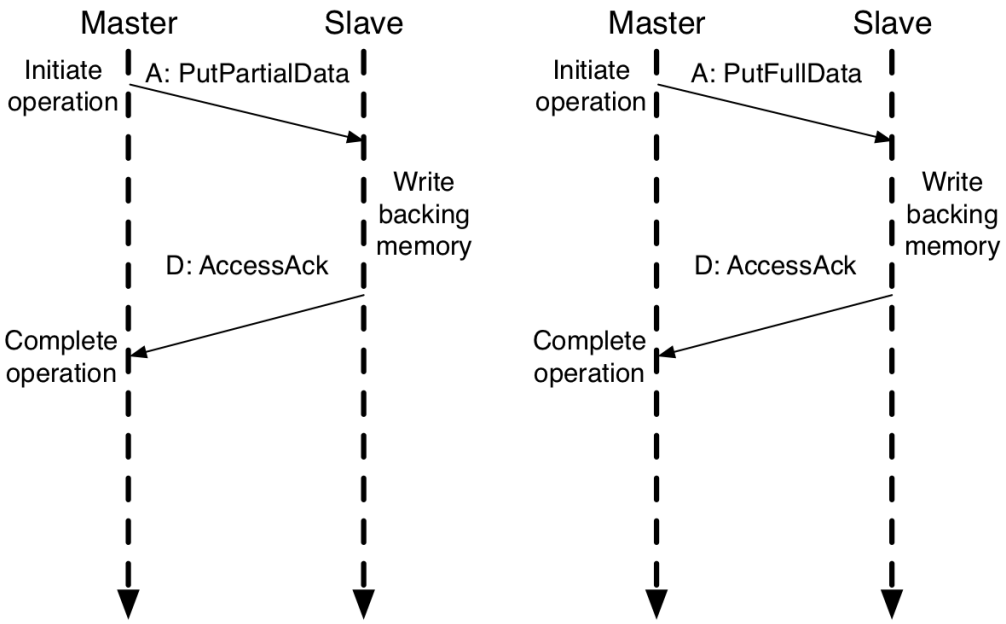

Figure 20. An overview of the `Put` message flows. A master sends

an `PutPartialData` or `PutFullData` to a slave. After writing the included data, the slave responds to the master with an `AccessAck`.

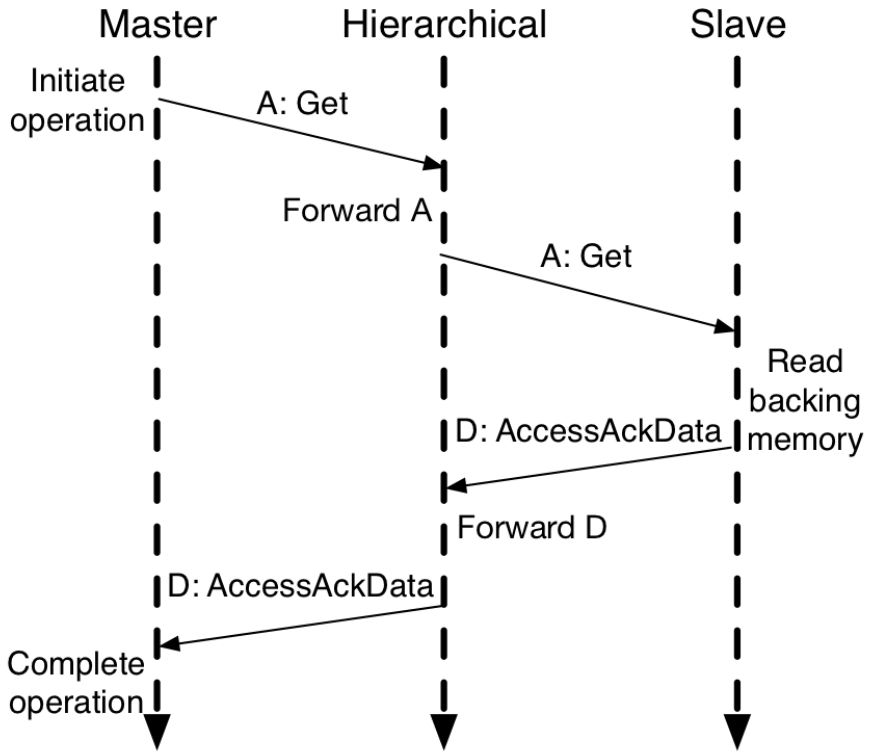

Figure 21. A message flow across multiple hierarchical agents

to perform a memory access that reads a block of data. The Hierarchical Agent forwards the Get to the outer Slave Agent and then also forwards the response AccessAckData to the Master Agent.

7.2. TL-UL Messages

This section defines the encodings used for the signals comprising the five message types included in TL-UL.

7.2.1. Get

A Get message is a request made by an agent that would like to access a particular block of data in order to read it. Table 16 shows the encodings used for the signals of Channel A for this message.

a_opcode must be Get, which is encoded as 4.

a_param is currently reserved for future performance hints and must be 0.

a_size indicates the total amount of data the requesting agent wishes to read, in terms of log2(bytes). a_size represents the size of the resulting AccessAckData response message, not this particular Get message. In TL-UL, a_size cannot be larger than the width of the physical data bus.

a_source is the transaction identifier of the Master Agent issuing this request. It will be copied by the Slave Agent to ensure the response is routed correctly (Section 6.4).

a_address must be aligned to a_size.

a_mask selects the byte lanes to read (Section 4.5). a_size, a_address and a_mask are required to correspond with one another. Get must have a contiguous mask that is naturally aligned.

a_data is ignored and may take any value. a_corrupt is reserved and must be 0.

Table 16. Fields of Get messages.

Channel A	Type	Width	Encoding
a_opcode	C	3	Must be Get (4).
a_param	C	3	Reserved; must be 0.
a_size	C	z	2^n bytes will be read by the slave and returned.
a_source	C	o	The master source identifier issuing this request.
a_address	C	a	The target address of the Access, in bytes.
a_mask	D	w	Byte lanes to be read from.
a_corrupt	D	1	Reserved; must be 0.
a_data	D	$8w$	Ignored; can be any value.

7.2.2. PutFullData

A PutFullData message is a request made by an agent that would like to access a particular block of data in order to write it. The motivation for including a special opcode identifying a full write mask will

be explained in Section 8. Table 16 shows the encodings used for the signals of Channel A for this message.

a_opcode must be PutFullData, which is encoded as 0.

a_param is currently reserved for future performance hints and must be 0.

a_size indicates the total amount of data the requesting agent wishes to write, in terms of log2(bytes). a_size also represents the size of this request message. In TL-UL, a_size cannot be larger than the width of the physical data bus.

a_source is the transaction identifier of the Master Agent issuing this request. It will be copied by the Slave Agent to ensure the response is routed correctly (Section 6.4).

a_address must be aligned to a_size. The entire contents of a_address to (a_address + 2^{a_size} – 1) will be written.

a_mask selects the byte lanes to write (Section 4.5). One HIGH bit of a_mask corresponds to one byte of data written. a_size, a_address and a_mask are required to correspond with one another. PutFullData must have a contiguous mask, and if a_size is greater than or equal the width of the physical data bus then all a_mask must be HIGH.

a_data is the actual data payload to be written. Any byte of a_data that is not masked by a_mask is ignored and can take any value.

Table 17. Fields of PutFullData messages.

Channel A	Type	Width	Encoding
a_opcode	C	3	Must be PutFullData.
a_param	C	3	Reserved; must be 0.
a_size	C	z	2^n bytes will be written by the slave.
a_source	C	o	The master source identifier issuing this request.
a_address	C	a	The target address of the Access, in bytes.
a_mask	D	w	Byte lanes to be written; must be contiguous.
a_corrupt	D	1	Whether this beat of data is corrupt.
a_data	D	$8w$	Data payload to be written.

7.2.3. PutPartialData

A PutPartialData message is a request made by an agent that would like to access a particular block of data in order to write it. PutPartialData can be used to write arbitrary-aligned data at a byte granularity. Table 18 shows the encodings used for the signals of Channel A for this message.

a_opcode must be PutPartialData, which is encoded as 1.

a_param is currently reserved for future performance hints and must be 0.

a_size indicates the range of data the requesting agent will possibly write, in terms of log2(bytes). a_size also represents the size of this request message. In TL-UL, a_size cannot be larger than the

width of the physical data bus.

a_source is the transaction identifier of the Master Agent issuing this request. It will be copied by the Slave Agent to ensure the response is routed correctly (Section 6.4).

a_address must be aligned to a_size. Some subset of the contents of a_address to (a_address + 2^a_size - 1) will be written.

a_mask selects the byte lanes to write (Section 4.5). One HIGH bit of a_mask corresponds to one byte of data written. a_size, a_address and a_mask are required to correspond with one another. However, PutPartialData may write less data than a_size, depending on the contents of a_mask. Any HIGH bits of a_mask must be contained within an aligned region of a_size, but the HIGH bits do not have to be contiguous.

a_data is the actual data payload to be written. Any byte of a_data that is not masked by a_mask is ignored and can take any value. a_corrupt being HIGH indicates that masked data in this beat is corrupt.

Table 18. Fields of PutPartialData messages.

Channel A	Type	Width	Encoding
a_opcode	C	3	Must be PutParialData (1).
a_param	C	3	Reserved; must be 0.
a_size	C	z	Up to 2^n bytes will be written by the slave.
a_source	C	o	The master source identified issuing this request.
a_address	C	a	The target base address of the Access, in bytes.
a_mask	D	w	Byte lanes to be written.
a_corrupt	D	1	Whether this beat of data is corrupt.
a_data	D	$8w$	Data payload to be written.

7.2.4. AccessAck

AccessAck serves as a data-less acknowledgement message to the original requesting agent. Table 19 shows the encodings used for the signals of Channel D for this message.

d_opcode must be AccessAck, which is encoded as 0.

d_param is reserved for use with TL-C opcodes and must be assigned 0.

d_size contains the size of the data that was accessed, though this particular message contains no data itself. In a request/response message pair, d_size and a_size must always correspond. In TL-UL, d_size cannot be larger than the width of the physical data bus.

d_source was saved from a_source in the request and is now used to route this response to the correct destination (Section 6.4).

d_sink is ignored and can be assigned any value.

d_data is ignored and can be assigned any value.

d_corrupt is reserved and must be 0.

d_denied indicates that the slave did not process the memory access.

Table 19. Fields of AccessAck messages.

Channel D	Type	Width	Encoding
d_opcode	C	3	Must be AccessAck (0).
d_param	C	2	Reserved; must be 0.
d_size	C	*z*	2^n bytes were accessed by the slave.
d_source	C	*o*	The master source identifier receiving this response.
d_sink	C	*i*	Ignored, can be any value.
d_denied	C	1	The slave was unable to service the request.
d_corrupt	D	1	Reserved, must be 0.
d_data	D	*8w*	Ignored; can be any value.

7.2.5. AccessAckData

AccessAckData serves as an acknowledgement message including data to the original requesting agent. Table 20 shows the encodings used for the signals of Channel D for this message.

d_opcode must be AccessAckData, which is encoded as 1.

d_param is reserved for use with TL-C opcodes and must be 0.

d_size contains the size of the data that was accessed, which corresponds to the size of the data being included in this particular message. In a request/response message pair, d_size and a_size must always correspond. In TL-UL, d_size cannot be larger than the width of the physical data bus.

d_source was saved from a_source in the request and is now used to route this response to the correct destination (Section 6.4).

d_sink is ignored and can be assigned any value.

d_data contains the data that was accessed by the operation.

d_corrupt being HIGH indicates that masked data in this beat is corrupt.

d_denied indicates that the slave did not process the memory access. If d_denied is HIGH then d_corrupt must also be high.

Table 20. Fields of AccessAckData messages.

Channel D	Type	Width	Encoding
d_opcode	C	3	Must be AccessAckData (1).
d_param	C	2	Reserved; must be 0.
d_size	C	*z*	2^n bytes were accessed by the slave.
d_source	C	*o*	The master source identifier receiving this response.
d_sink	C	*i*	Ignored; can be any value.
d_denied	C	1	The slave was unable to service the request.
d_corrupt	D	1	Whether this beat of data is corrupt.
d_data	D	*8w*	The data payload.

8. TileLink Uncached Heavyweight (TL-UH)

TileLink Uncached Heavyweight (TL-UH) is intended for use beyond the outermost cache layer, where no permission transfer operations are required. It builds on TL-UL by providing additional operations:

atomic operation

> Atomically read and return the extant data value while simultaneously writing a new value that is the result of some logical or arithmetic operation.

hint operation

> Provide an optional hint related to some performance optimization.

burst messages

> Allow messages with data larger than the width of the physical data bus to be transmitted as bursts occurring over multiple cycles. Applies to various data-containing messages within the **Get**, **Put** and **Atomic** operations.

Atomic operations allow agents to access a particular block of data in order to perform a memory operation that atomically reads and returns the current data value while simultaneously writing a new value that is the result of some logical or arithmetic operation. Each operation takes two operands; one is the data carried with the Atomic message, and the second is the extant data value at the target address. This operation returns a copy of the original data to the requestor. Identifying the logical vs arithmetic operations is useful because the ALU requirements significantly differ for implementing the two sub-classes of operation.

Hint operations serve as a mechanism for implementing optional performance optimizations. While they may cause agents to act to change the permissions available on certain data blocks, they never modify the value of data. The information provided by a Hint may always be safely ignored by any Slave Agent that receives it, though the recipient must still send an acknowledgement message.

Burst messages allow operations to target larger address ranges, and specifically enable messages with data sizes bigger than the width of the physical data bus. Any of the various messages within Get, Put and Atomic operations that contain `*Data` in their name can be a burst. No new message types are added with the burst capability; instead, certain signaling restrictions from Section 7 are removed. See Section 4.1 and Section 4.2 for details on how operations including bursts are serialized and sequenced.

The new operations are also completed using the paired request/response transaction structure laid out in Section 4.2. In total there are three request messages and one response message added by TL-UH to the messages defined for TL-UL. Table 21 enumerates these messages. The listed responses are the only ones possible.

Table 21. Summary of TL-UH messages.

from TL-UL					
Message	**Opcode**	**Operation**	**A**	**D**	**Response Message**
`Get`	4	Get	y	.	`AccessAckData`
`AccessAckData`	1	Get or Atomic	.	y	
`PutFullData`	0	Put	y	.	`AccessAck`
`PutPartialData`	1	Put	y	.	`AccessAck`
`AccessAck`	0	Put	.	y	
added in **TL-UH**					
`ArithmeticData`	2	Atomic	y	.	`AccessAckData`
`LogicalData`	3	Atomic	y	.	`AccessAckData`
`Intent`	5	Intent	y	.	`HintAck`
`HintAck`	2	Intent	.	y	

8.1. Flows and Waves

The figures in this section provide waveforms and message sequence charts for each of the additional TL-UH operations. Figure 22 shows a waveform containing both Atomic and Hint operations between a single pair of agents. Figure 22 shows the message flow employed by Atomic operations between a single pair of agents. Figure 24 shows the message flow employed by Hint operations between a single pair of agents.

For waveforms of burst messages please refer to Section 4.1 and Section 4.2

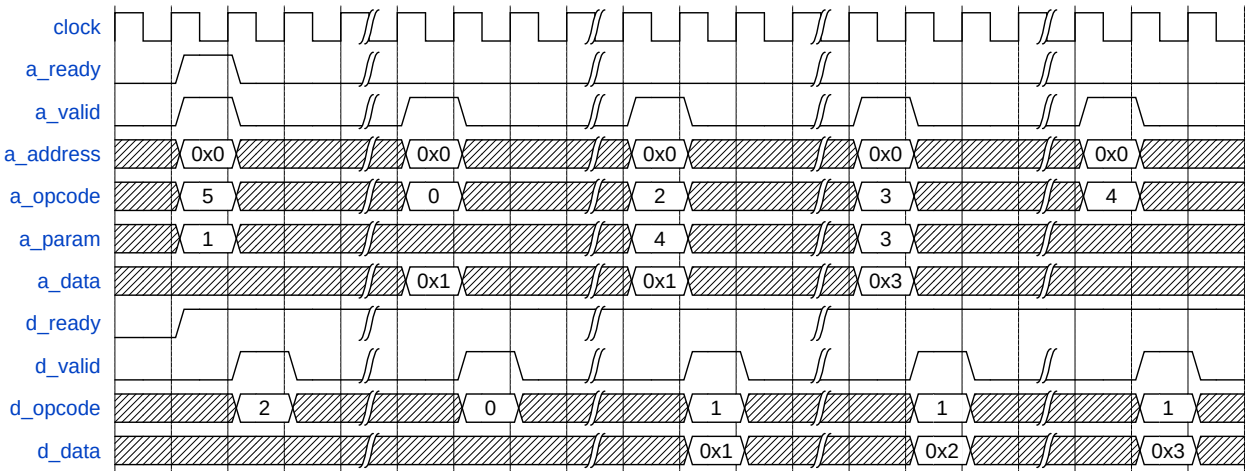

Figure 22. Waveform containing Atomic and Hint operations.

It shows Prefetch with intent to write; Put storing 0x1; Atomic add of 0x1 returning 0x1; Atomic swap of 0x3 returning 0x2; Get loading 0x3.

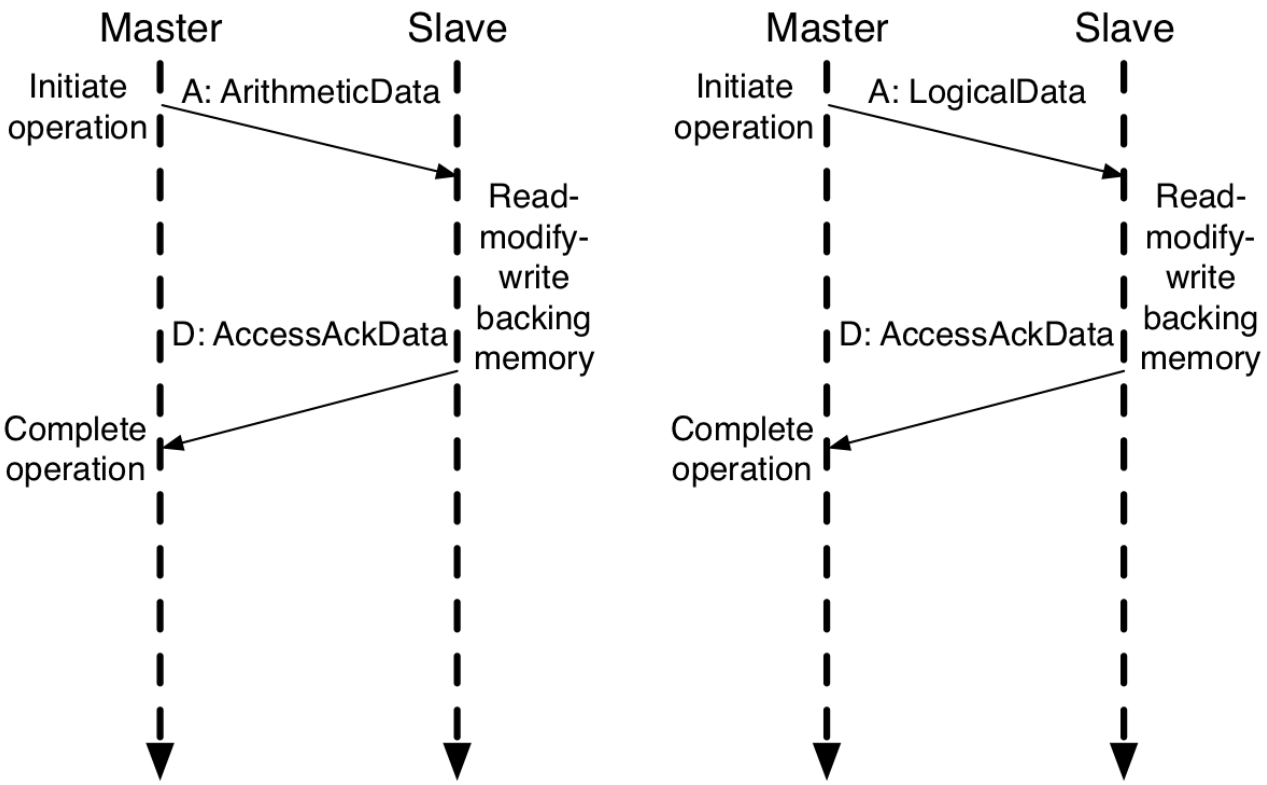

Figure 23. Message flow to perform an atomic memory access operation.

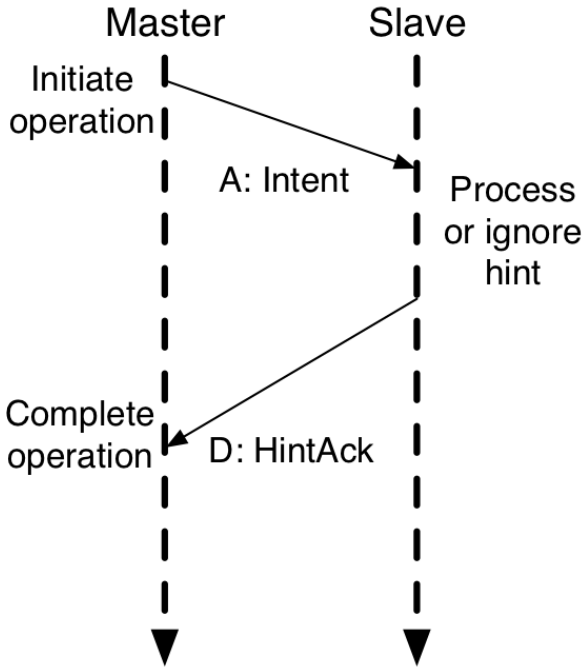

Figure 24. Message flow to perform a hint operation.

8.2. TL-UH Messages

This section defines the encodings used for the signals comprising the four message types included in TL-UH: `ArithmeticData`, `LogicalData`, `Intent`, `HintAck`.

8.2.1. ArithmeticData

An `ArithmeticData` message is a request made by an agent that would like to access a particular block of data in order to read-modify-write it by applying an arithmetic operation. Table 22 shows the encodings used for the signals of Channel A for this message.

`a_opcode` must be `ArithmeticData`, which is encoded as 2.

`a_param` specifies the specific atomic arithmetic operation to perform. The set of supported arithmetic operations is listed in Table 23. It consists of `MIN`, `MAX`, `MINU`, `MAXU` and `ADD`, representing signed and unsigned integer maximum and minimum functions, as well as integer addition.

`a_size` is the operand size, in terms of log2(bytes). It reflects both the size of this request's data as well as the size of the `AccessAckData` response.

`a_source` is the transaction identifier of the Master Agent issuing this request. It will be copied by the Slave Agent to ensure the response is routed correctly. (Section 6.4)

`a_address` must be aligned to `a_size`.

`a_mask` selects the byte lanes to read-modify-write (Section 4.5). One HIGH bit of `a_mask` corresponds to one byte of data used in the atomic operation. `a_size`, `a_address` and `a_mask` are required to correspond with one another. The HIGH bits of `a_mask` must also be naturally aligned and contiguous within that alignment.

`a_data` contains one of the arithmetic operands (the other is found at the target address). Any byte of `a_data` that is not masked by `a_mask` is ignored and can take any value. `a_corrupt` being HIGH indicates that masked data in this beat is corrupt.

Table 22. Fields of `ArithmeticData` messages.

Channel A	Type	Width	Encoding
`a_opcode`	C	3	Must be `ArithmeticData` (2).
`a_param`	C	3	See Table 23.
`a_size`	C	z	2^n bytes will be read and written by the slave.
`a_source`	C	o	The master source identifier issuing this request.
`a_address`	C	a	The target address of the Access, in bytes.
`a_mask`	D	w	Byte lanes to be read and written.
`a_corrupt`	D	1	Whether this beat of data is corrupt.
`a_data`	D	$8w$	Data payload to be used as operand.

Table 23. ArithmeticData param field.

Name	Param	Effect
MIN	0	Write the signed minimum of the two operands, and return the old value.
MAX	1	Write the signed maximum of the two operands, and return the old value.
MINU	2	Write the unsigned minimum of the two operands, and return the old value.
MAXU	3	Write the unsigned maximum of the two operands, and return the old value.
ADD	4	Write the sum of the two operands, and return the old value.

8.2.2. LogicalData

A LogicalData message is a request made by an agent that would like to access a particular block of data in order to read-modify-write it by applying a bitwise logical operation. Table 24 shows the encodings used for the signals of Channel A for this message.

a_opcode must be LogicalData, which is encoded as 3.

a_param specifies the specific atomic bitwise logical operation to perform. The set of supported logical operations is listed in Table 25. It consists of { XOR, OR, AND, SWAP }, representing bitwise logical xor, or, and, as well as a simple swap of the operands.

a_size is the operand size, in terms of log2(bytes). It reflects both the size of this request's data as well as the size of the AccessAckData response.

a_source is the transaction identifier of the Master Agent issuing this request. It will be copied by the Slave Agent to ensure the response is routed correctly (Section 6.4).

a_address must be aligned to a_size.

a_mask selects the byte lanes to read-modify-write (Section 4.5). One HIGH bit of a_mask corresponds to one byte of data used in the atomic operation. a_size, a_address and a_mask are required to correspond with one another. The HIGH bits of a_mask must also be naturally aligned and contiguous within that alignment.

a_data contains one of the logical operands (the other is found at the target address). Any byte of a_data that is not masked by a_mask is ignored and can take any value. a_corrupt being HIGH indicates that masked data in this beat is corrupt.

Table 24. Fields of LogicalData messages.

Channel A	Type	Width	Encoding
a_opcode	C	3	Must be LogicalData (3).
a_param	C	3	See Table 25.
a_size	C	z	2^n bytes will be read and written by slave.
a_source	C	o	The master source identifier issuing this request.
a_address	C	a	The target address of the Access, in bytes.
a_mask	D	w	Byte lanes to be read and written.
a_corrupt	D	1	Whether this beat of data is corrupt.
a_data	D	$8w$	Data payload to be written.

Table 25. LogicalData param field.

Name	Param	Effect
XOR	0	Bitwise logical xor the two operands, write the result, and return the old value.
OR	1	Bitwise logical or the two operands, write the result, and return the old value.
AND	2	Bitwise logical and the two operands, write the result, and return the old value.
SWAP	3	Swap the two operands and return the old value.

8.2.3. Intent

A Intent message is a request made by an agent that would like to signal its future intention to access a particular block of data. Table 26 shows the encodings used for the signals of Channel A for this message.

a_opcode must be Intent, which is encoded as 5.

a_param specifies the specific intention being conveyed by this Hint operation. Note that its intended effect applies to the slave interface and possibly agents further out in the hierarchy. The set of supported intentions is listed in Table 27. It consists of { PrefetchRead, PrefetchWrite }, representing prefetch-data-with-intent-to-read and prefetch-data-with-intent-to-write.

a_size is the size of the memory to which this intention applies.

a_source is the transaction identifier of the Master Agent issuing this request. It will be copied by the Slave Agent to ensure the response is routed correctly (Section 6.4).

a_address must be aligned to a_size.

a_mask indicates the bytes to which the intention applies (Section 4.5). a_size, a_address and a_mask are required to correspond with one another.

a_data is ignored and can take any value. a_corrupt is reserved and must be 0.

Table 26. Fields of `Intent` messages.

A Channel	Type	Width	Encoding
`a_opcode`	C	3	Must be Intent (5).
`a_param`	C	3	Intention encoding; See table Table 27.
`a_size`	C	z	2^n bytes to which this intention applies.
`a_source`	C	o	The master source identifier issuing this request.
`a_address`	C	a	The target address of the Hint, in bytes.
`a_mask`	D	w	Byte lanes to which the Hint applies.
`a_corrupt`	D	1	Reserved; must be 0.
`a_data`	D	$8w$	Ignored; can be any value.

Table 27. `Intent` messages' `param` field.

Name	Param	Effect
`PrefetchRead`	0	Issuing agent intends to read target data.
`PrefetchWrite`	1	Issuing agent intends to write target data.

8.2.4. HintAck

`HintAck` serves as an acknowledgement message for a hint operation. Table 28 shows the encodings used for the signals of Channel D for this message.

`d_opcode` must be `HintAck`, which is encoded as 2.

`d_param` is reserved and must be assigned 0.

`d_size` contains the size of the data that was hinted about, though this particular message contains no data itself.

`d_source` was saved from `a_source` in the request and is now used to route this response to the correct destination (Section 6.4)

`d_sink` is ignored and can be assigned any value.

`d_denied` indicates that the slave did not process the hint.

`d_data` is ignored and can be assigned any value. `d_corrupt` is reserved and must be 0.

Table 28. Fields of HintAck messages.

D Channel	Type	Width	Encoding
d_opcode	C	3	Must be HintAck (2).
d_param	C	2	Reserved; must be 0.
d_size	C	*z*	2^n bytes were hinted about.
d_source	C	*o*	The master source identifier receiving this response.
d_sink	C	*i*	Ignored; can be any value.
d_denied	C	1	The slave was unable to service the request.
d_corrupt	D	1	Reserved; must be 0.
d_data	D	*8w*	Ignored, can be any value.

8.3. Burst Messages

Burst messages can contain data that is larger than the width of the physical data bus. The subset of data that can be sent over a link in a single cycle is called a beat. Burst messages can be any of the various messages within Get, Put and Atomic operations that contain *Data in their name.

See Section 4.1 for details on how a burst message is serialized. Three of the types of channel signals delineated in Table 3 are distinguished by whether they can be toggled between beats of a burst. The Data type signals (i.e., *_data, *_mask) are allowed to toggle between each beat, including situations where a particular beat was not accepted by the recipient. The Control type signals (i.e., *_address, *_size, *_param) must be held constant for the entire burst. See Section 4.2 for details on how burst requests and responses comprising an operation can be ordered with respect to one another.

The PutFullData opcode is included in the protocol because it is useful to agents that can make performance optimizations in the presence of full write masks. If the PutFullData message is a burst, such optimizing agents do not have to first collect the mask from every beat in order to determine whether the mask of the entire message is full.

9. TileLink Cached (TL-C)

TileLink Cached (TL-C) completes TileLink by affording master agents the capability to cache copies of blocks of shared data. These local copies must then be kept coherent according to an implementation-defined coherence policy. The TL-C standard coherence *protocol* defined in this chapter dictates *what* memory access operations are allowed to be performed on which cached copies of the data, and *what* messages are available to transfer copies of data blocks. The overlaid, implementation-defined coherence *policy* dictates *how* copies and permissions are propagated through a specific TileLink agent network in response to received memory access operations. Description of specific coherence policies is beyond the scope of this document. In total, TL-C adds to the TileLink protocol specification: three new operations, three new channels, a new five-step message sequence template, and ten new message types.

The new operations are **transfer** operations that create or remove cached copies of data blocks. Transfer operations never modify the value of data blocks, but rather transfer the read/write permissions available on copies of them. Transfer operations interoperate seamlessly with the previously-defined TL-UL and TL-UH memory access operations, in that they are serialized with respect to one another. Because each transfer operation logically either happens before or happens after each memory access operation, and all agents agree on this ordering, the coherence invariant is preserved across the TileLink network.

As a memory access operation proceeds through the TileLink network, an interstitial cache may nest a recursive transfer operation within it. The cache intercedes by first using a transfer operation to obtain sufficient permissions on the block, then servicing the memory access using its coherent local copy.

Cacheability is a property of the address, and TileLink implementations must prevent copies of uncacheable addresses from being created (Section 6.3). Conversely, the memory access operations previously defined in TL-UL and TL-UH may be used by masters to access a cacheable address without caching it themselves. Certain masters may choose cache a particular data block, while other masters at the same level of the memory hierarchy may choose not to.

The next section outlines the fundamental operations, messages, and permissions available for use by designers in defining particular, implementation-dependent coherency policies. This specification does not mandate the use of any one particular policy, but instead defines a protocol substrate on top of which policies can be built.

9.1. Implementing Cache Coherence Using TileLink

All TileLink-based coherence policies are comprised of protocol operations that transfer permissions to read and write copies of data blocks. Memory access operations require the correct permissions to have been acquired by an agent before the agent can apply the access operation to the cached copy. When an agent wants to process an access operation locally, it must first use transfer operations to obtain the necessary permissions. Transfer operations create or remove copies across the network, and thereby modify the permissions that each copy offers.

The fundamental permissions it is possible for a particular agent's copy of a block to have are **None**, **Read**, or **Read+Write**. The permissions available on a particular cached copy depend on the current

presence of copies in the cache hierarchy, as described below.

For any given address, there is exactly one path between any given master and the slave that owns that address. When all such paths are overlaid on the TileLink network DAG, they form a tree with a single slave at the root. For each address, this tree contains the paths along which all operations targeting that address execute. If we elide all agents that cannot cache data, we are left with a tree of caching agents, describing all the locations at which a particular address's data could possibly be cached.

At any given moment in logical time, some subset of those agents actually contain copies of cached data. These agents form a *Coherence Tree*. The inclusive TileLink coherence protocol requires the tree to grow and shrink in response to memory access operations. Every node in the graph falls into one of four categories describing its position on the tree:

Nothing

A node that does not currently cache a copy of the data. Has neither read nor write permissions.

Trunk

A node with a cached copy that is on the path between the Tip and the Root. Has neither read nor write permissions on the copy. The copy may be out of date with respect to writes occurring at the Tip.

Tip (with no Branches)

A node with a cached copy that is serving as the point of memory access serialization. Has Read/Write permissions on its copy, which may contain dirty data.

Tip (with Branches)

A node with a cached copy that is serve as the point of write serialization. Has read and writer permissions on its copy, which may contain dirty data from past writes.

Branch

A node with a cached copy that is above the Tip. Has read-only permissions on its copy.

Figure 25 shows several coherence trees overlaid on a single TileLink network. In A, the root node of the tree has the only copy, which makes it both the root and the tip of the tree. In B, a master has acquired write+read permissions by growing the trunk until it is at the tip. In C, another master has acquired read permissions by growing a branch, meaning that the previous tip is now also a read-only branch and the common parent node is the trunk tip. In D, another master has grown a branch, further moving the tip back towards the root, and the original requestor has voluntarily pruned its branch.

Table 29 describes which access operations can be performed on a node in which state, which is defined relative to its position in the tree. Additional, policy-defined states can be based off of these fundamental states.

Table 29. Relationships between permissions and access operations.

Permissions	Supported Accesses
None	None
Branch	Get
Trunk	Get
Tip (w/ Branches)	Get
Tip (no Branches)	Get, PutPartial, PutFull, Logical, Arithmetic

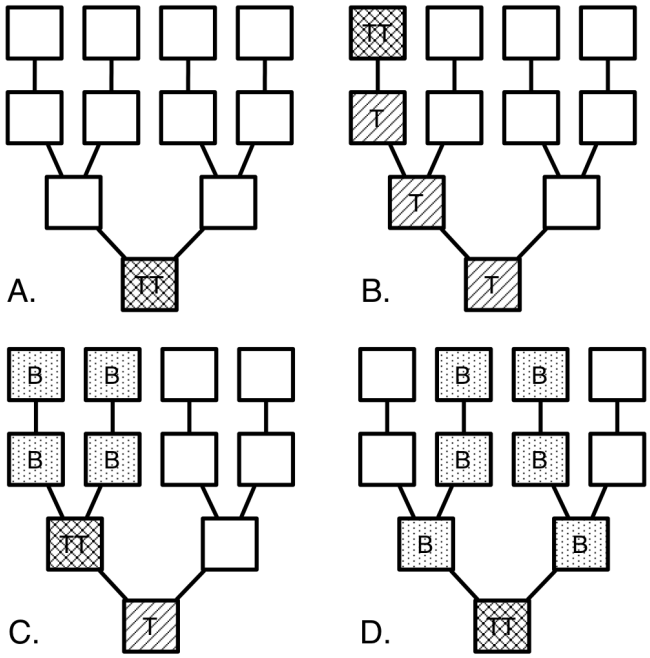

Figure 25. Different possible coherence trees overlaid on a single TileLink network graph.

TT is Trunk Tip, *T* is trunk, and *B* is branch.

In Figure 25, several different possible coherence trees are overlaid on a single TileLink network graph. *TT* is Trunk Tip, *T* is trunk, and *B* is branch. The four example trees are: * A: The root has write+read permission on the only copy. * B: A single master has write+read permissions on the trunk tip. * C: Multiple masters have read permissions on branches. * D: Multiple masters have read permissions on branches, and some branches have been pruned.

9.1.1. Operations

The three new operations are termed **transfer** operations (Section 6) because they transfer a copy of a block of data to a new location in the memory hierarchy:

Acquire

Creates a new copy of a block (or particular permissions on it) in the requesting master.

Release

Relinquishes a copy of the block (or particular permissions on it) back to the slave from the requesting master.

Probe

Forcibly removes of a copy of the block (or particular permissions on it) from a master to the requesting slave.

Acquire operations grow the tree, either by extending the trunk or by adding a new branch from an existing branch or the tip. In order to do so, the old trunk or branches may have to be pruned with recursive **Probe** operations before the new branch can be grown. **Release** operations prune the tree by voluntarily shrinking it, typically in response to cache capacity conflicts.

9.1.2. Channels

To provide support for transfer operations, TL-C adds three new channels to the two extant channels that were required to perform memory access operations. The A and D channels are also repurposed to send additional messages to effect transfer operations. The five channels used by transfer operations are:

Channel A

A master initiates acquiring permission to read or write a copy of a cache block.

Channel B

A slave queries or modifies a master's permissions on a cached data block, or forwards a memory access to a master.

Channel C

A master acknowledges a Channel B message, potentially releasing permissions on the block along with any dirty data. Also used to voluntarily write back dirtied cache data.

Channel D

A slave provides data or permissions to the original requestor, granting access to the cache block. Also used to acknowledge voluntary writebacks of dirty data.

Channel E

A master provides final acknowledgment of transaction completion, used by the slave for transaction serialization.

9.1.3. TL-C Messages

Across the five channels, TL-C specifies ten messages comprising three operations. Table 30 enumerates these messages. The listed responses are the only ones possible, but note that for certain request message types multiple response message types may be received, depending on whether the respondent returned a copy of the data.

Table 30. Summary of TL-C Permission Transfer Operation Messages.

Message	Opcode	Operation	A	B	C	D	E	Response
AcquireBlock	6	Acquire	y	.	.	.	.	Grant, GrantData
AcquirePerm	7	Acquire	y	.	.	.	.	Grant
Grant	4	Acquire	.	.	.	y	.	GrantAck
GrantData	5	Acquire	.	.	.	y	.	GrantAck
GrantAck	-	Acquire	.	.	.	.	y	
ProbeBlock	6	Probe	.	y	.	.	.	ProbeAck, ProbeAckData
ProbePerm	7	Probe	.	y	.	.	.	ProbeAck
ProbeAckData	5	Probe	.	.	y	.	.	
Release	6	Release	.	.	y	.	.	ReleaseAck
ReleaseData	7	Release	.	.	y	.	.	ReleaseAck
ReleaseAck	6	Release	.	.	.	y	.	

9.1.4. Permissions Transitions

Transfers logically operate on permissions, and so messages that comprise them must specify an intended outcome: an upgrade to more permissions, a downgrade to fewer permissions, or a no-op leaving permissions unchanged. These changes are specified in terms of their effect on the shape of the coherence tree for a particular address. We break the set of possible permission transitions into six subsets; different subsets are available as parameters to certain messages, as defined in the following subsection.

Table 31. Categories of permissions transitions and their encodings.

Category	Contents
Permissions	**N**one, **B**ranch, **T**runk
Cap	toT (0), toB (1), toN (2)
Grow	NtoB (0), NtoT (1), BtoT (2)
Prune	TtoB (0), TtoN (1), BtoN (2)
Report	TtoT (3), BtoB (4), NtoN (5)

Table 31 shows all the permissions transitions any coherence policy based on TileLink could want to express. They are group into four subsets.

Prune

comprises permissions downgrades that shrink the tree, and notes both the previous permissions and the new, lesser permissions.

Grow

comprises permissions upgrades that grow the tree, and notes both the previous permissions and the new, greater permissions.

Report

comprises no-ops where the permissions remain unchanged, but reports what those permissions currently are.

Cap

comprises permissions changes without specifying what the original permissions were, but rather only what they should become.

9.2. Flows and Waves

Transfer operations introduce new transaction flows which can be composed to form complete cache coherence policy transactions. Figure 26 provides an overview of the three new flows. `Acquire` requests always trigger a recursive `Grant` request and `GrantAck` response. Depending on the state of the block's permissions and the coherence policy, an Acquire may also trigger one or more recursive Release or Probe operations.

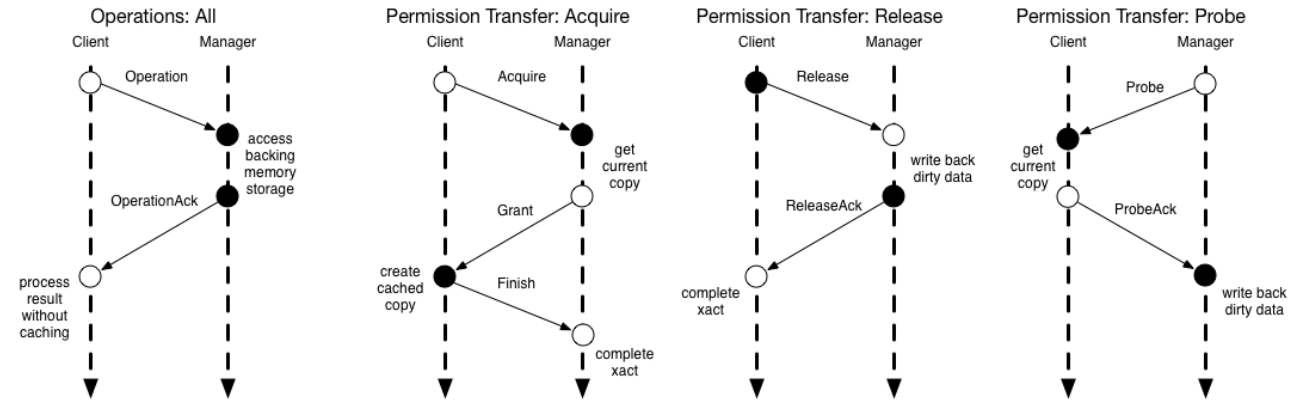

Figure 26. Provides an overview of the transaction flows of TileLink operations.

Movement of the black dot indicates the point of transaction serialization has been affected by the operation.

Figure 27 shows a message flow that illustrates in more detail a transaction that contains all three new operations. In this flow a master reacts to a memory access operation request by acquiring permissions to read or write data in a local copy of the target data block. After this transaction has completed, the master has acquired permissions to either read or write the cache block, as well as a copy of the block's data. Other masters were probed to force them to release their permissions on the block and write back dirty data in their possession. Additionally, the master that issued the `Acquire` also used a `Release` to voluntarily release their permissions on a cache block. Typically, this type of transaction occurs when a cache must evict a block that contains dirty data, in order to replace it with the block being refilled into the cache. After this transaction has completed, the master has lost permissions to read or write the second cache block, as well as its copy of that data. If the slave is capable of tracking which masters have copies of the block using a directory, this metadata has been updated to reflect the change in permissions to both blocks.

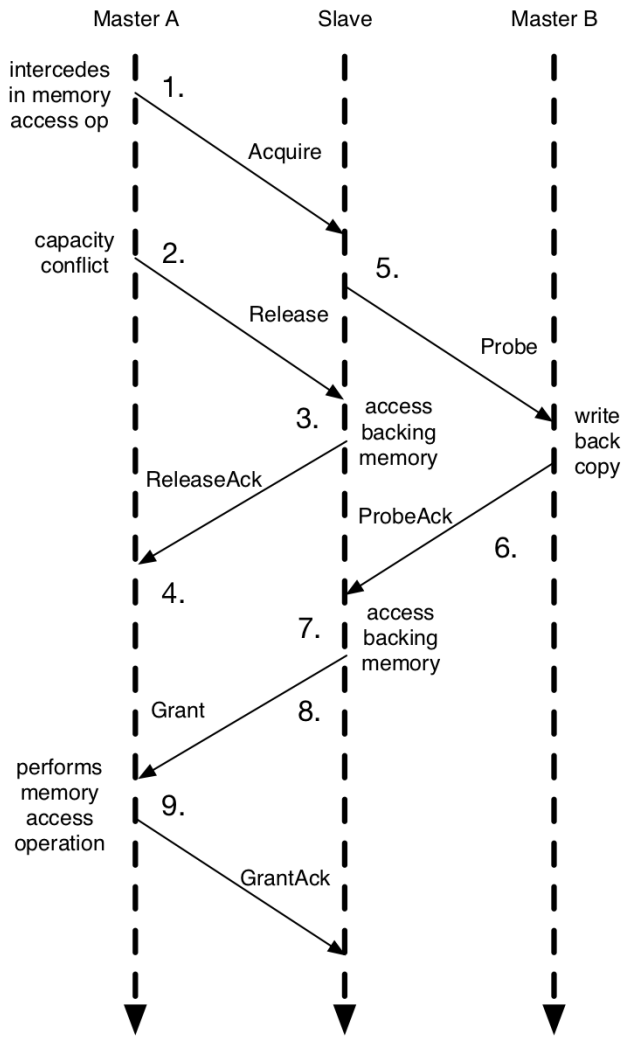

Figure 27. Overview of a transaction flow containing all three transfer operations.

1. A caching master sends an `Acquire` to a slave.

2. To make room for the expected response, the same master sends a `Release`.

3. The slave communicates with backing memory if required.

4. The slave acknowledges completion of the writeback transaction using a `ReleaseAck`.

5. The slave also sends any necessary `Probes` to other masters.

6. The slave waits to receive a `ProbeAck` for every `Probe` that was sent.

7. The slave communicates with backing memory if required.

8. The slave responds to the original requestor with a `Grant`.

9. The original master responds with a `GrantAck` to complete the transaction.

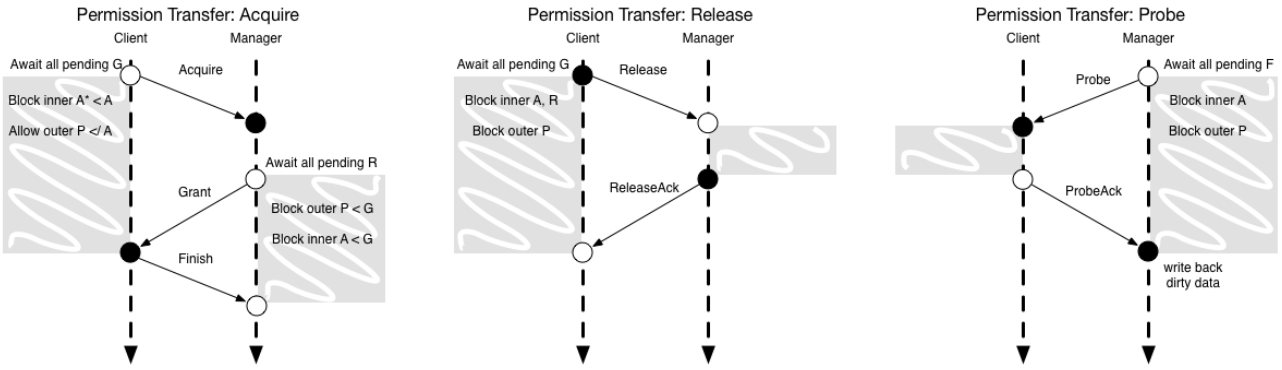

Figure 28. Operation Flows.

While these three flows form the basis of all TileLink transactions involving cache block transfers, there are a number of edge cases that arise when they are overlaid on each other temporally or composed hierarchically. We now discuss how responsibility for managing this concurrency is distributed across master and slave TileLink agents.

TileLink intentionally does not assume that there is point-to-point ordered delivery of messages. In fact, messages from higher priority channels must be able to bypass lower priority messages in the network, even if they are targeting the same agent. The slave serves as a convenient point of synchronization across all the masters connected to it. Since every transaction must be initiated via an Acquire message sent to a slave, the slave can trivially order the transactions. A very safe implementation would be to accept only a single transaction at a time, but the performance implications of doing so are dire, and it turns out we can be much more concurrent while continuing to provide a correct serialization. Imposing some restrictions on agent behavior makes it possible for us to guarantee that a total ordering of transactions can be constructed, despite the distributed nature of the problem. Figure 28 provides an overview of the limits put on concurrency for each operation.

Concurrency limits placed on TileLink agents are most easily understood in terms of issuing or blocking request messages. All request messages generate response messages, and response messages are guaranteed to eventually make forward progress. However, under certain conditions, recursive request messages targeting the same block should not be issued until an outstanding response message is received. We break these cases down by request message type:

`Acquire`

> A master should not issue an `Acquire` if there is a pending `Grant` on the block. Once the `Acquire` is issued the master should not issue further `Acquires` on that block until it receives a `Grant`.

`Grant`

> A slave should not issue a `Grant` if there is a pending `ProbeAck` on the block. Once the `Grant` is issued, the slave should not issue `Probes` on that block until it receives a `GrantAck`.

`Release`

A master should not issue a Release if there is a pending Grant on the block. Once the Release is issued, the master should not issue ProbeAcks, Acquires, or further Releases until it receives a ReleaseAck from the slave acknowledging completion of the writeback.

Probe

A slave should not issue a Probe if there is a pending GrantAck on the block. Once the Probe is issued, the slave should not issue further Probes on that block until it receives a ProbeAck.

We now offer some example flows demonstrating concurrency limits being obeyed in message sequence chart format. Figure 29 lays out a scenario where a Probe request is delayed. Masters must continue to process and respond to Probes even with an outstanding Grant pending in the network. Slaves must include an up-to-date copy of the data in Grants responding to Acquires upgrading permissions, unless they are certain that that master has not been probed since the Acquire was issued. Assuming a slave has blocked on processing a second transaction acquiring the same block, the critical question becomes: When is it safe for a slave to process the pending Acquire? If we were to assume point-to-point ordered delivery of messages to a particular agent, it would be sufficient for the slave merely to have sent the Grant message to the original master source. The slave could process further transactions on the block, and further Probes and Grants to the same master would arrive in order. Since this ordering is not guaranteed, we instead rely on the GrantAck message to allow the slave to serialize the two transactions.

We now turn to a second example of concurrency-limiting responsibility, which is put on the master. If a master has an outstanding Release transaction on a block, it cannot respond to an incoming Probe request on that block with ProbeAcks until it receives a ReleaseAck from the slave acknowledging completion of the writeback. Figure 30 lays out this scenario in message sequence chart form. This limitation serializes the ordering of the voluntary writeback relative to the ongoing Acquire operation that generated the Probes. The slave cannot simply block the voluntary Release transaction until the Acquire transaction completes, because the ProbeAck message in that transaction could be blocked in the network behind the voluntary Release. From the slave agent's perspective, it must handle the situation of receiving a voluntary Release for a block another master is currently attempting to Acquire. The slave must accept the voluntary Release as well as any overtaken ProbeAcks resulting from Probe messages that have already been processed by the master before the Release was sent, and afterwards provide a ReleaseAck and Grant message to each master before their transactions can be considered complete. The voluntary Release's data can be used to respond to the original requestor with a `Grant, but the Grant itself cannot be sent until the expected number of ProbeAcks have been collected by the slave. This scenario is an example of two transaction message flows being merged by the slave agent.

The final concurrency-limiting responsibility put on the Master Agent is to issue multiple Channel A requests for the same block only when the transactions can be differentiated from one another via unique transaction identifiers. For example, a Master Agent cache that has a write miss under a read miss may issue an Acquire asking for write permission before the Grant providing read permissions has arrived. However, it must use a unique transaction ID for the second Acquire even though it is targeting the same address. The Master Agent cannot expect that the Slave Agent will serialize multiple outstanding Acquires in any particular order, and it must send a GrantAck for the first Grant[Data] it receives without waiting to receive the second one.

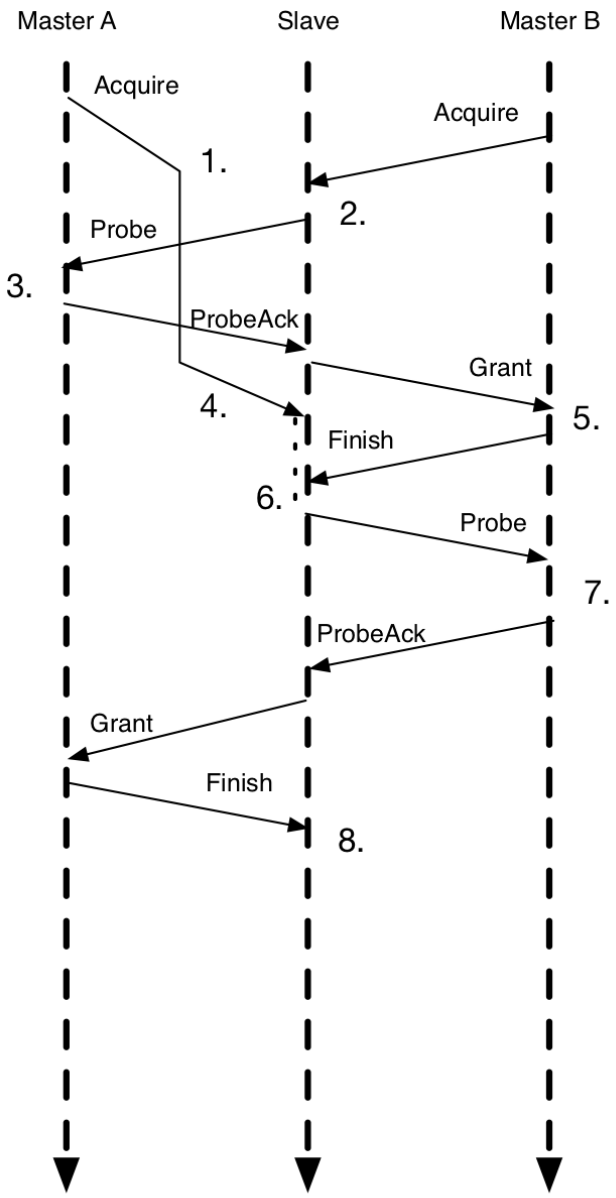

Figure 29. Interleaved but well-ordered message flows.

Figure 29 shows interleaved message flows demonstrating a slave using `GrantAck` to serialize `Grant` and `Probe`.

1. Master A sends an `Acquire` first, but it gets delayed in the network.
2. Master B sends an `Acquire` second, but it arrives at the slave first, and is serialized before A's.
3. The slave sends a `Probe` to Master A, which must process it even though it has pending `Grant`.
4. The slave receives Master A's `ProbeAck` and sends Master B a `Grant`.
5. Master A's `Acquire` arrives at the slave but cannot make forward progress due to the pending `GrantAck`.
6. Once Master B responds with a `GrantAck`, Master A's transaction can proceed as normal.
7. The slave probes Master B, but this probe is serialized relative to the previous `Grant`.

8. The slave must respond to Master A with the correct type of `Grant` (including a copy of the data), given that Master A has been probed since sending its `Acquire`.

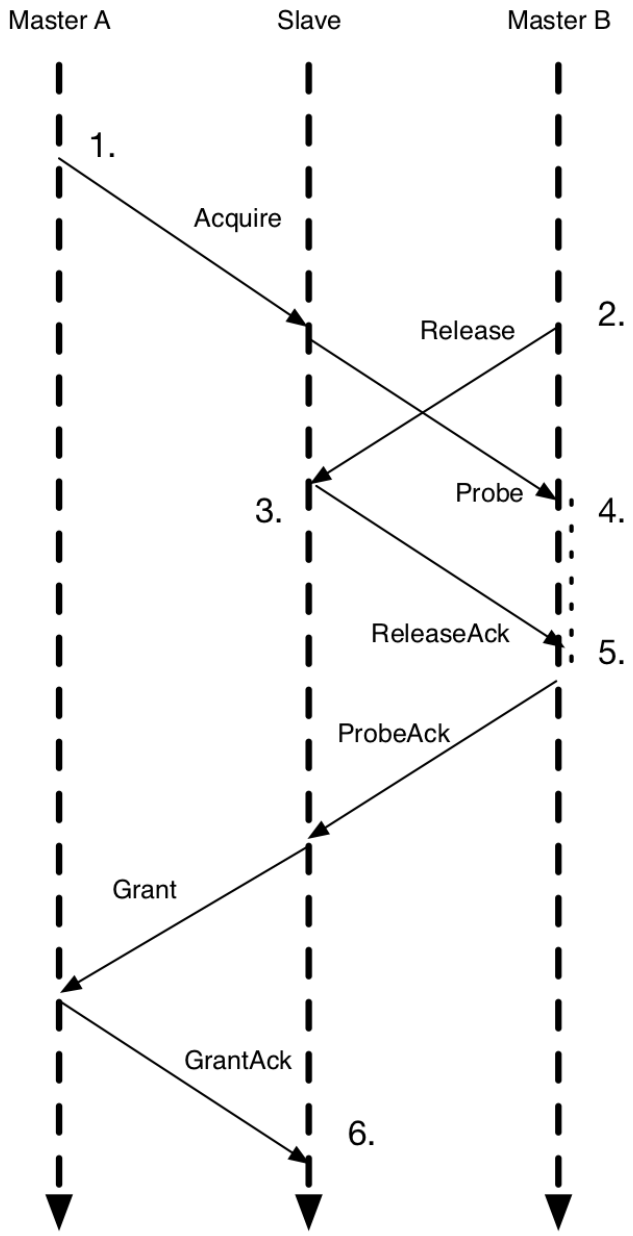

Figure 30. Interleaved and well-ordered voluntary writeback message flows.

Figure 30 presents interleaved message flows demonstrating using `ReleaseAck` to serialize `Release` and `Probe`.
1. Master A sends an `Acquire` to a slave.
2. At the same time, Master B chooses to evict the same block and issues a voluntary `Release`.
3. The slave then sends a `Probe` to Master B.
The slave waits to receive a `ProbeAck` for every `Probe` that was sent, but additionally also accepts the voluntary `Release`.

The slave sends a ReleaseAck that acknowledges receipt of the voluntary writeback.
5. Master B does not respond to the Probe with a ProbeAck until it gets the acknowledgment ReleaseAck.
6. Once Master B responds with a ProbeAck, Master A's transaction can proceed as normal.

9.3. TL-C Messages

Permissions transfers make use of three additional channels with six new messages, a new Chanel A message, and three new Channel D messages. The new channels are B, C, and E. The new message types are Acquire, Probe, ProbeAck[Data], Release[Data], ReleaseAck, Grant[Data] and GrantAck.

9.3.1. AcquireBlock

An AcquireBlock message is a request message type used by a Master Agent with a cache to obtain a copy of a block of data that it plans to cache locally. Master Agents can also use this message type to upgrade the permissions they have on a block already in their possession (i.e., to gain write permissions on a read-only copy). Table 32 shows the encodings used for the fields of A for this message type.

a_opcode must be Acquire, which is encoded as 6.

a_param indicates the specific type of permissions change the Master Agent intends to occur. Possible transitions are selected from the **Grow** category of Table 31.

a_size indicates the total amount of data the requesting Master Agent wishes to cache, in terms of log2(bytes).

a_address must be aligned to a_size.

a_mask provides the byte select lanes, in this case indicating which bytes to read. See Section 4.5 for details. a_size, a_address and a_mask are required to correspond with one another. Acquires must have a contiguous mask that is naturally aligned.

a_source is the ID of the Master Agent issuing this request. It will be used by the responding Slave Agent to ensure the response is routed correctly.

a_data is ignored and can be assigned any value. a_corrupt is reserved and must be 0.

Table 32. Fields of AcquireBlock messages on Channel A.

Channel A	Type	Width	Encoding
a_opcode	C	3	Must be AcquireBlock (6).
a_param	C	3	Permissions transfer: Grow (NtoB, NtoT, BtoT).
a_size	C	*z*	2^n bytes will be read by the slave and returned.
a_source	C	*o*	The master source identifier issuing this request.
a_address	C	*a*	The target address of the Transfer, in bytes.
a_mask	D	*w*	Byte lanes to be read from.
a_corrupt	D	1	Reserved; must be 0.
a_data	D	*8w*	Ignored; can be any value.

9.3.2. AcquirePerm

An AcquirePerm message is a request message type used by a Master Agent with a cache to upgrade permissions on a block without supplying a copy of the data contained in the block. AcquirePerm must be only used in situations where no copy of the data is required to complete the initiating operation. The primary example is the case where the block is being acquired in order to be entirely overwritten. Table 33 shows the encodings used for the fields of Channel A for this message type.

a_opcode must be AcquirePerm, which is encoded as 7.

a_param indicates the specific type of permissions change the Master Agent intends to occur. Possible transitions are selected from the **Grow** category of Table 31.

a_size indicates the total amount of data the requesting Master Agent wishes to have permission to cache, in terms of log2(bytes). As in a Get message, an AcquirePerm message does not contain data itself. Additionally, no data will ever be returned to the requestor in response to this message.

a_address must be aligned to a_size.

a_mask provides the byte select lanes, in this case indicating which bytes to read. See Section 4.5 for details. a_size, a_address and a_mask are required to correspond with one another. AcquirePerm must have a contiguous mask that is naturally aligned.

a_source is the ID of the Master Agent issuing this request. It will be used by the responding Slave Agent to ensure the reponse is routed correctly.

a_data is ignored an can be assigned any value. a_corrupt is reserved and must be 0.

Table 33. Fields of ProbeBlock messages on Channel B.

Channel A	Type	Width	Encoding
a_opcode	C	3	Must be AcquirePerm (7).
a_param	C	3	Permissions transfer: Grow (NtoB, NtoT, BtoT).
a_size	C	*z*	2^n bytes will be read by the slave and returned.
a_source	C	*o*	The master source identifier issuing this request.
a_address	C	*a*	The target address of the Transfer, in bytes.
a_mask	D	*w*	Byte lanes to be read from.
a_corrupt	D	1	Reserved; must be 0.
a_data	D	*8w*	Ignored; can be any value.

9.3.3. ProbeBlock

A ProbeBlock message is a request message used by a Slave Agent to query or modify the permissions of a cached copy of a data block stored by a particular Master Agent. A Slave Agent may revoke a Master Agent's permissions on a cache block either in response to an Acquire from another master, or of its own volition. Table 34 shows all the fields of B for this message type.

b_opcode must be ProbeBlock, which is encoded as 6.

b_param indicates the specific type of permissions change the Slave Agent intends to occur. Possible transitions are selected from the **Cap** category of Table 31. Probing Master Agents to cap their permissions at a more permissive level than they currently have is allowed, and does not result in a permissions change.

b_size indicates the total amount of data the requesting agent wishes to probe, in terms of log2(bytes). If dirty data is written back in response to this probe, b_size represents the size of the resulting ProbeAckData message, not this particular ProbeBlock message.

b_address must be aligned to b_size.

b_mask provides the byte select lanes, in this case indicating which bytes to probe. See Section 4.5 for details. b_size, b_address and b_mask are required to correspond with one another. ProbeBlock messages must have a contiguous mask.

b_source is the ID of the Master Agent that is the target of this request. It is used to route the request, e.g., to a particular cache. See Section 6.4 for details.

b_data is ignored and can be assigned any value. b_corrupt is reserved and must be 0.

Table 34. Fields of ProbeBlock messages on Channel B.

Channel B	Type	Width	Encoding
b_opcode	C	3	Must be ProbeBlock (6).
b_param	C	3	Permission transfer: Cap (toN, toB, toT).
b_size	C	*z*	2^n bytes will be probed by the master and possibly returned.
b_source	C	*o*	The master source identifier being targeted by this request.
b_address	C	*a*	The target address of the Transfer, in bytes.
b_mask	D	*w*	Byte lanes to be read from.
b_corrupt	D	1	Reserved; must be 0.
b_data	D	*8w*	Ignored; can be any value.

9.3.4. ProbePerm

A ProbePerm message is a request message used by a Slave Agent to query or modify the permissions of a cached copy of a data block stored by a particular Master Agent. A Slave Agent may revoke a Master Agent's permissions on a cache block either in response to an Acquire from another master, or of its own volition. ProbePerm must only be used in situations where no copy of the data is required to complete the initiating operation. The primary example is the case where the block is being acquired in order to be entirely overwritten. Table 35 shows all the fields of Channel B for this message type.

b_opcode must be ProbePerm, which is encoded as 7.

b_param indicates the specific type of permissions change the Slave Agent intends to occur. Possible transactions are selected from the **Cap** category of Table 31. Probing Master Agents to cap their permissions at a more permissive level than they currently have is allowed, and does not result in a permissions change.

b_size indicates the total amount of data the requesting agent wishes to probe, in terms of log2(bytes). No data will ever be returned to the requestor in response to this message.

b_address must be aligned to b_size.

b_mask provides the byte select lanes, in this case indicating which bytes to probe. See Section 4.5 for details. b_size, b_address and b_mask are required to correspond with one another. ProbePerm messages must have a contiguous mask.

b_source is the ID of the Master Agent that is the target of this request. It is used to route the request, e.g. to a particular cache. See Section 6.4 for details.

b_data is ignored and can be assigned any value. b_corrupt is reserved and must be 0.

Table 35. Fields of ProbeBlock messages on Channel B.

Channel B	Type	Width	Encoding
b_opcode	C	3	Must be ProbePerm (7).
b_param	C	3	Permissions transfer: Cap (toN, toB, toT).
b_size	C	*z*	2^n bytes will be probed by the master.
b_source	C	*o*	The master source identifier being targeted by this request.
b_address	C	*a*	The target address of the Transfer, in bytes.
b_mask	D	*w*	Byte lanes to be read from.
b_corrupt	D	1	Reserved; must be 0.
b_data	D	*8w*	Ignored; can be any value.

9.3.5. ProbeAck

A ProbeAck message is a response message used by a Master Agent to acknowledge the receipt of a Probe. Table 36 shows all the fields of Channel C for this message type.

c_opcode must be ProbeAck, which is encoded as 4.

c_param indicates the specific type of permissions change that occurred in the Master Agent as a result of the Probe. Possible transitions are selected from the **Shrink** or **Report** category of Table 31. The former indicates that permissions were decreased whereas the latter reports what they were and continue to be.

c_size indicates the total amount of data that was probed, in terms of log2(bytes). This message itself does not carry data.

c_address is used to route the response to the original requestor. It must be aligned to c_size.

c_source is the ID of the Master Agent that is the source of this response.

c_data is ignored and can be assigned any value.

Table 36. Fields of ProbeAck messages on C.

Channel C	Type	Width	Encoding
c_opcode	C	3	Must be ProbeAck (4).
c_param	C	3	Permissions transfer: Shrink or Report (TtoB, TtoN, BtoN, TtoT, BtoB, NtoN).
c_size	C	*z*	2^n bytes were probed; copied from b_size.
c_source	C	*o*	The master source identifier of this response; copied from b_source.
c_address	C	*a*	The target address of the transfer; copied from b_address.
c_corrupt	D	1	Reserved; must be 0.
c_data	D	*8w*	Ignored; can be any value.

9.3.6. ProbeAckData

A ProbeAckData message is a response message used by a Master Agent to acknowledge the receipt of a Probe and write back dirty data that the requesting Slave Agent required. Table 37 shows all the fields of C for this message type.

c_opcode must be ProbeAckData, which is encoded as 5.

c_param indicates the specific type of permissions change that occurred in the Master Agent as a result of the Probe. Possible transitions are selected from the **Shrink** or **Report** category of Table 31. The former indicates that permissions were decreased whereas the latter reports what they were and continue to be.

c_size indicates the total amount of data that was probed, in terms of log2(bytes), as well as the amount of data contained in this message.

c_address is used to route the response to the original requestor.

c_source is the ID of the Master Agent that is the source of this response, copied from b_source.

c_data contains the data accessed by the operation. c_corrupt being HIGH indicates that the data in this beat is corrupt.

Table 37. Fields of ProbeAckData messages on C.

Channel C	Type	Width	Encoding
c_opcode	C	3	Must be ProbeAckData (5).
c_param	C	3	Permissions transfer: Shrink or report (TtoB, TtoN, BtoN, TtoT, BtoB, NtoN).
c_size	C	*z*	2^n bytes were probed and are being written back; copied from b_size.
c_source	C	*o*	The master source identifier of this response; copied from b_source.
c_address	C	*a*	The target address of the transfer; copied from b_address.
c_corrupt	D	1	Whether this beat of data is corrupt.
c_data	D	*8w*	The data payload being transferred.

9.3.7. Grant

A Grant message is both a response and a request message used by a Slave Agent to acknowledge the receipt of a Acquire and provide permissions to access the cache block to the original requesting Master Agent. Table 38 shows the encodings used for fields of D for this message type.

d_opcode must be Grant, which is encoded as 4.

d_param indicates the specific type of accesses that the Slave Agent is granting permission to occur on the cached copy of the block in the Master Agent as a result of the Acquire request. Possible permission transitions are selected from the **Cap** category of Table 31. Permissions are increased without specifying the original permissions. Permissions may exceed those requested by the a_param field of the original request.

d_size contains the size of the data whose permissions are being transferred, though this particular message contains no data itself. Must be identical to the original a_size.

d_sink is the identifier the of the agent issuing this message used to route its E response, whereas d_source should have been saved from a_source in the original Channel A request, and is now being re-used to route this response to the correct destination. See Section 6.4 for details.

d_data is ignored and can be assigned any value.

d_denied indicates that the slave did not process the permissions transfer. In this case, d_param should be ignored, meaning the coherence policy permissions of the block remain unchanged. d_corrupt is reserved and must be 0.

Table 38. Fields of Grant messages on D.

Channel D	Type	Width	Encoding
d_opcode	C	3	Must be Grant (4).
d_param	C	2	Permissions transfer: Cap (toT, toB, toN).
d_size	C	*z*	2^n bytes are being transferred by the slave; copied from a_size.
d_size	C	*o*	The master source identifier receiving this response; copied from a_source.
d_sink	C	*i*	The slave sink identifier issuing this request.
d_denied	C	1	The slave was unable to service the request.
d_corrupt	D	1	Reserved; must be 0.
d_data	C	*z*	Ignored; can be any value.

9.3.8. GrantData

A GrantData message is both a response and a request message used by a Slave Agent to provide an acknowledgement along with a copy of the data block to the original requesting Master Agent. Table 39 shows the encodings used for fields of the D for this message type.

d_opcode must be GrantData, which is encoded as 5.

d_param indicates the specific type of accesses that the Slave Agent is granting permissions to occur on the cached copy of the block in the Master Agent as a result of the Acquire request. Possible permission transitions are selected from the **Cap** category of Table 31. Permissions are increased without specifying the original permissions. Permissions may exceed those requested by the a_param field of the original request.

d_size contains the size of the data block whose permissions are being transferred, which corresponds to the size of the data being sent with this particular message. Must be identical to the original a_size.

d_sink is the identifier the of the agent issuing this response message, whereas used to route its E response, whereas d_source should have been saved from a_source in the original Channel A request, and is now being re-used to route this response to the correct destination. See Section Section 6.4 for details.

d_data contains the data being transferred by the operation, which will be cached by the Master Agent.

d_denied indicates that the slave did not process the permissions transfer. In this case, d_param should be ignored, meaning the coherence policy permissions ofthe block remain unchanged. d_corrupt being HIGH indicates that the data in this beat is corrupt. IF d_denied is HIGH then d_corrupt must also be high.

Table 39. Fields of GrantData messages on D.

Channel D	Type	Width	Encoding
d_opcode	C	3	Must be GrantData (5).
d_param	C	2	Permissions transfer: Cap (toT, toB, toN).
d_size	C	*z*	2^n bytes are being transferred by the slave; copied from a_size.
d_source	C	*o*	The master source identifier receiving this response; copied from a_source.
d_sink	C	*i*	The slave sink identifier issuing this response.
d_denied	C	1	The slave was unable to service the request.
d_corrupt	D	1	Indicates whether this data beat is corrupt.
d_data	D	*8w*	The data payload.

9.3.9. GrantAck

The GrantAck response message is used by the Master Agent to provide a final acknowledgment of transaction completion, and is in turn used to ensure global serialization of operations by the Slave Agent. Table 40 shows all the fields of this message on E.

e_sink should have been saved from the d_sink in the preceding Grant[Data] message, and is now being re-used to route this response to the correct destination.

Table 40. Fields of GrantAck messages.

Channel E	Type	Width	Encoding
e_sink	C	*i*	The slave sink identifier accepting this response; copied from d_sink.

9.3.10. Release

A Release message is a request message used by a Master Agent to voluntarily downgrade its permissions on a cached data block. Table 41 shows all the fields of C for this message type.

c_opcode must be Release, which is encoded as 6.

c_param indicates the specific type of permissions change that the Master Agent is initiating. Possible transitions are selected from the **Shrink** or **Report** category of Table 31, which indicates both what the permissions were and what they are becoming.

c_size indicates the total amount of cached data whose permissions are being released, in terms of log2(bytes). This message itself does not carry data.

c_address is used to route the response to the managing Slave Agentfor that address. It must be aligned to c_size.

c_source is the ID of the Master Agent that is the source of this request. The ID does not have to be the same as the ID used to Acquire the block originally, though it must correspond to the same Master Agent.

c_data is ignored and can be assigned any value.

c_corrupt is reserved and should be set to 0.

Table 41. Fields of Release messages on C.

Channel C	Type	Width	Encoding
c_opcode	C	3	Must be Release (6).
c_param	C	3	Permissions transfer: Shrink or Report (TtoB, TtoN, BtoN, TtoT, BtoB, NtoN).
c_size	C	***z***	2^n bytes are being downgraded by the master.
c_source	C	***o***	The master source identifier of this request.
c_address	C	***a***	The target address of the Transfer, in bytes.
c_data	D	***8w***	Ignored; can be any value.

9.3.11. ReleaseData

A ReleaseData message is a request message used by a Master Agent to voluntarily downgrade its permissions on a cached data block. and write back dirty data to the managing Slave Agent. Table 42 shows all the fields of C for this message type.

c_opcode must be ReleaseData, which is encoded as 7.

c_param indicates the specific type of permissions change that the Master Agent is initiating. Possible transitions are selected from the **Shrink** or **Report** category of Table 31, which indicates both what the permissions were and what they are becoming.

c_size indicates the total amount of cached data whose permissions are being released, in terms of log2(bytes), as well as the amount of data contained in this message.

c_address is used to route the response to the original requestor. It must be aligned to c_size.

c_source is the ID of the Master Agent that is the source of this response.

c_data contains the dirty data being written back by the operation. c_corrupt being HIGH indicates that this beat of data is corrupt.

Table 42. Fields of ReleaseData messages on C.

Channel C	Type	Width	Encoding
c_opcode	C	3	Must be ReleaseData (7).
c_param	C	3	Permissions transfer: Shrink or Report (BtoB, TtoN, BtoN, TtoT, BtoB, NtoN).
c_size	C	*z*	2^n bytes are being written back by the master.
c_source	C	*o*	The master source identifier of this reponse.
c_address	C	*a*	The target address of the Transfer, in bytes.
c_corrupt	D	1	Whether this beat of data payload is corrupt.
c_data	D	*8w*	The data payload.

9.3.12. ReleaseAck

A ReleaseAck message is a response message used by a Slave Agent to acknowledge the receipt of a Release[Data], and is in turn used to ensure global serialization of operations by the Slave Agent. Table 43 shows the encodings used for fields of Channel D for this message type.

d_opcode must be ReleaseAck, which is encoded as 6.

d_param is reserved and must be 0.

d_size contains the size of the data whose permissions were transferred, though this particular message contains no data itself. It can be saved from the c_size in the preceding Release[Data] message.

d_source should have been saved from the c_source in the preceding Release[Data] message and is now being re-used to route this response to the correct destination. d_sink is ignored and does not need to be unique across the ReleaseAcks that are being sent to respond to any unanswered releases. See Section 6.4 for details.

d_denied is reserved and must be 0.

d_data is ignored and can be assigned any value. d_corrupt is reserved and must be 0.

Table 43. Fields of ReleaseAck messages on D.

Channel D	Type	Width	Encoding
d_opcode	C	3	Must be ReleaseAck (6).
d_param	C	2	Reserved; must be 0.
d_size	C	*z*	Bytes transferred; copied from c_size.
d_source	C	*o*	The master source identifier receiving this response; copied from c_source.
d_sink	C	*i*	Ignored; can be any value.
d_denied	C	1	Reserved; must be 0.
d_corrupt	D	1	Reserved; must be 0.
d_data	D	*8w*	Ignored; can be any value.

9.4. TL-UL and TL-UH Messages on Channel A and D

In addition to new messages, TL-C specifies implied permission transfers for these existing messages from TL-UL and TL-UH.

- `Get` implicitly Cap the permissions as **None** (Invalid).
- `{PutFullData, PutPartialData, ArithmeticData, LogicalData}` implicitly Cap the permissions as *not* **Read+Write** [Trunk] or [Tip], i.e. as **None** [Invalid] or **Read** [Branch].

9.5. TL-UL and TL-UH messages on B and C

In addition to the three new operations (Acquire, Probe, Release), TL-C re-specifies all the operations from TL-UH on Channels B and C. This allows those channels to be used for forwarding Access and Hint operations to remote owners of cached data. In other words, implementations may choose to utilize an update-based protocol rather than an invalidation-based one.

9.5.1. Get

A `Get` message is a request made by an agent that would like to access a particular block of data in order to read it. Table 44 shows the encodings used for the fields of the B channel for this message type.

`b_opcode` must be `Get`, which is encoded as 4. `b_param` is currently reserved for future performance hints and must be 0.

`b_size` indicates the total amount of data the requesting agent wishes to read, in terms of log2(bytes). `b_size` represents the size of the resulting AccessAckData message, not this particular `Get` message.

`b_address` must be aligned to `b_size`.

`b_mask` provides the byte select lanes, in this case indicating which bytes to read. See Section 4.5 for details. `b_size`, `b_address` and `b_mask` are required to correspond with one another. `Get` messages must have a contiguous mask.

`b_source` is the ID of the Master Agent that is the target of this request. It is used to route the request. See Section 6.4 for details.

`b_data` is ignored and may take any value. `b_corrupt` is reserved and must be 0.

Table 44. Fields of Get messages on B.

Channel B	Type	Width	Encoding
b_opcode	C	3	Must be Get (4).
b_param	C	3	Reserved; must be 0.
b_size	C	*z*	2^n bytes will be read by the master and returned.
b_source	C	*o*	The master source identifier being targeted by this request.
b_address	C	*a*	The target address of the Access, in bytes.
b_mask	D	*w*	Byte lanes to be read from.
b_corrupt	D	1	Reserved; must b 0.
b_data	D	*8w*	Ignored; can be any value.

9.5.2. PutFullData

A PutFullData message is a request by an agent that would like to access a particular block of data in order to write it. Table 45 shows the encodings used for the fields of the B for this message type.

b_opcode must be PutFullData, which is encoded as 0. b_param is currently reserved for future performance hints and must be 0.

b_size indicates the total amount of data the requesting agent wishes to write, in terms of log2(bytes). In this case, b_size represents the size of this request message.

b_address must be aligned to b_size. The entire contents of b_address to (b_address + 2^{b_size} - 1) will be written.

b_mask provides the byte select lanes, in this case indicating which bytes to write. See Section 4.5 for details. One bit of b_mask corresponds to one byte of data written. b_size, b_address and *_mask are required to correspond with one another. PutFullData must have a contiguous mask, and if b_size is greater than or equal the width of the physical data bus then all b_mask must be HIGH.

b_source is the ID of the Master Agent that is the target of this request. It is used to route the request.

b_data is the actual data payload to be written. b_corrupt being HIGH indicates the data in this beat is corrupt.

Table 45. Fields of PutFullData messages on B.

Channel B	Type	Width	Encoding
b_opcode	C	3	Must be PutFull (0).
b_param	C	3	Reserved; must be 0.
b_size	C	*z*	2^n bytes will be written by the master.
b_source	C	*o*	The master source identifier being targeted by this request.
b_address	C	*a*	The target address of the Access, in bytes.
b_mask	D	*w*	Byte lanes to be written, must be contiguous.
b_corrupt	D	1	Whether this beat of data is corrupt.
b_data	D	*8w*	Data payload to be written.

9.5.3. PutPartialData

A PutPartialData message is a request by an agent that would like to access a particular block of data in order to write it. PutPartialData can be used to write arbitrary-aligned data at a byte granularity. Table 46 shows the encodings used for the fields of the B for this message type.

b_opcode must be PutPartialData, which is encoded as 1. b_param is currently reserved for future performance hints and must be 0.

b_size indicates the range of data the requesting agent will possibly write, in terms of log2(bytes). b_size also represents the size of this request message's data.

b_address must be aligned to b_size. Some subset of the contents of b_address to (b_address + $2^{b\_size} - 1$) will be written.

b_mask provides the byte select lanes, in this case indicating which bytes to write. See Section 4.5 for details. One bit of b_mask corresponds to one byte of data written. b_size, b_address and b_mask are required to correspond with one another, but PutPartialData may write less data than b_size, depending on the contents of b_mask. Any set bits of b_mask must be contained within an aligned region of b_size.

b_source is the ID of the master that is the target of this request. It is used to route the request.

b_data is the actual data payload to be written. b_data in a byte that is unmasked is ignored and can take any value. b_corrupt being HIGH indicates that masked data in this beat is corrupt.

Table 46. Fields of PutPartialData messages on B.

Channel B	Type	Width	Encoding
b_opcode	C	3	Must be PutFull (1).
b_param	C	3	Reserved; must be 0.
b_size	C	*z*	Up to 2^n bytes will be written by the master.
b_source	C	*o*	The master source identifier being targeted by this request.
b_address	C	*a*	The target base address of the Access, in bytes.
b_mask	D	*w*	Byte lanes to be written.
b_corrupt	D	1	Whether this beat of data is corrupt.
b_data	D	*8w*	Data payload to be written.

9.5.4. AccessAck

AccessAck provides a dataless acknowledgement to the original requesting agent. Table 47 shows the encodings used for fields of the Channel C for this message type.

c_opcode must be AccessAck, which is encoded as 0. c_param is reserved for use with TL-C opcodes and should be assigned 0.

c_size contains the size of the data that was accessed, though this particular message contains no data itself. The size and address fields must be aligned. c_address must match the b_address from the request that triggered this response. It is used to route this response back to the Tip.

c_source is the ID the of the agent issuing this response message. See Section 6.4 for details.

c_data is ignored and can be assigned any value. c_corrupt is reserved and must be 0.

Table 47. Fields of AccessAck messages on C.

Channel C	Type	Width	Encoding
c_opcode	C	3	Must be AccessAck (0).
c_param	C	3	Reserved; must be 0.
c_size	C	*z*	2^n bytes were accessed by the master.
c_source	C	*o*	The master source identifier issuing this response.
c_address	C	*a*	The target address of the operation, in bytes.
c_corrupt	D	1	Reserved; must be 0.
c_data	D	*8w*	Ignored, can be any value.

9.5.5. AccessAckData

AccessAckData provides an acknowledgement with data to the original requesting agent. Table 48 shows the encodings used for fields of the Channel C for this message type.

c_opcode must be AccessAckData, which is encoded as 1. c_param is reserved for use with TL-C opcodes and should be assigned 0.

c_size contains the size of the data that was accessed, which corresponds to the size of the data associated with this particular message. The size and address fields must be aligned.

c_address must match the b_address from the request that triggered this response. It is used to route this response back to the Tip.

c_source is the ID the of the agent issuing this response message. See Section 6.4 for details.

c_data contains the data accessed by the operation. Data can be changed between beats of a AccessAckData that is a burst. c_corrupt being HIGH indicates that this beat of data is corrupt.

Table 48. Fields of AccessAckData messages on C.

Channel C	Type	Width	Encoding
c_opcode	C	3	Must be AccessAckData (1).
c_param	C	3	Reserved; must be 0.
c_size	C	z	2^n bytes were accessed by the master.
c_source	C	o	The master source identifier issuing this response.
c_address	C	a	The target address of the Access, in bytes.
c_corrupt	D	1	Indicates whether this beat of data is corrupt.
c_data	D	$8w$	The data payload for messages with data.

9.5.6. ArithmeticData

An ArithmeticData message is a request made by an agent that would like to access a particular block of data in order to read-modify-write it with an arithmetic operation. Table 49 shows the encodings used for the fields of the B channel for this message type.

b_opcode must be ArithmeticData, which is encoded as 2.

b_param specifies the specific atomic operation to perform. The set of supported arithmetic operations is listed in Table 23. It consists of MIN, MAX, MINU, MAXU, ADD, representing signed and unsigned integer maximum and minimum, as well as integer addition.

b_size is the arithmetic operand size and reflects both the size of this request's data as well as the AccessAckData response.

b_address must be aligned to b_size.

b_mask provides the byte select lanes, in this case indicating which bytes to read-modify-write. See Section 4.5 for details. One bit of b_mask corresponds to one byte of data used in the atomic operation. b_size, b_address and b_mask are required to correspond with one another (i.e., the mask is also naturally aligned and fully set HIGH contiguously within that alignment).

b_source is the ID of the master that is the target of this request. It is used to route the request.

b_data contains one of the operands (the other is found at the target address). b_data in a byte that is unmasked is ignored and can take any value. b_corrupt being HIGH indicates that masked data in this beat is corrupt.

Table 49. Fields of ArithmeticData messages on B.

Channel B	Type	Width	Encoding
b_opcode	C	3	Must be ArithmeticData (3).
b_param	C	3	See Table 23.
b_size	C	*z*	2^n bytes will be read and written by the master.
b_source	C	*o*	The master source identifier being targeted by this request.
b_address	C	*a*	The target address of the Access, in bytes.
b_mask	D	*w*	Byte lanes to be read and written.
b_corrupt	D	1	Whether this beat of data is corrupt.
b_data	D	*8w*	Data payload to be used as operand.

9.5.7. LogicalData

A LogicalData message is a request made by an agent that would like to access a particular block of data in order to read-modify-write it with an logical operation. Table 50 shows the encodings used for the fields of the B channel for this message type.

b_opcode must be LogicalData, which is encoded as 2.

b_param specifies the specific atomic operation to perform. The set of supported logical operations is listed in Table 25. It consists of { XOR, OR, AND, SWAP }, representing bitwise logical xor, or, and, as well as a simple swap of the operands.

b_size is the operand size and reflects both the size of this request's data as well as the AccessAckData response.

b_address must be aligned to b_size. See Section 4.5 for details.

b_mask provides the byte select lanes, in this case indicating which bytes to read-modify-write. See Section 4.5 for details. One bit of b_mask corresponds to one byte of data used in the atomic operation. b_size, b_address and b_mask are required to correspond with one another (i.e., the mask is also naturally aligned and fully set HIGH contiguously within that alignment).

b_source is the ID of the master that is the target of this request. It is used to route the request.

b_data contains one of the operands (the other is found at the target address). b_data in a byte that is unmasked is ignored and can take any value. b_corrupt being HIGH indicates that masked data in this beat is corrupt.

Table 50. Fields of `LogicalData` messages on B.

Channel B	Type	Width	Encoding
`b_opcode`	C	3	Must be LogicalData (3).
`b_param`	C	3	See Table 25.
`b_size`	C	z	2^n bytes will be read and written by the master.
`b_source`	C	o	The slave source identifier being targeted by this request.
`b_address`	C	a	The target address of the Access, in bytes.
`b_mask`	D	w	Byte lanes to be read and written.
`b_corrupt`	D	1	Whether this beat of data is corrupt.
`b_data`	D	$8w$	Data payload to be written.

9.5.8. Intent

An `Intent` message is a request made by an agent that would like to signal its future intention to access a particular block of data. Table 51 shows the encodings used for the fields of the B channel for this message type.

`b_opcode` must be `Intent`, which is encoded as 5.

`b_param` specifies the specific intention being conveyed by this Hint operation. Note that its intended effect applies to the slave interface and further out in the hierarchy. The set of supported intentions is listed in Table 27. It consists of { `PrefetchRead`, `PrefetchWrite` }, representing prefetch-data-with-intent-to-read and prefetch-data-with-intent-to-write.

`b_size` is the size of data to which the attention applies. `b_address` must be aligned to `b_size`. `b_mask` provides the byte select lanes, in this case indicating the bytes to which the intention applies. See Section 4.5 for details. `b_size`, `b_address` and `b_mask` are required to correspond with one another.

`b_source` is the ID of the master that is the target of this request. It is used to route the request.

`b_data` is ignored and can take any value. `b_corrupt` is reserved and must be 0.

Table 51. Fields of `Intent` messages on B.

Channel B	Type	Width	Encoding
`b_opcode`	C	3	Must be `Intent` (5).
`b_param`	C	3	Intention encoding; See Table 27.
`b_size`	C	z	2^n bytes to which this intention applies.
`b_source`	C	o	The master source identifier being targeted by this request.
`b_address`	C	a	The address of the targeted cached block, in bytes.
`b_mask`	D	w	Byte lanes to which the Hint applies.
`b_corrupt`	D	1	Reserved; must be 0.
`b_data`	D	$8w$	Ignored; can be any value.

9.5.9. HintAck

HintAck serves as an acknowledgement response for a Hint operation. Table 52 shows the encodings used for fields of Channel C for this message type.

c_opcode must be HintAck, which is encoded as 2. c_param is reserved must be assigned 0.

c_size contains the size of the data that was hinted about, though this particular message contains no data itself. c_address is only required to be aligned to c_size.

c_source is the ID the of the agent issuing this response message, whereas c_source should have been saved from the request and is now being re-used to route this response to the correct destination. See Section 6.4 for details.

c_data is ignored and can be assigned any value. c_corrupt is reserved and must be 0.

Table 52. Fields of HintAck messages on C.

Channel C	Type	Width	Encoding
c_opcode	C	3	Must be HintAck (2).
c_param	C	3	Reserved; must be 0.
c_size	C	***z***	2^n bytes were hinted about.
c_address	C	***a***	The target address of the operation, in bytes.
c_source	C	***o***	The master source identifier issuing this response.
c_corrupt	D	1	Reserved; must be 0.
c_data	D	***8w***	Ignored; can be any value.

Glossary

Access

An operation that reads and/or writes the data at a specified address.

acknowledgement message

a message the other agent is required to send back if you send a request

Acquire

A Transfer operation whereby the master acquires permissions to cache a copy of the block from the slave.

agent

An active participant in the protocol that sends and receives messages in order to complete operations.

Atomic

An Access operation allowing the master to read-modify-write addresses managed by the slave.

beat

A single-clock-cycle slice of any message that takes multiple cycles to transmit over a channel of a particular width.

burst

A multi-beat message.

channel

A one-way communication link between a master interface and a slave interface carrying messages of homogenous priority.

Channel A

Transmits a request that an operation can be performed at a specified address, accessing or caching the data.

Channel B

Transmits a request that an operation be performed at a specified master, accessing or un-caching the data.

Channel C

Transmits a data or permissions acknowledgement for a Channel B request.

Channel D

Transmits a data or permissions acknowledgement to the original requestor.

Channel E

Transmits a final acknowledgement of a cache block transfer for the requestor, used for serialization.

DAG

Directed Acyclic Graph.

follow-up message

Any message sent as a result of receiving some other message.

forwarded message

A recursive message that is at the same level of priority as the message that initiated it.

Get

An Access operation allowing the master to read addresses managed by the slave.

Hint

An operation that is informational only and has no direct effect on data values.

Intent

A Hint operation that indicates the master intends to read or write data at addresses managed by the slave.

link

The set of channels required to complete operations between two agents.

master interface

Through which agents may request that memory operations be performed, or for permission to cache copies of data.

message

A set of control and data values sent over a particular channel.

MOESI

A cache coherence policy featuring an ownership state.

operation

A change to an address range’s data values, permissions or location in the memory hierarchy.

Probe

A Transfer operation whereby the slave revokes permissions to cache a copy of the block from the master.

Put

An Access operation allowing the master to write addresses managed by the slave.

receiver

The interface that accepts messages on a channel (raises ready).

recursive message

optional messages sent as a means of implementing an operation.

Release

A Transfer operation whereby the master voluntarily releases permission on the block back to the slave.

request message

A message specifying an access to perform or a change of permissions on a cached block.

response message

a message the other agent is required to send back if you send a request.

sender

The interface that originates messages on a channel (raises valid).

slave interface

Through which agents may grant permissions and access to a range of addresses, and respond with completed memory operations.

SoC

System-on-Chip.

TL-C

TileLink Cached.

TL-UH

TileLink Uncached Heavyweight.

TL-UL

TileLink Uncached Lightweight.

Transfer

An operation that moves permissions or cached copies of data through the network.