# SiFive TileLink Specification

© SiFive, Inc.

December 3, 2018

# SiFive TileLink Specification


## Copyright Notice

Copyright © 2018, SiFive Inc. All rights reserved.


## Release Information

| Version | Date | Changes |
| --- | --- | --- |
| 1.7.1-draft | August 22, 2017 | Pre-Release version. |
| 1.7.1 | December 3, 2018 | Release version. |

# Contents

**SiFive TileLink Specification** **i**

**1 Introduction** **1**

1.1 Protocol Conformance Levels . . . 2

1.2 Document Overview . . . 2

**2 Architecture** **3**

2.1 Network Topology . . . 4

2.2 Channel Priorities . . . 6

2.3 Address Space Properties . . . 7

**3 Signal Descriptions** **9**

3.1 Signal Naming Conventions . . . 10

3.2 Clocking, Reset, and Power . . . 11

3.2.1 Clock . . . 11

3.2.2 Reset . . . 11

3.2.3 Power or Clock Crossing . . . 11

3.3 Channel A (Mandatory) . . . 12

3.4 Channel B (TL-C only) . . . 13

3.5 Channel C (TL-C only) . . . 14

3.6 Channel D (Mandatory) . . . 15

3.7 Channel E (TL-C only) . . . 16

**4 Serialization** **17**

4.1 Flow Control Rules . . . 18

4.2 Deadlock Freedom . . . 20

4.2.1 Definitions Used in Rules . . . 20

4.2.2 Forward Progress Rules for Agents . . . 21
4.2.3 Topology Rules for Networks . . . 23
4.3 Request-Response Message Ordering . . . 24
4.3.1 Burst Responses . . . 25
4.3.2 Burst Requests . . . 26
4.3.3 Burst Requests and Responses . . . 27
4.4 Interfacing with Legacy Buses . . . 28
4.5 Errors . . . 29
4.6 Byte Lanes . . . 30

**5 Operations and Messages 33**
5.1 Operation Taxonomy . . . 33
5.2 Message Taxonomy . . . 34
5.3 Addressing . . . 37
5.4 Source and Sink Identifiers . . . 39
5.5 Operation Ordering . . . 41

**6 TileLink Uncached Lightweight (TL-UL) 43**
6.1 Flows and Waves . . . 44
6.2 Messages . . . 47
6.2.1 Get . . . 47
6.2.2 PutFullData . . . 48
6.2.3 PutPartialData . . . 49
6.2.4 AccessAck . . . 50
6.2.5 AccessAckData . . . 51

**7 TileLink Uncached Heavyweight (TL-UH) 53**
7.1 Flows and Waves . . . 55
7.2 Messages . . . 57
7.2.1 ArithmeticData . . . 58
7.2.2 LogicalData . . . 59
7.2.3 Intent . . . 60
7.2.4 HintAck . . . 61
7.3 Burst messages . . . 62

**8 TileLink Cached (TL-C) 63**

8.1 Implementing Cache Coherence Using TileLink . . . . . 64
8.1.1 Operations . . . . . 66
8.1.2 Channels . . . . . 67
8.1.3 Messages . . . . . 68
8.1.4 Permissions Transitions . . . . . 69
8.2 Flows and Waves . . . . . 70
8.3 TL-C Messages . . . . . 76
8.3.1 Acquire . . . . . 77
8.3.2 Probe . . . . . 78
8.3.3 ProbeAck . . . . . 79
8.3.4 ProbeAckData . . . . . 80
8.3.5 Grant . . . . . 81
8.3.6 GrantData . . . . . 82
8.3.7 GrantAck . . . . . 83
8.3.8 Release . . . . . 84
8.3.9 ReleaseData . . . . . 85
8.3.10 ReleaseAck . . . . . 86
8.4 TL-UL and TL-UH messages on Channel B and Channel C . . . . . 87
8.4.1 Get . . . . . 88
8.4.2 PutFullData . . . . . 89
8.4.3 PutPartialData . . . . . 90
8.4.4 AccessAck . . . . . 91
8.4.5 AccessAckData . . . . . 92
8.4.6 ArithmeticData . . . . . 93
8.4.7 LogicalData . . . . . 94
8.4.8 Intent . . . . . 95
8.4.9 HintAck . . . . . 96

**Glossary** **97**

# 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 MOESI-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
- power-aware signal encoding

| | 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 |

Table 1.1: TileLink conformance levels

## 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 1.1. 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.

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:

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

# Chapter 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.

**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.

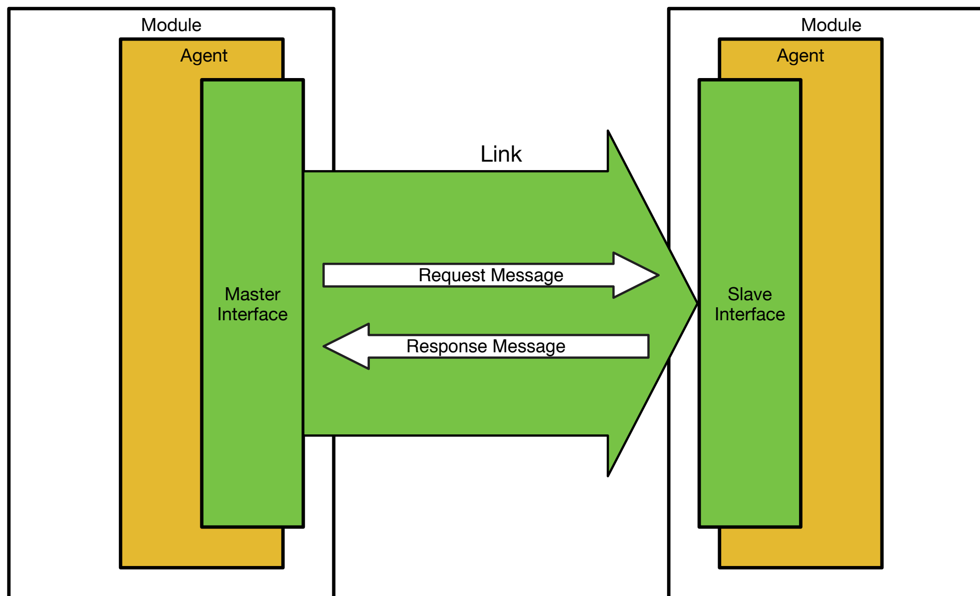


Figure 2.1: Overview of the most basic TileLink network operation. Two modules are connected by a link, with one module containing an agent with a master interface and the other module containing an agent with a slave interface. The agent with a master interface sends a request to an agent with a slave 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.

## 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 on the master interface.

Figure 2.1 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.

TileLink supports a wide variety of network topologies. Specifically, any topology that can be described as a Directed Acyclic Graph (DAG) is a legal topology, where agents are the vertices and links are the edges, with edges directed from master interfaces to slave interfaces. Figure 2.2 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.

It is important to note that a single hardware module can contain multiple independent TileLink agents. An example is shown in Figure 2.3, where the crossbar has one agent that routes data between links and a second agent that allows configuration state to be accessed. References to DAG topology in this specification refer to the graph of agents, not the hierarchy of modules.

Copyright © 2016, SiFive Inc. All rights reserved. 

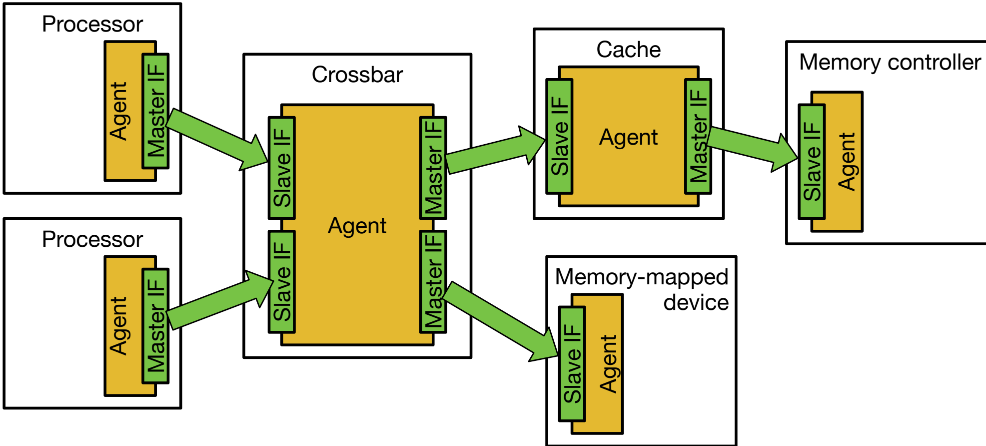


Figure 2.2: Example of a more complicated TileLink network topology (DAG), in which two modules contain an agent that has both a master interface and a slave interface.

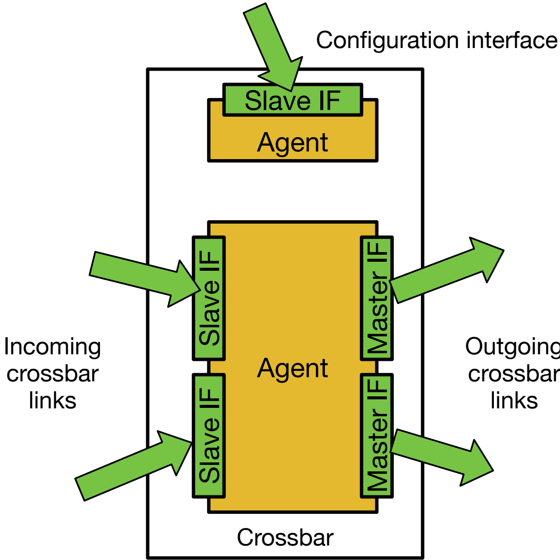


Figure 2.3: Example of a more complicated crossbar module that contains two agents. One agent has multiple interfaces and is used to route data in normal operation, while the other agent has a single slave interface to access configuration data for the crossbar.

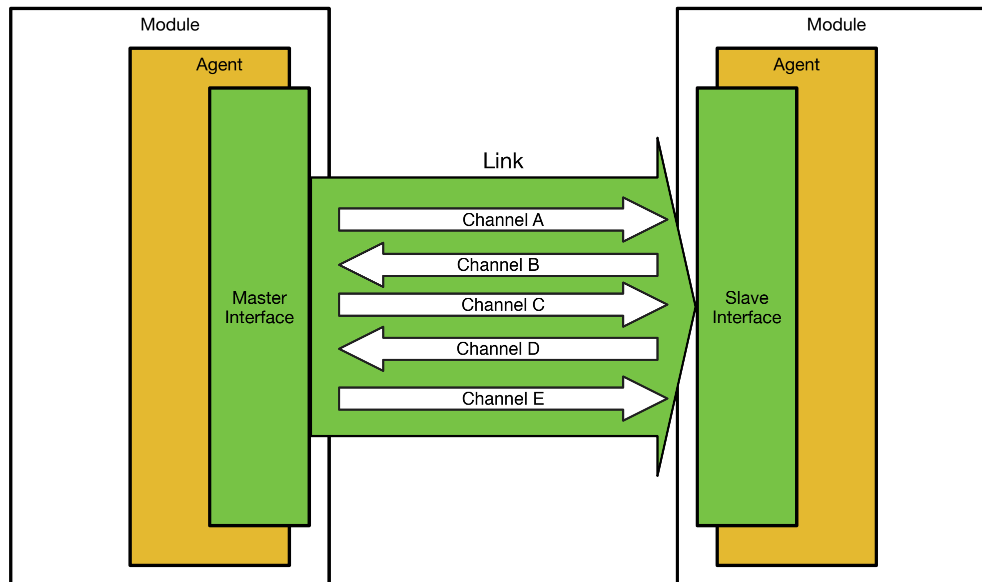


Figure 2.4: The five channels that comprise a TileLink link between any pair of agents.

## 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 actual copies of data. Channels are directional, in that each passes messages either from master to slave interface or from slave to master interface. Figure 2.4 illustrates the directionality of the five channels.

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 Channel B 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 a DAG. This is a necessary property for TileLink to remain deadlock free.

## 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, privelege 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.

# Chapter 3

# Signal Descriptions

This chapter tabulates all signals used by TileLink's five channels, which are summarized in Table 3.1. When combined with each channel's direction, the signal type in Table 3.2 determines signal direction. The widths of these signals are parameterized by values described in Table 3.3.

| 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 3.1: Overview of TileLink channels.

| 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 |
| **F** | Channel direction | Final signals, changing only once within each burst |
| **V** | Channel direction | Valid signal, indicates **C**/**D**/**F** contain valid data |
| **R** | Reverse direction | Ready signal, indicating that **V** was accepted |

Table 3.2: TileLink signal types. Channel direction is as indicated in Table 3.1.

| 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. |

Table 3.3: TileLink per-link channel parameters.

### 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.

| 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. |

Table 3.4: TileLink Clock and Reset Signals common to all channels

### 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.

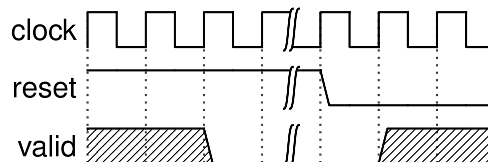


Figure 3.1: Valid must be driven LOW for at least 100 cycles during reset

### 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 coordinate with the rest of the SoC to ensure that there are no inflight 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.

| Signal | Type | Width | Description |
|---|---|---|---|
| `a_opcode` | C | 3 | Operation code. Identifies the type of message carried by the channel. (Table 5.2) |
| `a_param` | C | 3 | Parameter code. Meaning depends on `a_opcode`; specifies a transfer of caching permissions or a sub-opcode. (Sections 6.2, 7.2, 8.3) |
| `a_size` | C | $z$ | Logarithm of the operation size: $2^n$ bytes. (Section 4.6) |
| `a_source` | C | $o$ | Unique, per-link master source identifier. (Section 5.4) |
| `a_address` | C | $a$ | Target byte address of the operation. Must be aligned to `a_size`. (Section 4.6) |
| `a_mask` | D | $w$ | Byte lane select for messages with data. (Section 4.6) |
| `a_data` | D | $8w$ | Data payload for messages with data. (Section 4.6) |
| `a_valid` | V | 1 | The channel carries valid data. (Section 4.1) |
| `a_ready` | R | 1 | The channel data was accepted. (Section 4.1) |

Table 3.5: Channel A signals.

## 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.

| Signal | Type | Width | Description |
|---|---|---|---|
| b_opcode | C | 3 | Operation code. Identifies the type of message carried by the channel. (Table 5.2) |
| b_param | C | 3 | Parameter code. Meaning depends on b_opcode; specifies a transfer of caching permissions or a sub-opcode. (Section 8.3) |
| b_size | C | $z$ | Logarithm of the operation size: $2^n$ bytes. (Section 4.6) |
| b_source | C | $o$ | Unique, per-link master source identifier. (Section 5.4) |
| b_address | C | $a$ | Target byte address of the operation. Must be aligned to b_size. (Section 4.6) |
| b_mask | D | $w$ | Byte lane select for messages with data. (Section 4.6) |
| b_data | D | $8w$ | Data payload for messages with data. (Section 4.6) |
| b_valid | V | 1 | The channel carries valid data. (Section 4.1) |
| b_ready | R | 1 | The channel data was accepted. (Section 4.1) |

Table 3.6: Channel B signals.

### 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.

| Signal | Type | Width | Description |
|---|---|---|---|
| `c_opcode` | C | 3 | Operation code. Identifies the type of message carried by the channel. (Table 5.2) |
| `c_param` | C | 3 | Parameter code. Meaning depends on `c_opcode`; specifies a transfer of caching permissions. (Section 8.3) |
| `c_size` | C | $z$ | Logarithm of the operation size: $2^n$ bytes. (Section 4.6) |
| `c_source` | C | $o$ | Unique, per-link master source identifier. (Section 5.4) |
| `c_address` | C | $a$ | Target byte address of the operation. Must be aligned to `c_size`. (Section 4.6) |
| `c_data` | D | $8w$ | Data payload for messages with data. (Section 4.6) |
| `c_error` | F | 1 | The master agent was unable to service the request. (Section 4.5) |
| `c_valid` | V | 1 | The channel carries valid data. (Section 4.1) |
| `c_ready` | R | 1 | The channel data was accepted. (Section 4.1) |

Table 3.7: Channel C signals.

## 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.

| Signal | Type | Width | Description |
|---|---|---|---|
| d_opcode | C | 3 | Operation code. Identifies the type of message carried by the channel. (Table 5.2) |
| d_param | C | 2 | Parameter code. Meaning depends on d_opcode; specifies permissions to transfer or a sub-opcode. (Sections 6.2, 7.2, 8.3) |
| d_size | C | $z$ | Logarithm of the operation size: $2^n$ bytes. (Section 4.6) |
| d_source | C | $o$ | Unique, per-link master source identifier. (Section 5.4) |
| d_sink | C | $i$ | Unique, per-link slave sink identifier. (Section 5.4) |
| d_data | D | $8w$ | Data payload for messages with data. (Section 4.6) |
| d_error | F | 1 | The slave was unable to service the request. (Section 4.5) |
| d_valid | V | 1 | The channel carries valid data. (Section 4.1) |
| d_ready | R | 1 | The channel data was accepted. (Section 4.1) |

Table 3.8: Channel D signals.

### 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.

| Signal | Type | Width | Description |
|---|---|---|---|
| e_sink | C | $i$ | Unique, per-link slave sink identifier. (Section 5.4) |
| e_valid | V | 1 | The channel carries valid data. (Section 4.1) |
| e_ready | R | 1 | The channel data was accepted. (Section 4.1) |

Table 3.9: Channel E signals.

# Chapter 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 the receiver. For the B and D channels, the agent with the slave interface is the sender and the agent with the master interface 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.

To regulate the flow of beats in TileLink channels, receivers raise the channel `ready` 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. Conversely, the sender of a beat raises the channel `valid` signal to indicate the presence of a beat on the channel. Only when both `ready` and `valid` are raised is the beat exchanged.

The rest of this chapter lays out the flow control and deadlock avoidance rules used to govern when `ready` and `valid` may be toggled. We also define how TileLink agents may be connected together, and define 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 receiver may only hold `ready` LOW in accordance with the deadlock freedom rules in Section 4.2.

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.

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.
- Final signals changing only once within each burst.

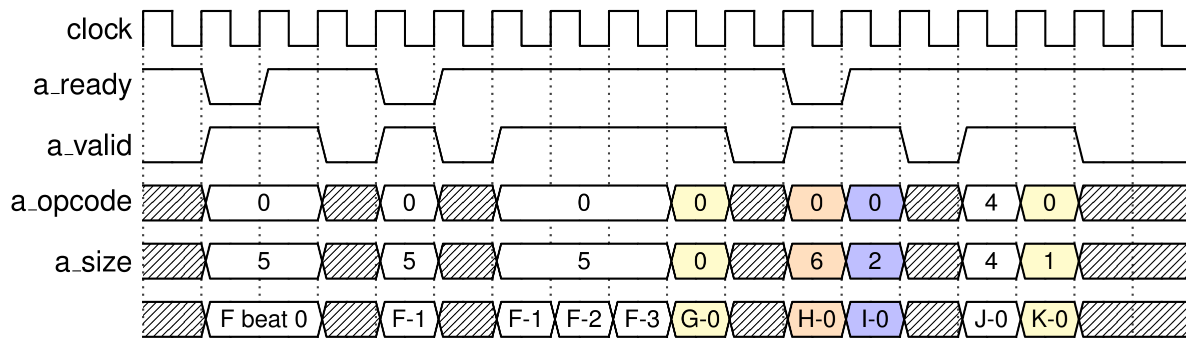


Figure 4.1: Ready-Valid Signaling for 6 messages in a 64-bit A Channel

One waveform which obeys these rules is illustrated for an 8-byte-wide channel in Figure 4.1. 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.

There are 6 messages sent in this figure: `F, G, H, I, J, K`. The first message, `F`, has size 5, which indicates the operation accesses $2^5 = 32$ bytes. Opcode 0 is a `PutFullData` message, so `F` carries data. Because Channel A carries 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 fits into 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.

## 4.2 Deadlock Freedom

TileLink is designed to be deadlock-free by construction. To guarantee that a TileLink network will never deadlock, we specify two sets of rules to which conforming systems must adhere. First, we define rules that govern the conditions under which a receiving agent may reject a beat of a message by lowering `ready`. Second, we define rules on allowable topologies of a TileLink network: The structure of agents and links must be a DAG.

*By combining these two rulesets with the strict prioritization of channels within the network, we can provably guarantee that correct TileLink implementations will not deadlock.*

### 4.2.1 Definitions Used in Rules

All TileLink operations comprise an ordered sequence of messages that are sent between agents. We use the following terms when defining the deadlock-freedom rules that apply to a given message based on its relative position in the overall ordering:

**request message:** a message specifying an operation to perform (access or transfer).

**follow-up message:** a message sent as a result of receiving some other message.

**response message:** a mandatory follow-up message paired to a request.

**recursive message:** any follow-up message nested inside a request/response pair.

**forwarded message:** a recursive message that is at the same level of priority as the message that initiated it.

Every request message must eventually be answered with a response message. A response message always has higher priority than its initiating request message. An individual message may be both a request and a response; responses that are also requests will trigger a further response.

A recursive message $X$ nested inside request $W$ and response $Z$ must have greater than or equal priority to $W$ and less than $Z$. $X$ itself must bet sent after $W$ and before $Z$. If the recursive request $X$ has a response $Y$, $Y$ must have a priority less than or equal to $Z$ and be received after $X$ is sent and before $Z$ is sent.

A recursive message that has the same priority as the message that triggered it is termed a forwarded message. An agent that has both one or more master interfaces and one or more slave interfaces, and that forwards messages from one to the other is termed a *forwarding agent*. Forwarding agents are important in constructing topologies of hierarchical memories and buses, and have additional rules governing how the `ready` signals of their master and slave interfaces are coupled, as explained in the following subsection.

### 4.2.2 Forward Progress Rules for Agents

A receiver is under no obligation to present `ready` HIGH when `valid` is LOW. However, when a sender presents `valid` HIGH, `ready` must be HIGH unless the receiver has a legitimate reason for rejecting the beat. In TileLink, there are only **four** legitimate reasons a conforming agent may reject a `valid` beat by lowering `ready`:

1. A receiver may choose to enter a bounded busy period, during which it never raises `ready`.
   - There must exist a fixed number of cycles that the bounded busy period is guaranteed to never exceed.
   - The receiver may enter a busy period arbitrarily, but between busy periods it must accept at least one beat.
   - For example, when dealing with periodic busy periods (e.g., a DDR refresh), this restriction can be met by placing a single entry buffer in front of the controller. The buffer agent raises `ready` until the buffer is filled. Then, when the controller has completed its refresh, it can drain the buffer and process the stored beat, making the buffer agent raise `ready` within a fixed number of cycles.
2. While a response to a request message received on channel $X$ is being rejected, the responding agent may lower `ready` on all channels with priority $\leq X$ indefinitely.
   - The complete list of response messages triggering this rule can be found in Table 5.3.
   - For example, consider a simple slave that received a `Get` on channel A and is now trying to send the response message `AccessAckData` out channel D. If that response is blocked because `d_ready` is LOW, then the slave may hold its `a_ready` LOW.
   - If a TL-C agent received a request on channel A and is blocked trying to send a response out channel D, this rule does **not** permit blocking channels B or C, but only channel A.
3. While a recursive message following a request message from channel $X$ is being rejected, the sender may lower `ready` on all channels with priority $\leq X$ indefinitely. *(Relevant only to forwarding or TL-C agents.)*
   - For example, consider a crossbar that is trying to forward a `Get` on its slave-side channel A output. While the crossbar waits for this message to be accepted, it may hold `ready` LOW on all of its master-side channel A inputs.
4. While a response to a message sent on channel $X$ has not been received, the receiver may lower `ready` on all channels with priority $\leq X$ indefinitely. *(Relevant only to forwarding or TL-C agents.)*
   - Agents may only wait for responses to messages whose last beat has already been sent.
   - For example, consider a crossbar that previously forwarded a `Get` on its slave-side channel A output. While the crossbar waits for a response, it may hold `ready` LOW on all of its master-side channel A inputs

These four rules are exhaustive. If you are writing an agent, you must ensure that if your `ready` is LOW while `valid` is HIGH one of the four rules applies. An agent which fails to abide by these rules in all cases is non-conforming and jeopardizes the forward progress of the whole TileLink network.

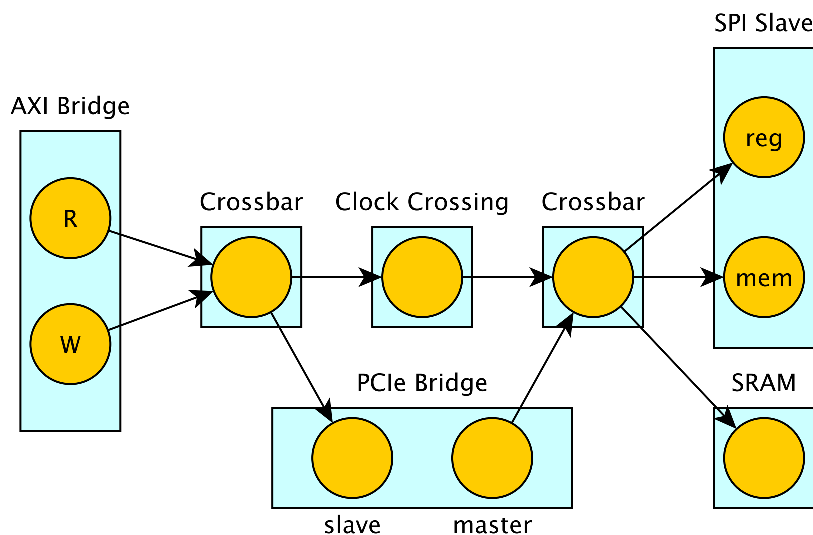


Figure 4.2: A TileLink agent graph, with boxes denoting RTL modules and circles agents.

### 4.2.3 Topology Rules for Networks

Every TileLink network can be represented as an agent graph, and this graph can be used to determine whether the network guarantees deadlock freedom. The TileLink agent graph contains one node per agent and one edge for each TileLink link. Edges in the agent graph point from master to slave. Figure 4.2 illustrates an example of a TileLink agent graph. Blue boxes indicate RTL modules, while yellow circles indicate agents. *A legal TileLink system must have a directed acyclic agent graph (DAG).* Any cycles in the graph can lead to deadlock.

To clarify this restriction, we need a more precise definition of a TileLink agent. The intuitive definition is that two TileLink links are connected to the same TileLink agent if a message from one link can result in a recursive message on the other link without first passing through any other TileLink link. Notice that even if two different TileLink links connect to the same RTL module, this does not mean that the module is a single agent with two links; there might be two distinct agents inside the module.

Consider, for example, a TileLink crossbar connected to many TileLink links. A message received on any slave port might be forwarded to any master port, and vice versa. Therefore, all the ports of the crossbar (and all of its links) connect to a single TileLink crossbar agent.

Conversely, consider a TileLink to PCIe bridge. It has one slave port and one master port. TileLink messages sent to the slave port do not cause recursive message to arrive on the master port. Similarly, messages sent to the master port do not cause recursive message to arrive on the slave port. Therefore, the bridge is actually two independent agents contained in one RTL module.

After constructing the agent graph and ruling it a DAG, and assuming that all agents abide by the forward progress rules delineated in the previous section, we can then prove that the TileLink network is guaranteed to be deadlock free. The general form of this proof is beyond the scope of this document.

## 4.3 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 may be presented on the response channel:

- on the same cycle that the first beat of the request message is accepted, but not before.
- before all the beats of a burst request message have been accepted.
- after an arbitrarily long delay.

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 meanwhile.

The fact that a response message can be recieved concurrently and combinationally with the first beat of the request message being accepted (and possibly before the request message has even finished being sent), interacts with the forward progress rules in Section 4.2.2. 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 `ready` 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 buffer after its Channel D input. The buffer absorbs a concurrent Channel D response message and presents `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.

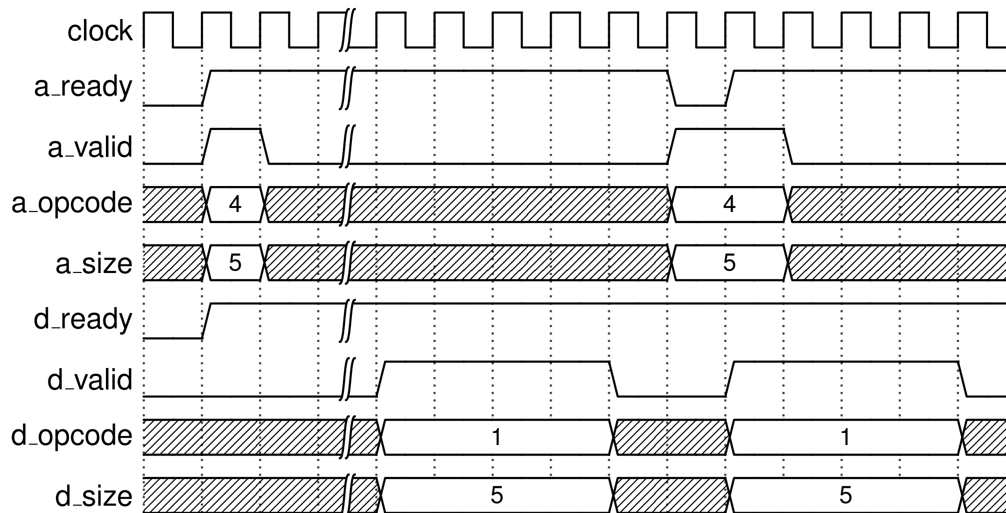


Figure 4.3: Max and min delay between a Get (4) and an AccessAckData (1) on an 8-byte bus.

### 4.3.1 Burst Responses

Figure 4.3 illustrates two Get operations. The `Get` request messages (opcode 4) are sent out Channel A. They are both accessing $2^5 = 32$ bytes, which takes 4 beats of data on an 8-byte bus. We see their 4-beat `AccessAckData` (opcode 1) response messages arriving on Channel D. The first response message arrives after an arbitrary delay. The master interface must be willing to wait indefinitely for this response 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.

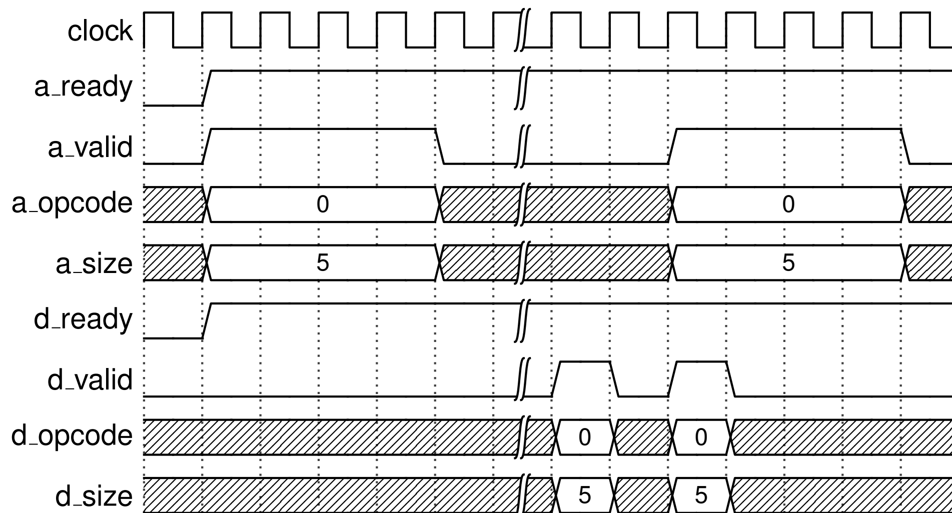


Figure 4.4: Max and min delay between a PutFullData (0) and an AccessAck (0) on an 8-byte bus.

### 4.3.2 Burst Requests

Figure 4.4 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 beat 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.

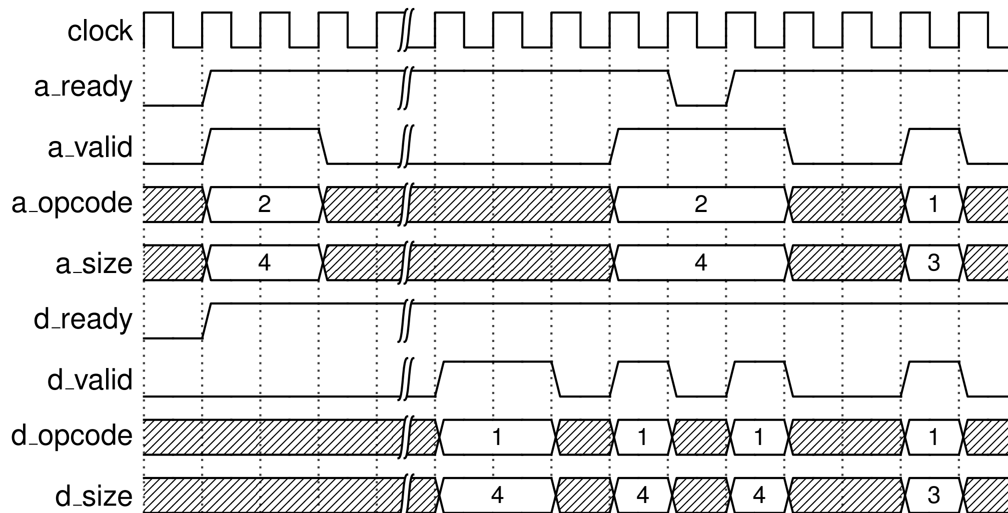


Figure 4.5: Delay between an ArithmeticData (2) and an AccessAckData (1) on an 8-byte bus.

### 4.3.3 Burst Requests and Responses

The situation for request messages and response messages that both carry a data payload follows the same rules. The first beat of the response message 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 4.5 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.

### 4.4 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 first forward progress rule. 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.

## 4.5 Errors

The C and D channels contain a single bit field with which to signal errors. Use of this field depends on the specific message type as identified by `*_opcode` and described in Chapters 6–8. As every TileLink request message requires a mandatory response message of a mandatory size, this field allows compliant messages to be created even when data corruption is detected by an agent.

The error field indicates that any one or more of the data beats associated with the message may contain erroneous data. All beats of the message must be sent regardless of whether an error is finally raised.

The error field can only be raised HIGH a single time within a burst. Once raise HIGH it must remain HIGH for the duration of the burst.

How errors that have been signaled via this field are reported to the broader SoC environment is beyond the scope of this document.

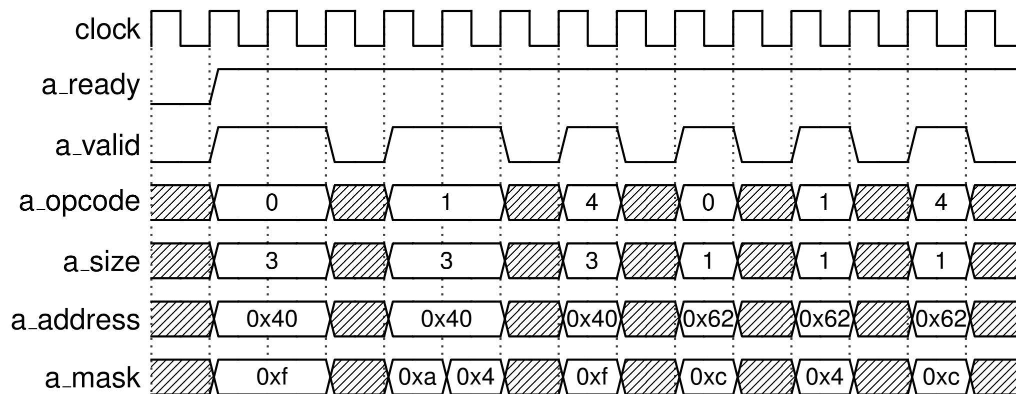


Figure 4.6: Example of the mask bits carried by byte lanes. `PutFullData` (0) must drive all active lanes of `mask` HIGH. Thus, the first message has all beats HIGH over multiple beats. In comparison, `PutPartialData` (1) may drive active lanes of `mask` HIGH or LOW for all beats. `Get` (4) messages 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).

## 4.6 Byte Lanes

TileLink channels which carry a data field always carry payload data little-endian naturally aligned. In other words, if the data bus width is $w$ bytes (which must be a power of two), then $(\texttt{address} \ \& \ !(w-1))$ is the address of the data found in the zeroth byte lane. For example, 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 4.7.

TileLink operations always describe power-of-two-sized byte ranges with an aligned address. Mathematically, $(\texttt{address} \ \& \ ((1 << \texttt{size}\ ) - 1) = 0)$ always holds. Therefore, either an operation uses all of the data byte lanes or it uses a power-of-two slice of them. The byte lanes used by an operation are called the active byte lanes. In Figure 4.7, the inactive byte lanes are crossed out.

On channels A and B, which carry a `mask` field, the `mask` must 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.

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 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 4.6.

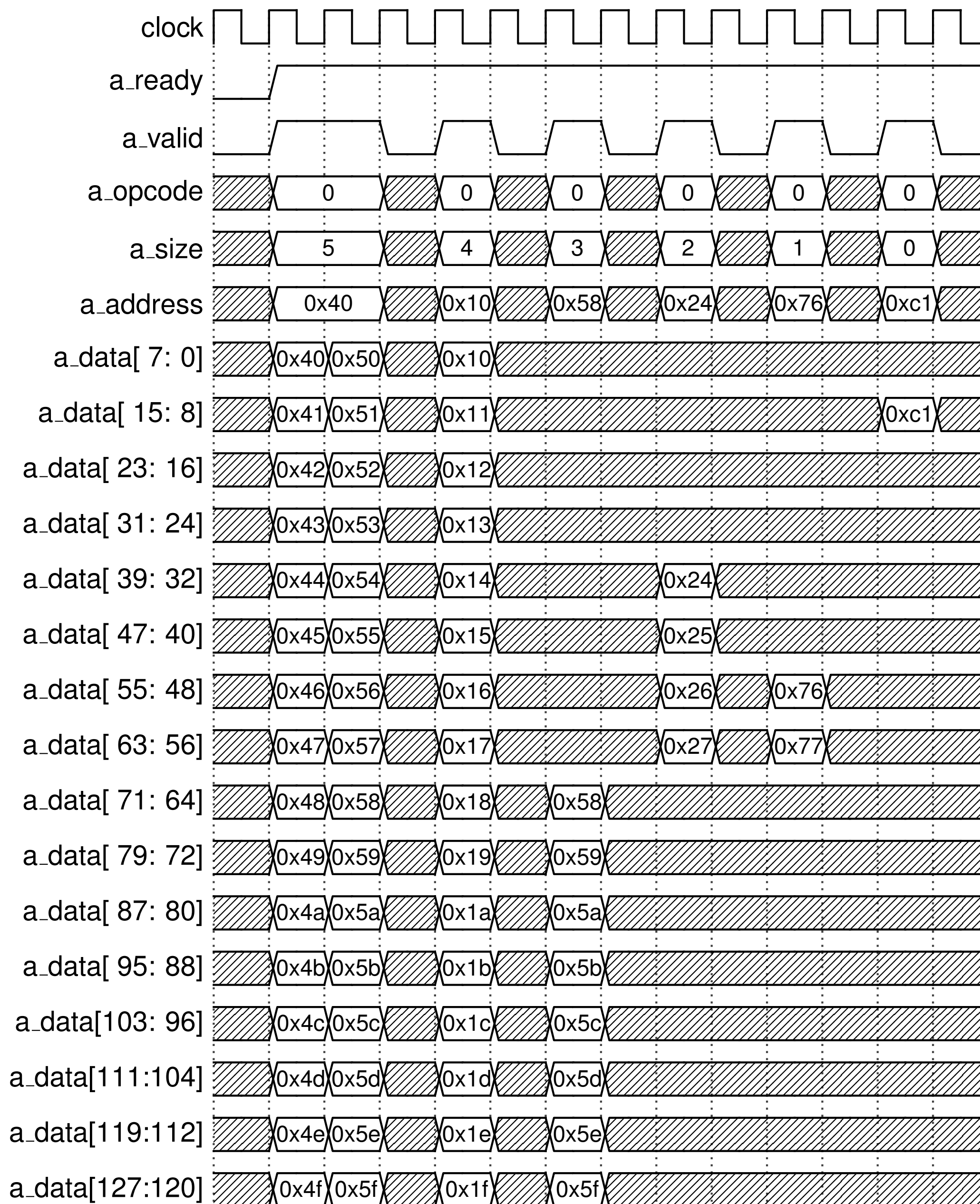


Figure 4.7: Example of the addresses of data carried in byte lanes on a 16-byte data bus. Notice that the lowest nibble of the address of data carried in each byte lane is constant. Meanwhile, not all byte lanes are used if the `size` is smaller than the data bus width. Multi-beat burst operations auto-increment their data addresses, while control signals remain constant.

# Chapter 5

# 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 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: TL-UL in 6, TL-UH in 7, and TL-C in 8.

## 5.1 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 5.1.

| 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 |

Table 5.1: Summary of TileLink Operations

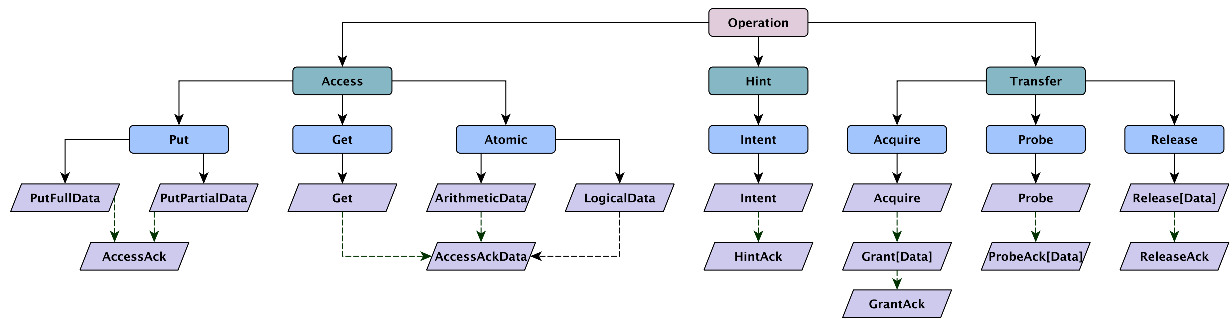


Figure 5.1: Taxonomy of all operations (blue boxes) and their constituent messages (purple parallelograms). Dotted arrows indicate request-response message pairs. TL-UL conformance only requires supporting Get and Put Access operations. TL-UH conformance requires all Hint and Access operations. TL-C conformance requires all operations.

## 5.2 Message Taxonomy

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 some messages must result in the eventual exchange of a response message sent to the requestor. A graphical representation of the operation and message taxonomy with responses shown can be seen in Figure 5.1. Table 5.2 lists all messages used in TileLink, grouped by conformance level and operation. Table 5.3 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.

*Copyright © 2016, SiFive Inc. All rights reserved.*


| Message | Operation | Opcode | A | B | C | D | E | Response |
|---|---|---|---|---|---|---|---|---|
| `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-UL*** |
| `ArithmeticData` | Atomic | 2 | y | y | . | . | . | `AccessAckData` |
| `LogicalData` | Atomic | 3 | y | y | . | . | . | `AccessAckData` |
| `Intent` | Intent | 5 | y | y | . | . | . | `HintAck` |
| `HintAck` | Intent | 2 | . | . | y | y | . | ***TL-UH*** |
| `Acquire` | Acquire | 6 | y | . | . | . | . | `Grant` or `GrantData` |
| `Grant` | Acquire | 4 | . | . | . | y | . | `GrantAck` |
| `GrantData` | Acquire | 5 | . | . | . | y | . | `GrantAck` |
| `GrantAck` | Acquire | - | . | . | . | . | y | |
| `Probe` | Probe | 6 | . | y | . | . | . | `ProbeAck` or `ProbeAckData` |
| `ProbeAck` | Probe | 4 | . | . | y | . | . | |
| `ProbeAckData` | Probe | 5 | . | . | y | . | . | |
| `Release` | Release | 6 | . | . | y | . | . | `ReleaseAck` |
| `ReleaseData` | Release | 7 | . | . | y | . | . | `ReleaseAck` |
| `ReleaseAck` | Release | 6 | . | . | . | y | . | ***TL-C*** |

Table 5.2: Summary of TileLink messages, grouped by conformance level and operation.

| Channel | Opcode | Message | Operation | Response |
|---|---|---|---|---|
| A | 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 | `Acquire` | Acquire | `Grant` or `GrantData` |
| 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 | `Probe` | Acquire | `ProbeAck` or `ProbeAckData` |
| C | 0 | `AccessAck` | Put | |
| | 1 | `AccessAckData` | Get or Atomic | |
| | 2 | `HintAck` | Intent | |
| | 4 | `ProbeAck` | Acquire | |
| | 5 | `ProbeAckData` | Acquire | |
| | 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 | |

Table 5.3: Summary of TileLink messages, ordered by channel and opcode.

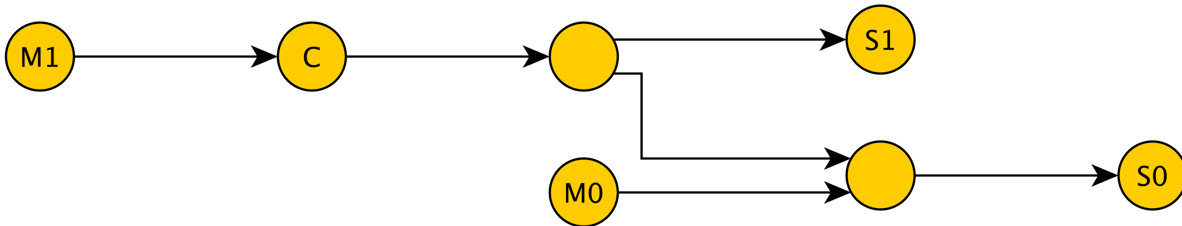


Figure 5.2: A simple agent graph, showing two masters M0 and M1 who can both access slave S0, while only M1 can access S1, via a cache C.

## 5.3 Addressing

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, privelege 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 5.2. 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 5.2 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.

| Channel | Dest. | Sequence | Routed By | Provides | → For Use As |
|---|---|---|---|---|---|
| A | slave | request | a_address | a_source | → d_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_sink | . | |

Table 5.4: Summary of TileLink routing fields

## 5.4 Source and Sink Identifiers

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 inflight message can be uniquely identified with a specific ongoing operation. Table 5.4 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 5.4. For example, if a Get request had a_source = 4, then the AccessAckData 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 inflight operations, we impose per-link and per-channel uniqueness constraints on inflight identifiers. An identifer is said to be *inflight* if an outstanding request using it has not yet received a response. Each inflight *request identifier* of a channel in a particular link must be unique, as defined below.

Channels A and C route their requests based soley 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 inflight A and 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 is inflight.

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 uniquely inflight. 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 `c_source` associated with the sender; this signal is ignored for those message types.

Channel D must provide unique `d_sink` transaction identifiers for inflight `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.

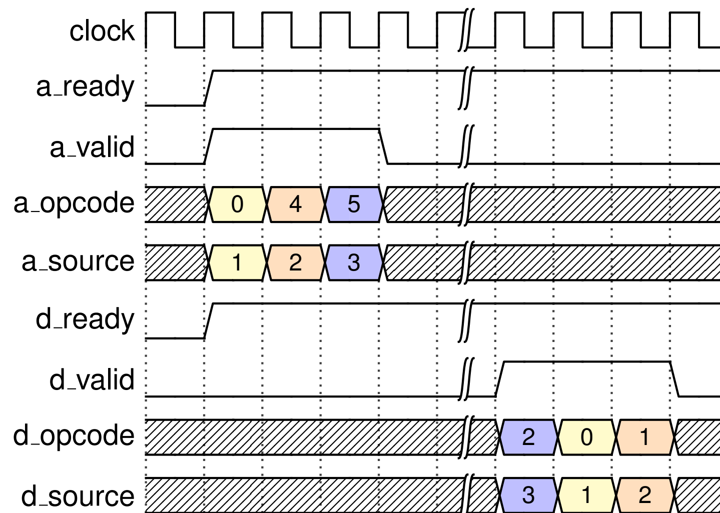


Figure 5.3: Operations do not necessarily receive responses in order.

## 5.5 Operation Ordering

Within a TileLink network, there may be multiple outstanding operations inflight 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 can not 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 instrutions 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.

# Chapter 6

# 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 4.3. 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 6.1 enumerates these messages.

| Message | Opcode | Operation | A | D | Response |
|---|---|---|---|---|---|
| `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 | |

Table 6.1: Summary of TL-UL messages.

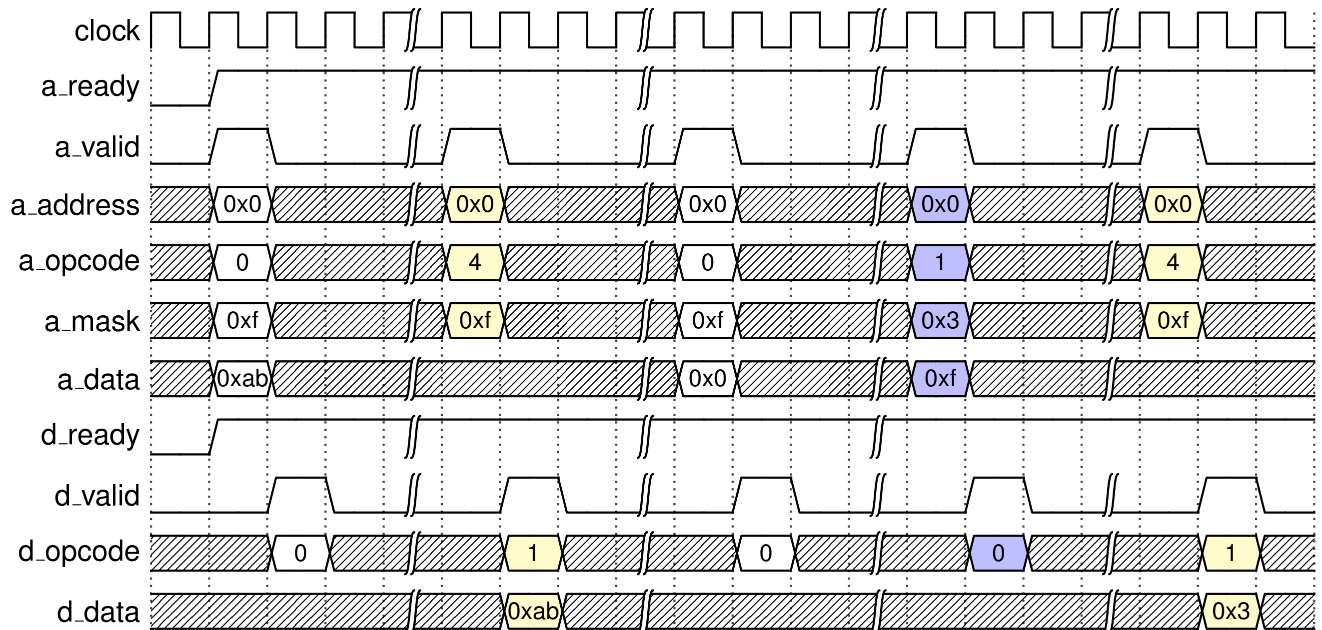


Figure 6.1: Waveform containing Get and Put operations. PutFullData writes 0xab; Get reads 0xab; PutFullData writes 0x0; PutPartialData writes 0x3; Get reads 0x3.

## 6.1 Flows and Waves

The figures in this section provide waveforms and message sequence charts for the TL-UL operations. Figure 6.1 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 6.2 shows the message flow employed by Get operations between a single pair of agents. Figure 6.3 shows the message flow employed by Put operations between a single pair of agents. Figure 6.4 shows the message flow employed by either operation through two levels of master-slave agent pairs.

Copyright © 2016, SiFive Inc. All rights reserved. 

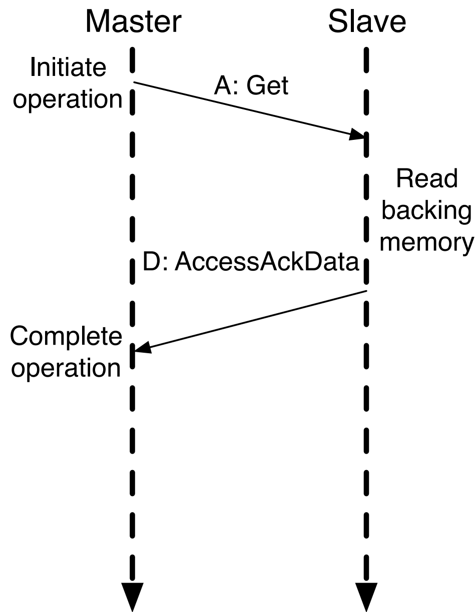


Figure 6.2: Overview of the Get message flow. A master sends a `Get` to a slave. Having read the required data, the slave responds to the master with an `AccessAckData`.

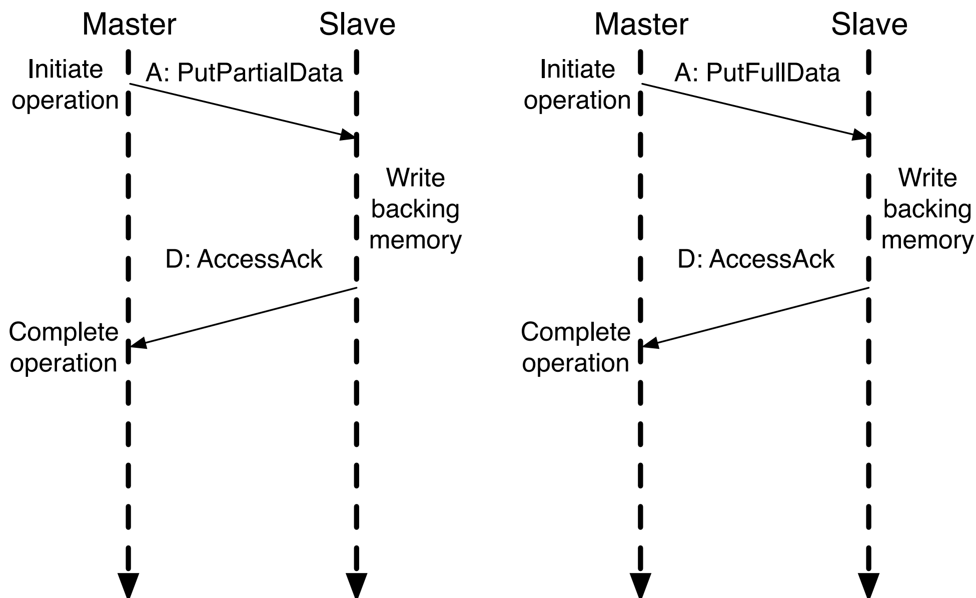


Figure 6.3: Overview of the Put message flows. A master sends an `PutPartialData` or `PutFullData` to a slave. After writing the included data, the slave responds to the master with a `AccessAck`.

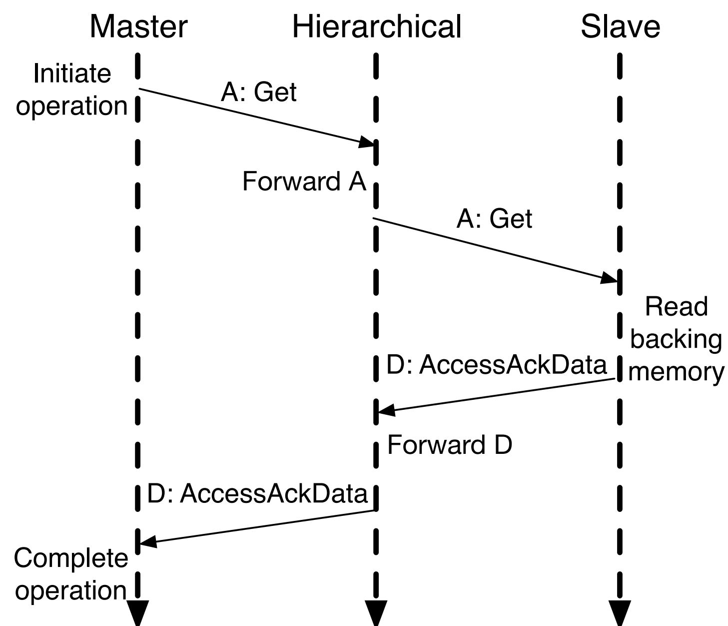


Figure 6.4: Message flow across multiple hierarchical agents to perform a memory access that reads a block of data. The Hierarchical Agent forwards the `Get` to the outer Slave Agent and then also forwards the response `AccessAckData` to the Master Agent.

## 6.2 Messages

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

### 6.2.1 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 6.2 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 5.4).

`a_address` must be aligned to `a_size`.

`a_mask` selects the byte lanes to read (Section 4.6). `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.

| **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_data` | D | $8w$ | Ignored; can be any value. |

Table 6.2: Fields of `Get` messages.

### 6.2.2 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 Chapter 7. Table 6.2 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 5.4).

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 4.6). 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.

| Channel A | Type | Width | Encoding |
|---|---|---|---|
| a_opcode | C | 3 | Must be PutFullData (0). |
| 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_data | D | $8w$ | Data payload to be written. |

Table 6.3: Fields of PutFullData messages.

### 6.2.3 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 6.4 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 posibly 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 5.4).

`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 4.6). 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.

| Channel A | Type | Width | Encoding |
|---|---|---|---|
| `a_opcode` | C | 3 | Must be `PutPartialData` (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 identifier 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_data` | D | $8w$ | Data payload to be written. |

Table 6.4: Fields of PutPartialData messages.

### 6.2.4 AccessAck

AccessAck serves as a data-less acknowledgement message to the original requesting agent. Table 6.5 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 5.4).

d_sink is ignored and can be assigned any value.

d_data is ignored and can be assigned any value.

d_error indicates that an error occured when the slave attempted to process the memory access.

| 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_data | D | $8w$ | Ignored; can be any value. |
| d_error | F | 1 | The slave was unable to service the request. |

Table 6.5: Fields of AccessAck messages.

### 6.2.5 AccessAckData

`AccessAckData` serves as an acknowledgement message including data to the original requesting agent. Table 6.6 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 5.4).

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

`d_data` contains the data that was accessed by the operation.

`d_error` indicates that an error occured when the slave attempted to process the memory access.

| 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_data` | D | $8w$ | The data payload. |
| `d_error` | F | 1 | The slave was unable to service the request. |

Table 6.6: Fields of `AccessAckData` messages.

# Chapter 7

# 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 signalling restrictions from Chapter 6 are removed. See Sections 4.1 and 4.3 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 4.3. In total there are three request messages and one response message added by TL-UH to the messages defined for TL-UL. Table 7.1 enumerates these messages.

| 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 | | *from* ***TL-UL*** |
| `ArithmeticData` | 2 | Atomic | y | . | `AccessAckData` | |
| `LogicalData` | 3 | Atomic | y | . | `AccessAckData` | |
| `Intent` | 5 | Intent | y | . | `HintAck` | |
| `HintAck` | 2 | Intent | . | y | | *added in* ***TL-UH*** |

Table 7.1: Summary of TL-UH messages.

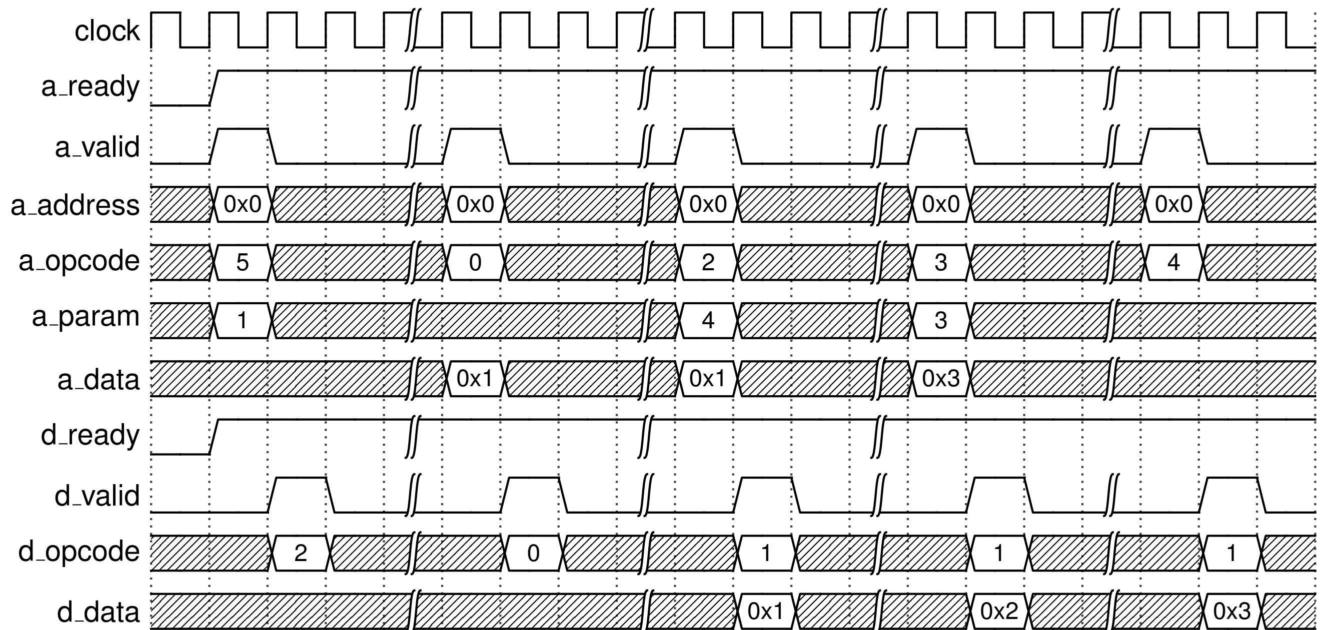


Figure 7.1: Waveform containing Atomic and Hint operations. Prefetch with intent to write; Put storing 0x1; Atomic add of 0x1 returning 0x1; Atomic swap of 0x3 returning 0x2; Get loading 0x3.

## 7.1 Flows and Waves

The figures in this section provide waveforms and message sequence charts for each of the additional TL-UH operations. Figure 7.1 shows a waveform containing both Atomic and Hint operations between a single pair of agents. Figure 7.2 shows the message flow employed by Atomic operations between a single pair of agents. Figure 7.3 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 4.3.

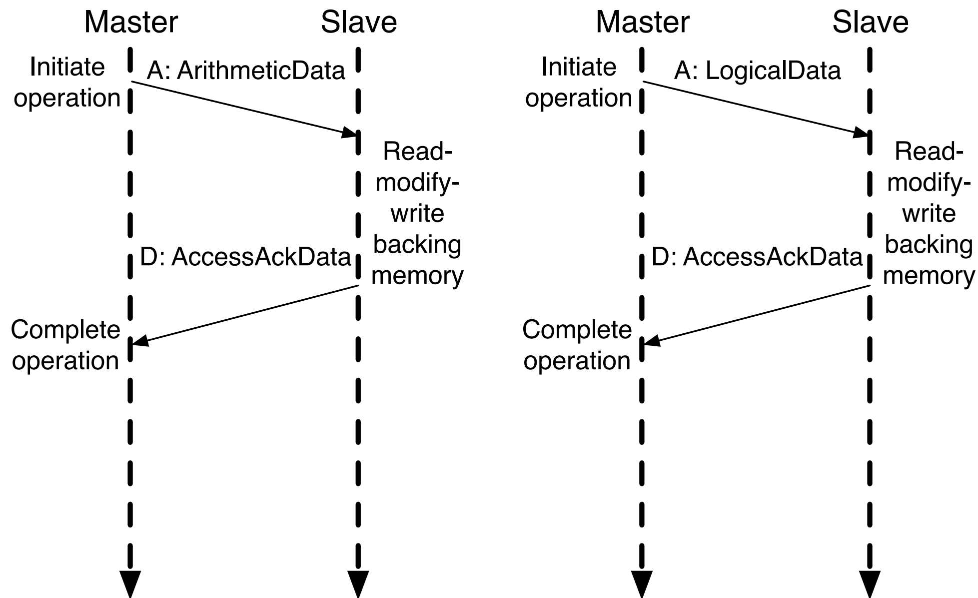


Figure 7.2: Message flow to perform an atomic memory access operation.

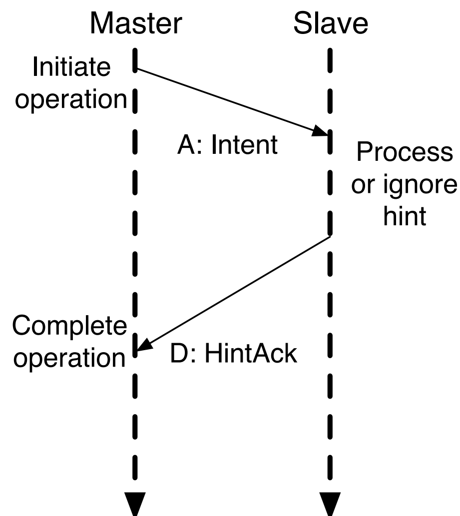


Figure 7.3: Message flow to perform a hint operation.

## 7.2 Messages

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

### 7.2.1 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 7.2 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 7.3. It consists of { `MIN`, `MAX`, `MINU`, `MAXU`, `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 5.4).

`a_address` must be aligned to `a_size`.

`a_mask` selects the byte lanes to read-modify-write (Section 4.6). 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.

| **Channel A** | **Type** | **Width** | **Encoding** |
|---|---|---|---|
| `a_opcode` | C | 3 | Must be `ArithmeticData` (2). |
| `a_param` | C | 3 | See Table 7.3. |
| `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_data` | D | $8w$ | Data payload to be used as operand. |

Table 7.2: Fields of `ArithmeticData` messages.

| **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. |

Table 7.3: `ArithmeticData` `param` field.

### 7.2.2 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 7.4 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 7.5. 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 the 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 5.4).

a_address must be aligned to a_size.

a_mask selects the byte lanes to read-modify-write (Section 4.6). 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.

| Channel A | Type | Width | Encoding |
|---|---|---|---|
| a_opcode | C | 3 | Must be LogicalData (3). |
| a_param | C | 3 | See Table 7.5. |
| 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_data | D | $8w$ | Data payload to be written. |

Table 7.4: Fields of LogicalData messages.

| 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. |

Table 7.5: LogicalData param field.

### 7.2.3 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 7.6 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 7.7. 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 5.4).

a_address must be aligned to a_size.

a_mask indicates the bytes to which the intention applies (Section 4.6). a_size, a_address and a_mask are required to correspond with one another.

a_data is ignored and can take any value.

| A Channel | Type | Width | Encoding |
|---|---|---|---|
| a_opcode | C | 3 | Must be Intent (5). |
| a_param | C | 3 | Intention encoding; See Table 7.7. |
| 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_data | D | $8w$ | Ignored; can be any value. |

Table 7.6: Fields of Intent messages.

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

Table 7.7: Intent messages' param field.

### 7.2.4 HintAck

`HintAck` serves as an acknowledgement message for a Hint operation. Table 7.8 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 5.4).

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

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

`d_error` indicates that an error occured when the slave attempted to perform the operation.

| 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_data` | D | $8w$ | Ignored; can be any value. |
| `d_error` | F | 1 | The slave was unable to service the request. |

Table 7.8: Fields of `HintAck` messages.

## 7.3 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 Tabel 3.2 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. The Final type signals (i.e., `*_error`) can be raised on any beat of the burst, but once raised must remain high for the duration for the burst. The Control type signals (i.e., `*_address`, `*_size`, `*_param`) must be held constant for the entire burst. See Section 4.3 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.

# Chapter 8

# 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** opertations 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 (Chapter 5.3). 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.

## 8.1 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 Read permissions on its copy, which may contain dirty data.

**Tip:** 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.

**Branch:** A node with a cached copy that on the trunk with a read-only copy of the data.

Figure 8.1 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 8.1 describes which access operations can be performed on a node in which state, which is defined raltive to its position in the tree Additional, policy-defined states can be based of off these fundamental states.

| Permissions | Supported Accesses |
|---|---|
| None | None |
| Branch | Get |
| Trunk | Get |
| Tip | Get, PutPartial, PutFull, Logical, Arithmetic |

Table 8.1: Relationships between permissions and access operations.

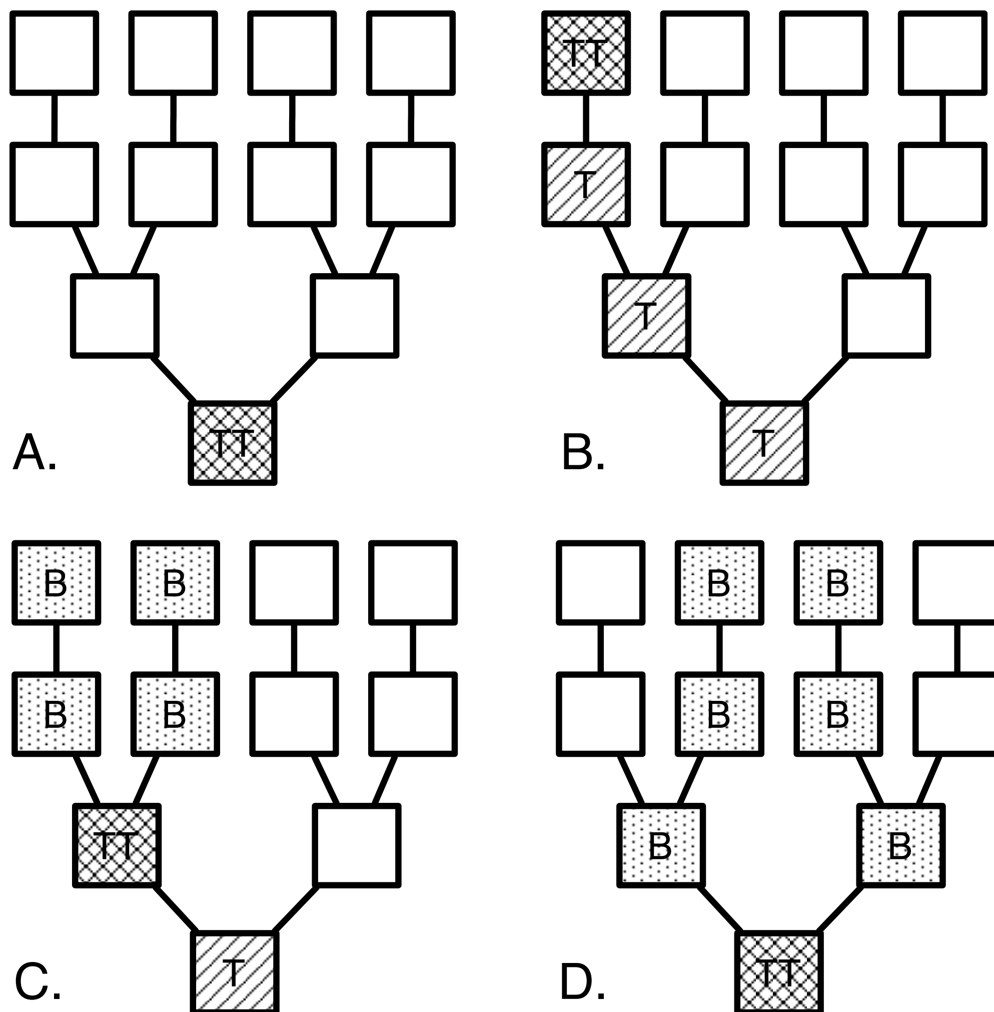


Figure 8.1: Different possible coherence trees overlaid on a single TileLink network graph. *TT* is Trunk Tip, *T* is trunk, *B* is branch. A. The root node has write+read permissions on the only copy. B. A single master has write+read permissions on the trunk tip. C. Multiple masters have read permissions on branches. D. Multiple masters have read permissions on branches, and some branches have been pruned.

### 8.1.1 Operations

The three new operations are termed **transfer** operations (Chapter 5) 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.

Copyright © 2016, SiFive Inc. All rights reserved. 

### 8.1.2 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.

### 8.1.3 Messages

Across the five channels, TL-C specifies ten messages comprising three operations.

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

Table 8.2: Summary of TL-C Permission Transfer Operation Messages.

### 8.1.4 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.

| Category | Contents |
|---|---|
| Permissions | **N**one, **B**ranch, **T**runk |
| Cap | toT, toB, toN |
| Grow | NtoB, NtoT, BtoT |
| Prune | TtoB, TtoN, BtoN |
| Report | TtoT, BtoB, NtoN |

Table 8.3: Categories of permissions transitions.

Table 8.3 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.

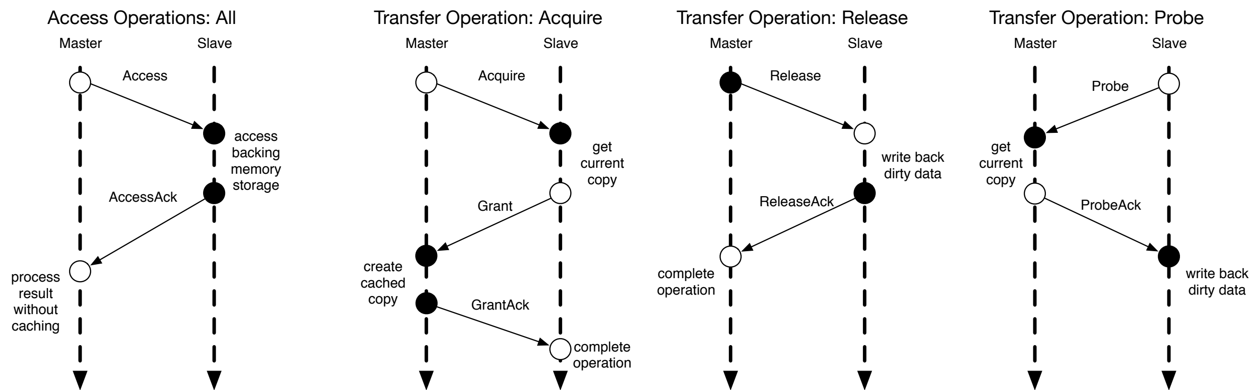


Figure 8.2: Overview of the transaction flows of TileLink operations. Movement of the black dot indicates the the point of transaction serializion has been affected by the operation.

## 8.2 Flows and Waves

Transfer operations introduce new transaction flows which can be composed to form complete cache coherence policy transactions. Figure 8.2 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 8.3 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.

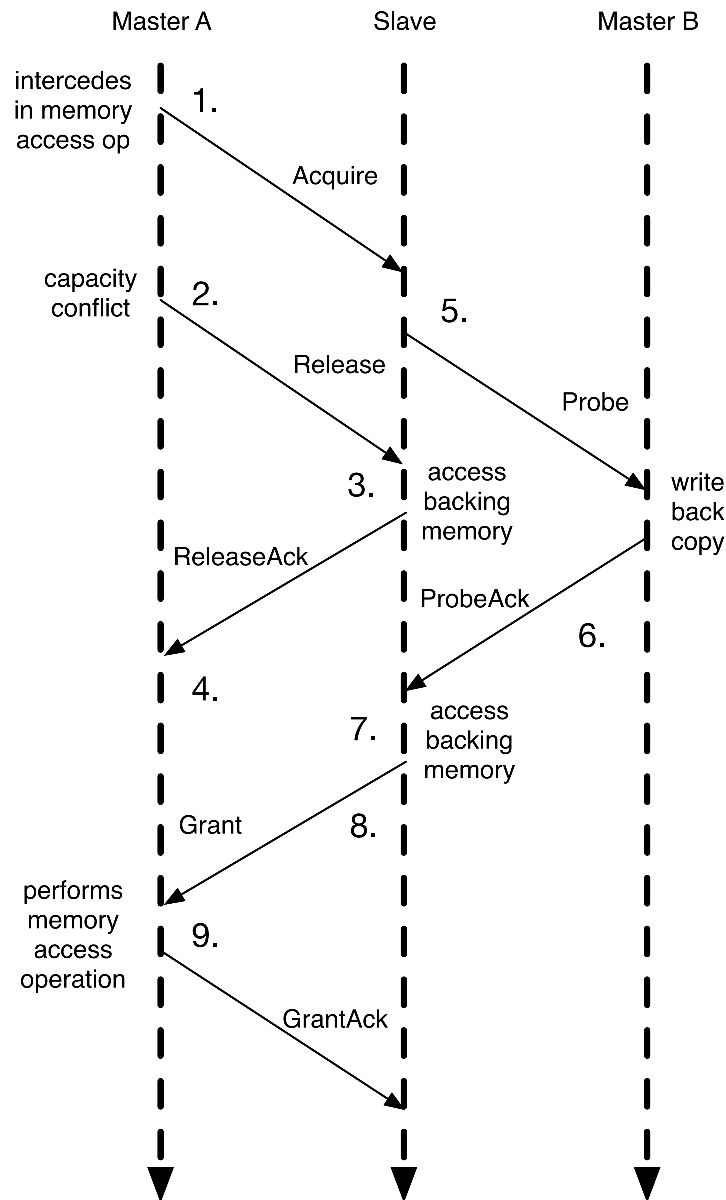


Figure 8.3: Overview of a transaction flow containing all three transfer operations.
1. A caching master sends an `Acquire` to a slave.
2. To make room for the expected response, the same master sends a `Release`.
3. The slave communicates with backing memory if required.
4. The slave acknowledges completion 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.

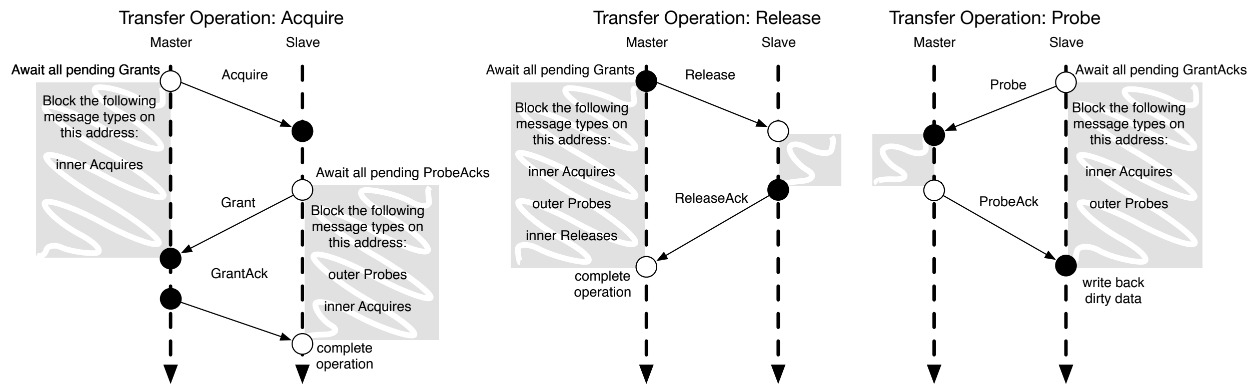


Figure 8.4: Concurrency management rules for transfer operations. These comply with and expand upon the forward progress rules in Section 4.2.2.

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 targetting 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 8.4 provides an overview of the limits put on concurrency for each operation. These rules comply with and expand upon the forward progress rules in Section 4.2.2.

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 8.5 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 `Aquire` 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 serialized 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 8.6 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 `ProbeAcks` resulting from `Probe` messages that have already been sent, and afterwards provide a `ReleaseAck` and `Grant` message to each master before their transactions can be considered complete. The voluntary write's data can be used to respond to the original requestor with a `Grant`, but the transaction cannot complete 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.

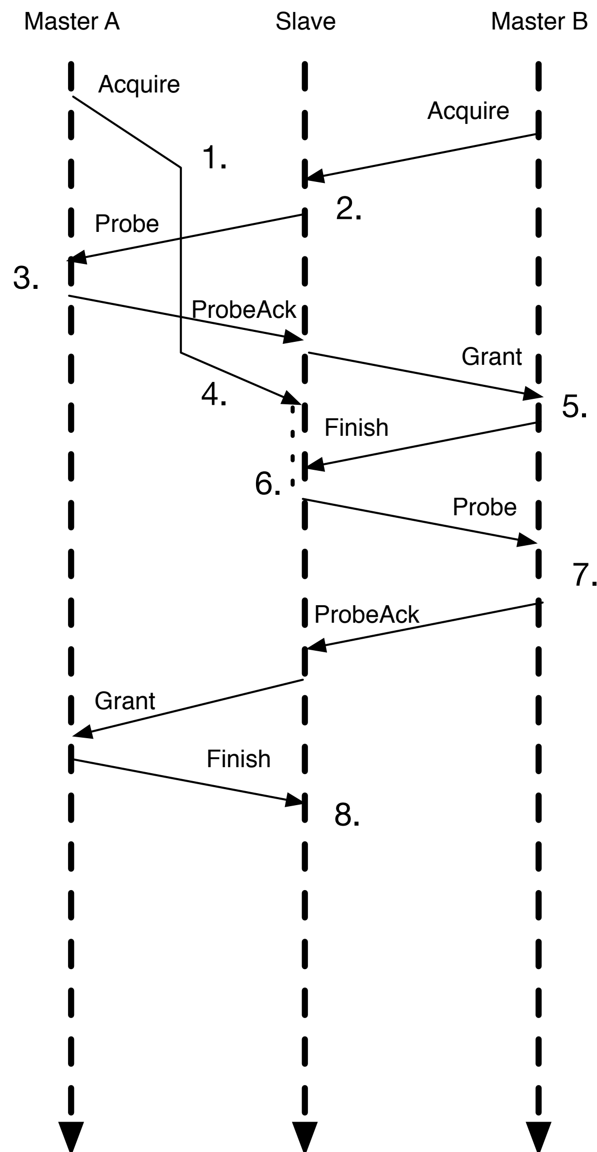


Figure 8.5: Interleaved message flows demonstrating a slave using `GrantAck` to serialize `Grant` and `Probe`.
1. Master A sends an `Acquire` first, but it gets delayed in the network.
2. Master B sends an `Acquire` second, but it arrives at the slave first, and is serialized before A's.
3. The slave 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`.

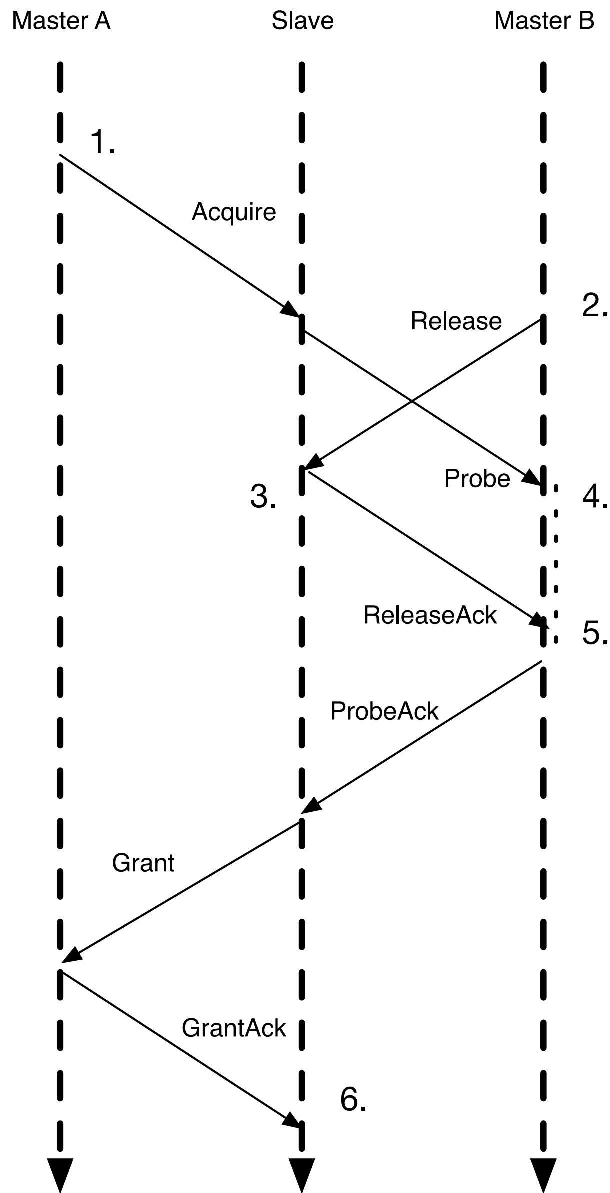


Figure 8.6: Interleaved message flows demonstrating using `ReleaseAck` to serialize `Release` and `Probe`.
1. Master A sends an `Acquire` to a slave.
2. At the same time, Master B chooses to evict the same block and issues a voluntary `Release`.
3. The slave then sends a `Probe` to Master B.
The slave 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.

## 8.3 TL-C Messages

Permissions transfers make use of three additional channels with six new messages, a new Channel 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`.

### 8.3.1 Acquire

An Acquire 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). Like a Get message, an Acquire message does not contain data itself. Table 8.4 shows the encodings used for the fields of Channel 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 8.3.

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 4.6 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.

| Channel A | Type | Width | Encoding |
|---|---|---|---|
| a_opcode | C | 3 | Must be Acquire (6). |
| a_param | C | 3 | Permissions transfer: Grow (NtoB, NtoT, BtoT). |
| a_size | C | $s$ | $2^n$ bytes will be read by the slave and returned. |
| a_source | C | $c$ | 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_data | D | $8w$ | Ignored; can be any value. |

Table 8.4: Fields of Acquire messages on Channel A.

### 8.3.2 Probe

A Probe 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 8.5 shows all the fields of Channel B for this message type.

b_opcode must be Probe, 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 8.3. 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 Probe 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 4.6 for details. b_size, b_address and b_mask are required to correspond with one another. Probe 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 5.4 for details.

| Channel B | Type | Width | Encoding |
|---|---|---|---|
| b_opcode | C | 3 | Must be Probe (6). |
| b_param | C | 3 | Permissions transfer: Cap (toN, toB, toT). |
| b_size | C | $s$ | $2^n$ bytes will be probed by the master and possibly returned. |
| b_source | C | $c$ | 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_data | D | $8w$ | Ignored; can be any value. |

Table 8.5: Fields of Probe messages on Channel B.

### 8.3.3 ProbeAck

A `ProbeAck` message is a response message used by a Master Agent to acknowledge the receipt of a `Probe`. Table 8.6 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 8.3. 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.

`c_error` is reserved and must be set to 0.

| 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 | $s$ | $2^n$ bytes were probed; copied from `b_size`. |
| `c_source` | C | $c$ | 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_data` | D | $8w$ | Ignored; can be any value. |
| `c_error` | F | 1 | Reserved; must be 0. |

Table 8.6: Fields of `ProbeAck` messages on Channel C.

### 8.3.4 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 8.7 shows all the fields of Channel 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 8.3. 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. Data can be changed between beats of a `ProbeAckData` that is a burst.

`c_error` indicates that an error occured when the master attempted to process the memory operation. The error flag can be raised on the final beat of a burst.

| 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 | $s$ | $2^n$ bytes were probed and are being written back; copied from `b_size`. |
| `c_source` | C | $c$ | 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_data` | D | $8w$ | The data payload. |
| `c_error` | F | 1 | The master was unable to service the request. |

Table 8.7: Fields of `ProbeAckData` messages on Channel C.

### 8.3.5 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 8.8 shows the encodings used for fields of Channel 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 8.3. 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 Channel 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 5.4 for details.

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

`d_error` is reserved and must be 0.

| **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 | $s$ | $2^n$ bytes were accessed by the slave; copied from `a_size`. |
| `d_source` | C | $c$ | The master source identifier receiving this response; copied from `a_source`. |
| `d_sink` | C | $m$ | The slave sink identifier issuing this request. |
| `d_data` | D | $8w$ | Ignored; can be any value. |
| `d_error` | F | 1 | Reserved; must be 0. |

Table 8.8: Fields of `Grant` messages on Channel D.

### 8.3.6 GrantData

A GrantData message is a 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 8.9 shows the encodings used for fields of the Channel 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 8.3. 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 Channel 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 5.4 for details.

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

d_error indicates that an error occured when the slave attempted to process the transfer operation. In this case, d_param should be ignored, meaning the coherence policy permissions of the block remain unchanged.

| 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 | $s$ | $2^n$ bytes are being transferred by the slave; copied from a_size. |
| d_source | C | $c$ | The master source identifier receiving this response; copied from a_source. |
| d_sink | C | $m$ | The slave sink identifier issuing this response. |
| d_data | D | $8w$ | The data payload. |
| d_error | F | 1 | The slave was unable to service the request. |

Table 8.9: Fields of GrantData messages on Channel D.

### 8.3.7 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 8.10 shows all the fields of this message on Channel 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.

| **Channel E** | **Type** | **Width** | **Encoding** |
|---|---|---|---|
| `e_sink` | C | $m$ | The slave sink identifier accepting this response; copied from `d_sink`. |

Table 8.10: Fields of `GrantAck` messages.

### 8.3.8 Release

A `Release` message is a request message used by a Master Agent to voluntarily downgrade its permissions on a cached data block. Table 8.11 shows all the fields of Channel 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** category of Table 8.3, 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_error` is reserved and should be set to 0.

| Channel C | Type | Width | Encoding |
|---|---|---|---|
| `c_opcode` | C | 3 | Must be Release (5). |
| `c_param` | C | 3 | Permissions transfer: Shrink or Report (`TtoB`, `TtoN`, `BtoN`, `TtoT`, `BtoB`, `NtoN`). |
| `c_size` | C | $s$ | $2^n$ bytes are being downgraded by the master. |
| `c_source` | C | $c$ | 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. |
| `c_error` | F | 1 | Reserved; must be 0. |

Table 8.11: Fields of `Release` messages on Channel C.

### 8.3.9 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 8.12 shows all the fields of Channel 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** category of Table 8.3, 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. Data can be changed between beats of a `ReleaseData` that is a burst.

`c_error` indicates that an error occured while attempting to process the memory operation. This flag can be used to indicate memory corruption of data being evicted from a cache. The error flag should be raised on the final beat of a burst.

| Channel C | Type | Width | Encoding |
|---|---|---|---|
| c_opcode | C | 3 | Must be ReleaseData (6). |
| c_param | C | 3 | Permissions transfer: Shrink (TtoB, TtoN, BtoN, TtoT, BtoB, NtoN). |
| c_size | C | $s$ | $2^n$ bytes are being written back by the master. |
| c_source | C | $c$ | The master source identifier of this response. |
| c_address | C | $a$ | The target address of the Transfer, in bytes. |
| c_data | D | $8w$ | The data payload. |
| c_error | F | 1 | The master was unable to write back this data. |

Table 8.12: Fields of `ReleaseData` messages on Channel C.

### 8.3.10 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 8.13 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 inflight. See Section 5.4 for details.

d_data is ignored and can be assigned any value.

d_error indicates that an error occured when the slave attempting to process the memory operation.

| **Channel D** | **Type** | **Width** | **Encoding** |
|---|---|---|---|
| d_opcode | C | 3 | Must be ReleaseAck (6). |
| d_param | C | 2 | Reserved; must be 0. |
| d_size | C | $s$ | Bytes transferred; copied from c_size. |
| d_source | C | $c$ | The master source identifier receiving this response; copied from c_source. |
| d_sink | C | $m$ | Ignored; can be any value. |
| d_data | D | $8w$ | Ignored; can be any value. |
| d_error | F | 1 | The slave was unable to service the request. |

Table 8.13: Fields of ReleaseAck messages on Channel D.

## 8.4 TL-UL and TL-UH messages on Channel B and Channel 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.

### 8.4.1 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 8.14 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 4.6 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 Chapter 5.4 for details.

b_data is ignored and may take any value.

| Channel B | Type | Width | Encoding |
|---|---|---|---|
| b_opcode | C | 3 | Must be Get (4). |
| b_param | C | 3 | Reserved; must be 0. |
| b_size | C | $s$ | $2^n$ bytes will be read by the master and returned. |
| b_source | C | $c$ | 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_data | D | $8w$ | Ignored; can be any value. |

Table 8.14: Fields of Get messages on Channel B.

### 8.4.2 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 8.15 shows the encodings used for the fields of the Channel 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 4.6 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.

| Channel B | Type | Width | Encoding |
|---|---|---|---|
| b_opcode | C | 3 | Must be PutFull (0). |
| b_param | C | 3 | Reserved; must be 0. |
| b_size | C | $s$ | $2^n$ bytes will be written by the master. |
| b_source | C | $c$ | 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_data | D | $8w$ | Data payload to be written. |

Table 8.15: Fields of PutFullData messages on Channel B.

### 8.4.3 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 8.16 shows the encodings used for the fields of the Channel 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 posibly 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 4.6 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 interface 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.

| **Channel B** | **Type** | **Width** | **Encoding** |
|---|---|---|---|
| b_opcode | C | 3 | Must be PutFull (1). |
| b_param | C | 3 | Reserved; must be 0. |
| b_size | C | $s$ | Up to $2^n$ bytes will be written by the master. |
| b_source | C | $c$ | 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_data | D | $8w$ | Data payload to be written. |

Table 8.16: Fields of PutPartialData messages on Channel B.

### 8.4.4 AccessAck

`AccessAck` provides an dataless acknowledgement to the original requesting agent. Table 8.17 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 Chapter 5.4 for details.

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

`c_error` indicates that an error occured when the master attempted to process the memory operation.

| **Channel C** | **Type** | **Width** | **Encoding** |
|---|---|---|---|
| `c_opcode` | C | 3 | Must be AccessAck (0). |
| `c_param` | C | 3 | Reserved; must be 0. |
| `c_size` | C | $s$ | $2^n$ bytes were accessed by the master. |
| `c_source` | C | $c$ | The master source identifier issuing this response. |
| `c_address` | C | $a$ | The target address of the operation, in bytes. |
| `c_data` | D | $8w$ | Ignored; can be any value. |
| `c_error` | F | 1 | The master was unable to service the request. |

Table 8.17: Fields of `AccessAck` messages on Channel C.

### 8.4.5 AccessAckData

AccessAckData provides an acknowledgement with data to the original requesting agent. Table 8.18 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 assosciated 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 Chapter 5.4 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_error indicates that an error occured when the master attempted to process the memory operation.

| Channel C | Type | Width | Encoding |
|---|---|---|---|
| c_opcode | C | 3 | Must be AccessAckData (1). |
| c_param | C | 3 | Reserved; must be 0. |
| c_size | C | $s$ | $2^n$ bytes were accessed by the master. |
| c_source | C | $c$ | The master source identifier issuing this response. |
| c_address | C | $a$ | The target address of the Access, in bytes. |
| c_data | D | $8w$ | The data payload for messages with data. |
| c_error | F | 1 | The master was unable to service the request. |

Table 8.18: Fields of AccessAckData messages on Channel C.

### 8.4.6 ArithmeticData

A 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 8.19 shows the encodings used for the fields of the Channel 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 7.3. 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 4.6 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 interface 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.

| Channel B | Type | Width | Encoding |
|---|---|---|---|
| b_opcode | C | 3 | Must be ArithmeticData (3). |
| b_param | C | 3 | See Table 7.3. |
| b_size | C | $s$ | $2^n$ bytes will be read and written by the master. |
| b_source | C | $c$ | 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_data | D | $8w$ | Data payload to be used as operand. |

Table 8.19: Fields of ArithmeticData messages on Channel B.

### 8.4.7 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 8.20 shows the encodings used for the fields of the Channel 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 7.5. 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 the this request's data as well as the `AccessAckData` response.

`b_address` must be aligned to `b_size`. See Section 4.6 for details.

`b_mask` provides the byte select lanes, in this case indicating which bytes to read-modify-write. See Section 4.6 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 interface 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.

| **Channel B** | **Type** | **Width** | **Encoding** |
|---|---|---|---|
| `b_opcode` | C | 3 | Must be LogicalData (3). |
| `b_param` | C | 3 | See Table 7.5. |
| `b_size` | C | $s$ | $2^n$ bytes will be read and written by the master. |
| `b_source` | C | $c$ | 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_data` | D | $8w$ | Data payload to be written. |

Table 8.20: Fields of `LogicalData` messages on Channel B.

### 8.4.8 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 8.21 shows the encodings used for the fields of the Channel 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 7.7. 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 4.6 for details. b_size, b_address and b_mask are required to correspond with one another.

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

b_data is ignored and can take any value.

| Channel B | Type | Width | Encoding |
|---|---|---|---|
| b_opcode | C | 3 | Must be Intent (5). |
| b_param | C | 3 | Intention encoding; See Table 7.7. |
| b_size | C | $s$ | $2^n$ bytes to which this intention applies. |
| b_source | C | $c$ | 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_data | D | $8w$ | Ignored; can be any value. |

Table 8.21: Fields of Intent messages on Channel B.

### 8.4.9 HintAck

`HintAck` serves as an acknowledgement response for a Hint operation. Table 8.22 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 Chapter 5.4 for details.

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

`c_error` indicates that an error occured when the master attempted to process the memory operation.

| **Channel C** | **Type** | **Width** | **Encoding** |
|---|---|---|---|
| `c_opcode` | C | 3 | Must be `HintAck` (2). |
| `c_param` | C | 3 | Reserved; must be 0. |
| `c_size` | C | $s$ | $2^n$ bytes were hinted about. |
| `c_source` | C | $c$ | The master source identifier issuing this response. |
| `c_address` | C | $a$ | The target address of the operation, in bytes. |
| `c_data` | D | $8w$ | Ignored; can be any value. |
| `c_error` | F | 1 | The master was unable to service the request.. |

Table 8.22: Fields of `HintAck` messages on Channel C.

# Glossary

**Access** An operation that reads and/or writes the data at a specified address. 33, 34

**acknowledgement message** a message the other agent is required to send back if you send a request. 4, 50, 51, 53, 61

**Acquire** A Transfer operation whereby the master acquires permissions to cache a copy of the block from the slave. 33, 35, 36

**agent** An active participant in the protocol that sends and receives messages in order to complete operations. 3, 4

**Atomic** An Access operation allowing the master to read-modify-write addresses managed by the slave. 27, 33, 35, 36, 43, 53, 54, 62

**beat** A single-clock-cycle slice of any message that takes multiple cycles to transmit over a channel of a particular width. 17, 20, 24–27, 62

**burst** A multi-beat message. iv, 17, 18, 24, 26, 43, 53, 62

**channel** A one-way communication link between a master interface and a slave interface carrying messages of homogeneous priority. 3, 6, 9

**Channel A** Transmits a request that an operation be performed at a specified address, accessing or caching the data. iii, 6, 12, 15, 19, 25, 26, 47–49, 58–60, 76, 77, 81, 82

**Channel B** Transmits a request that an operation be performed at a specified master, accessing or un-caching the data. iii, v, 6, 13, 14, 39, 78, 87–90, 93–95

**Channel C** Transmits a data or permissions acknowledgment for a Channel B request. iii, v, 6, 14, 15, 39, 79, 80, 84, 85, 87, 91, 92, 96

**Channel D** Transmits a data or permissions acknowledgement to the original requestor. iii, 6, 15, 16, 24–26, 39, 40, 50, 51, 61, 76, 81, 82, 86

**Channel E** Transmits a final acknowledgment of a cache block transfer from the requestor, used for serialization. iii, 6, 16, 81–83

**DAG** Directed Acyclic Graph. 4–6, 20, 23

**deadlock** . 6, 25

**follow-up message** any message sent as a result of receiving some other message. 20

**forwarded message** a recursive message that is at the same level of priority as the message that initiated it. 20

**Get** An Access operation allowing the master to read addresses managed by the slave. 2, 33–36, 43–45, 53, 54, 62

**Hint** An operation that is informational only and has no direct effect on data values. 33, 34, 53, 60, 61, 95, 96

**Intent** A Hint operation that indicates the master intends to read or write data at addresses managed by the slave. 33, 35, 36, 54

**link** The set of channels required to complete operations between two agents. 3, 4, 6, 11

**master interface** Through which agents may request that memory operations be performed, or for permission to cache copies of data. 4, 5, 11, 17, 24–26, 33, 90, 93–95

**message** A set of control and data values sent over a particular channel. iv, v, 3, 17, 33–36, 43, 54, 62, 87

**MOESI** A cache coherence policy featuring an ownership state. 1

**operation** A change to an address range's data values, permissions or location in the memory hierarchy. iv, 3, 20, 33, 34

**Probe** A Transfer operation whereby the slave revokes permissions to cache a copy of the block from the master. 35

**Put** An Access operation allowing the master to write addresses managed by the slave. 2, 33–36, 43–45, 53, 54, 62

**receiver** The interface that accepts messages on a channel (raises ready). 17

**recursive message** optional messages sent as a means of implementing an operation. 20, 21, 23

**Release** A Transfer operation whereby the master voluntarily releases permissions on the block back to the slave. 33, 35, 36

**request message** a message specifying an access to perform or a change of permissions on a cached block. 4, 9, 20, 21, 24–27, 43

**response message** a message the other agent is required to send back if you send a request. 9, 20, 24–27, 39, 43, 47, 54

**sender** The interface that originates messages on a channel (raises valid). 17

**slave interface** Through which agents may grant permissions and access to a range of addresses, and respond with completed memory operations. 4, 5, 11, 17, 24, 60, 95

**SoC** System-on-Chip. 1

**TL-C** TileLink Cached. 2, 6, 9, 13, 14, 16, 33, 50, 51, 63, 91, 92

**TL-UH** TileLink Uncached Heavyweight. 2, 33, 53, 54

**TL-UL** TileLink Uncached Lightweight. 2, 24, 33, 43, 44, 47–51, 53, 54

**Transfer** An operation that moves permissions or cached copies of data through the network.. 33