MIPS® Coherence Protocol Specification

Document Number: MD00605
Revision 01.01
September 14, 2015

Template: nB1.03, Built with tags: 2B

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This document describes a cache coherence *architecture*. The flexible and modular nature of the architecture allows for a wide range of implementations within the coherent multiprocessor design space.

1.1 The Usefulness of Coherent Shared Memory Systems

Multiple CPU systems are appropriate solutions for a broad class of embedded applications. There are broadly two programming models for multiprocessor systems, *shared memory* and *message passing*. Each model has its advantages and disadvantages.

A message passing model requires less complex hardware support, but is more constrained in the way it allows concurrency to be exploited.

A shared memory model is more general, but more complex to implement and more subtle to exploit efficiently and correctly in software. The programming semantics of a shared memory model requires user-transparent transfer of data between processors and memories and to provide the view of an apparent single-user-at-a-time semantics. If caches are present in the system, there is a need for a coherence protocol imposed on caches and cores to keep their views of storage coherent.

1.2 The Need for a Coherence Architecture

The need for a coherence architecture comes from the need for a technology infrastructure that can be shared and leveraged across multiple product families and generations.

If an implementation adheres to the architecture, it should be possible to re-use common coherency management logic across multiple processor core families and across multiple interconnect families.

It will also allow for the use of reference simulators and models to aid in designing a range of implementations.

1.3 Components of the Architecture

This architecture is composed of :

1. A set of abstract models of components necessary within a multiprocessor system. This set of models is called the Coherent Framework. This is discribed in Chapter 2, “Coherence Framework” on page 11.

2. A snoopy coherence protocol which determines how coherent caches interact with each other to maintain a shared view of memory. This protocol is suitable for small to medium sized (4-8 processors) multiprocessor systems. This is described in Chapter 3, “Snoopy Coherence Protocol” on page 21.

3. A memory ordering and consistency model which determines how memory transactions from multiple coherent masters are made visible to the rest of the system. This is described in Chapter 4, “Memory Consistency, Ordering and Synchronization” on page 33.

1.4 Intended Audience

This document is meant to be read by system architects building coherent systems; CPU (micro)-architects building coherent CPUs, kernel-level software developers and low-level device driver writers for coherent systems.

1.5 Document Conventions

```
Code and pseudo-code examples use this font and are indented.

Comments in the code are prefaced with the // symbol and use this grey color.
```

Chapter 2

Coherence Framework

This chapter serves as an introduction to a coherence framework on which various directory and snoopy coherence protocols could be built. The next chapter describes such a protocol.

2.1 Introduction to the Framework

The coherence framework is composed of abstract models of master devices, slave devices and system interconnect within a coherent system.

The framework only assumes that the class of coherence protocols will be invalidation-based, and does *not* restrict:

- The type of interconnect used. As an example it does not restrict the implementor to use a broadcast capable interconnect like a bus as opposed to a more scalable but potentially higher latency interconnect with only point to point ordering properties.
- The protocol states used. Instead of using the familiar MESI schemes an implementor may choose to support the dirty shared state (O), leading to a MOESI coherence protocol if interchip sharing is a significant factor in the target workloads.
- Protocol specific Optimizations: Cache to cache sharing, Snarfing, Forwarding, 3-hop forwarding, migration, predictive broadcast (for low power), could be implemented within the framework.
- The use of heterogeneous Agents: Allows DSPs and graphics/network/I/O processors to share memory with a CPU.

2.2 Definitions

The framework assumes a set of *agents*, which are processors, memories, and peripherals, tied together by an *interconnect*. Agents may be *masters* or *slaves* to the interconnect. Processors are generally master agents. Memories are slave agents. Peripherals may be slave agents, master agents (in the case of DMA), or both. A *coherent agent* is one which contains or controls some amount of cached data or metadata, and which participates in the coherence framework. The framework defines an *abstract master model*, an *abstract slave model*, and an *abstract interconnect model*.

Each agent has one or more *ports*, which are interfaces to an interconnect. A *master port* is a port capable of sending *requests* into the interconnect, and receiving *responses*. A *slave port* is a port capable of receiving requests from the interconnect, and sending responses. A response is always associated with a request, but not all requests necessarily have an associated response.

One simple illustration of this model is a processor-memory bus, onto which the CPU agent has a master port, and a memory controller agent has a slave port onto the bus. A read request by the CPU becomes a request from the master port, which translates to a memory read request cycle on the bus. The memory controller's bus interface picks up the read request cycle as an incoming request on a slave port, performs the desired memory reference, and sends the

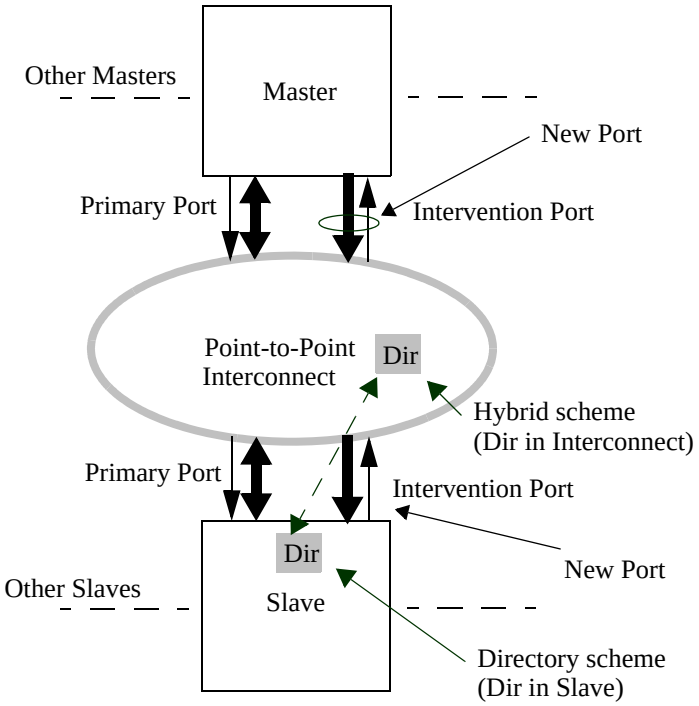
memory data as a response on the slave port, which translates to a read data response cycle on the bus. That bus cycle is picked up by the CPU as a response on the master port, which completes the load operation.

Depending on the nature of the interconnect, the same physical interface may support both a master and a slave port, or the port functions may be split across two distinct physical interfaces.

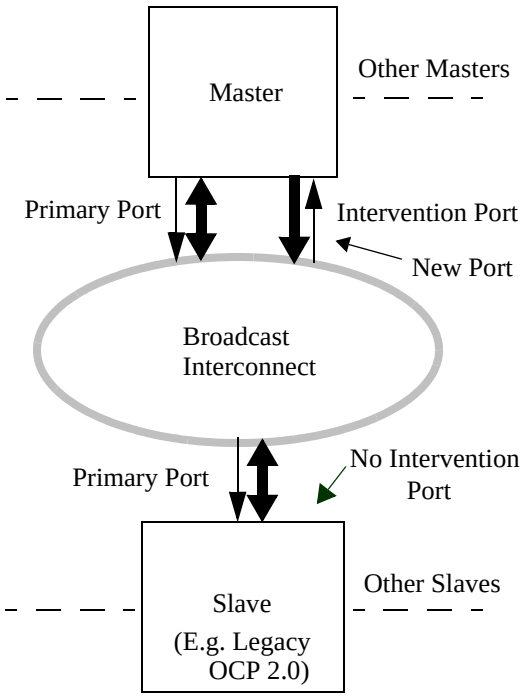
Coherent agents which have associated cache state must have an *intervention port* which is logically independent of the *primary* master/slave ports, and which is reserved for cache coherence intervention traffic. The intervention port is complimentary to the primary port: e.g. for a processor, where the primary port is a master port, the intervention port is a slave port.

A diagram of a couple of coherent system models implementing the framework is shown in Figure 2.1 below. On the left is shown a directory-based coherent system with a directory resident in the slave agent. Serialization is done in the slave agent. Another implementation of the framework is shown on the right. This is a broadcast-based, snoopy coherent system with a legacy memory controller as the slave agent. The new ports are shown as well as the 3 main entities: the master and slaves and the interconnect, which are described in the next sections.

Figure 2.1 Coherent System Implementations/Models.



(Directory scheme with a **Coherent** Memory slave)



(Snoopy scheme with a **Legacy** Memory slave)

The MIPS coherent multiprocessor framework associates coherency information with each request and response, and introduces a set of new requests and responses that are specific to maintaining cache coherence.

The unit of coherency is a *cacheline*, which corresponds to a consecutive set of one or more word addresses in a shared address space. Every word address falls within exactly one cacheline. At least one instance of each line is present in a system, typically in main memory, but multiple instances are possible. The coherency framework protocols guarantee that all agents will see the same values for a given location, regardless of the distribution of instances of the associated line across the system.

Each coherent agent sees each line of storage as being in one of 6 *coherency states*, as called out in Table 2.1. Not all agents will associate the same state with a given line at a given point in time. A line that is only being referenced by one coherent agent may be *Modified* or *Exclusive* with respect to that agent, but *Invalid* with respect to all other coherent masters.

Table 2.1 Possible Line Coherency States

Name	Mnemonic	Description	Compliance
Invalid	I	Cache line not present	Required
Shared	S	Cache line in more than one agent with read only capability	Required
Modified	M	Only cache line in system with the latest data	Required
Exclusive	E	Exclusive cached copy	Required
Owned	O	Cached in more than 1 agent and this agent has the latest copy while memory has a stale copy	Optional
Migratory	T	Indicates to the requestor that the cache line is migratory	Optional

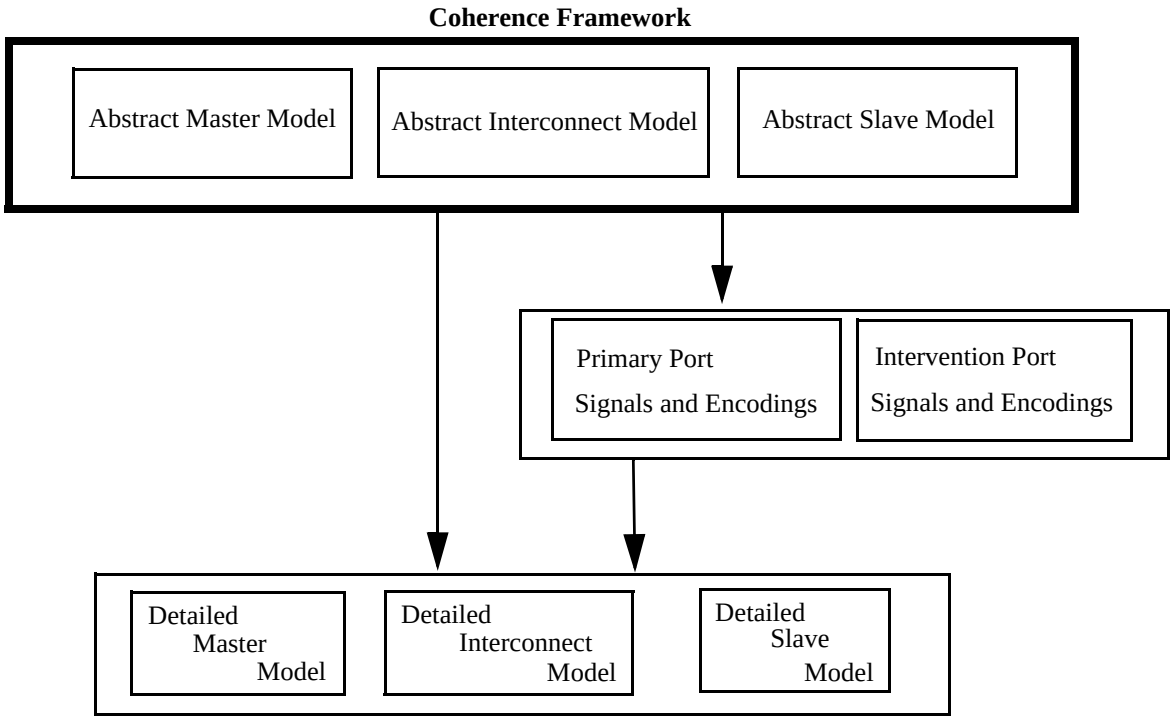
A *coherence domain* is a set of agents which have a common, coherent view of a range of storage addresses.

2.3 Abstract Models as a guide for Implementations

By defining an abstract model of the coherent system with very specific interfaces and transactions the framework bridges two generally opposing requirements, those of *versatility* and *specificity*. The model is abstract enough to allow varied protocols, from snoopy to optimized directory based schemes . The model is specific enough in its interfaces to be modular and allow re-use of the differing parts of a coherent system.

Figure 2.2 diagrams the relationships between the abstract models of the coherence framework and a coherent system implementation. The detailed master, slave, and interconnect models of the system are instantiations of the abstract models which map to the specific interconnect and protocol used.

Figure 2.2 Coherence framework and Implementations.



A coherent implementation built on this framework.

2.4 Coherent Master Model

2.4.1 Intervention

In a non-coherent system, when a master requests for data, it looks up in its local cache hiearchy (if one exists) to see whether the requested address is resident or not. If the address is not resident or not valid within the local cache hiearchy, then the master will start a memory request for that address.

In a coherent system, when the master makes a request for a coherent address, it is not sufficient to just look up in the local cache hierarchy as another coherent master might hold the most up to date version of the data. To always receive the up-to-date version of the data, all coherent caches in the system must be looked up for the requested address. The look up results from all of the coherent caches must be first collected before the next appropriate step is taken to deliver the requested data.

In the coherence framework, the term *Intervention* is used for the request to do a cache look up that occurs within a non-local cache that is caused by the local master requesting a coherent address. Similarly, the term is also used for a cache lookup request within the local cache that is caused by another master.

2.4.2 Self-Intervention

Self-intervention in the coherence framework is provided to resolve races which occur between the issue of coherence requests from a master agent against potentially conflicting incoming coherence requests without causing a potential

deadlock. Every coherent request issued by a master agent is returned back to the master's intervention port, to indicate where in the global order this particular request was placed. This is useful for both directory and snoopy schemes.

A self-intervention is a marker which is useful in resolving races in deeply pipelined request pipelines, and in cache hierarchies. It can also be thought of providing a *logical* ordering capability even though the *physical* network is a distributed, but ordered network like a ring, or a tree etc.

Self-intervention requests are special cases of intervention requests, and must be distinguished as such on their arrival at the intervention port. Whether this is done with an explicit interface signal, or derived from other information in the request (e.g. source/destination agent identifiers), is implementation-specific.

2.4.3 Master Model Description

A diagram of the master model in the framework with associated dependencies is shown in Figure 2.3 below.

- Processor requests are issued at the processor interface. The Processor looks up the *cache state* and generate a coherence transaction request to memory via the primary port. No state in the cache is changed at this point.

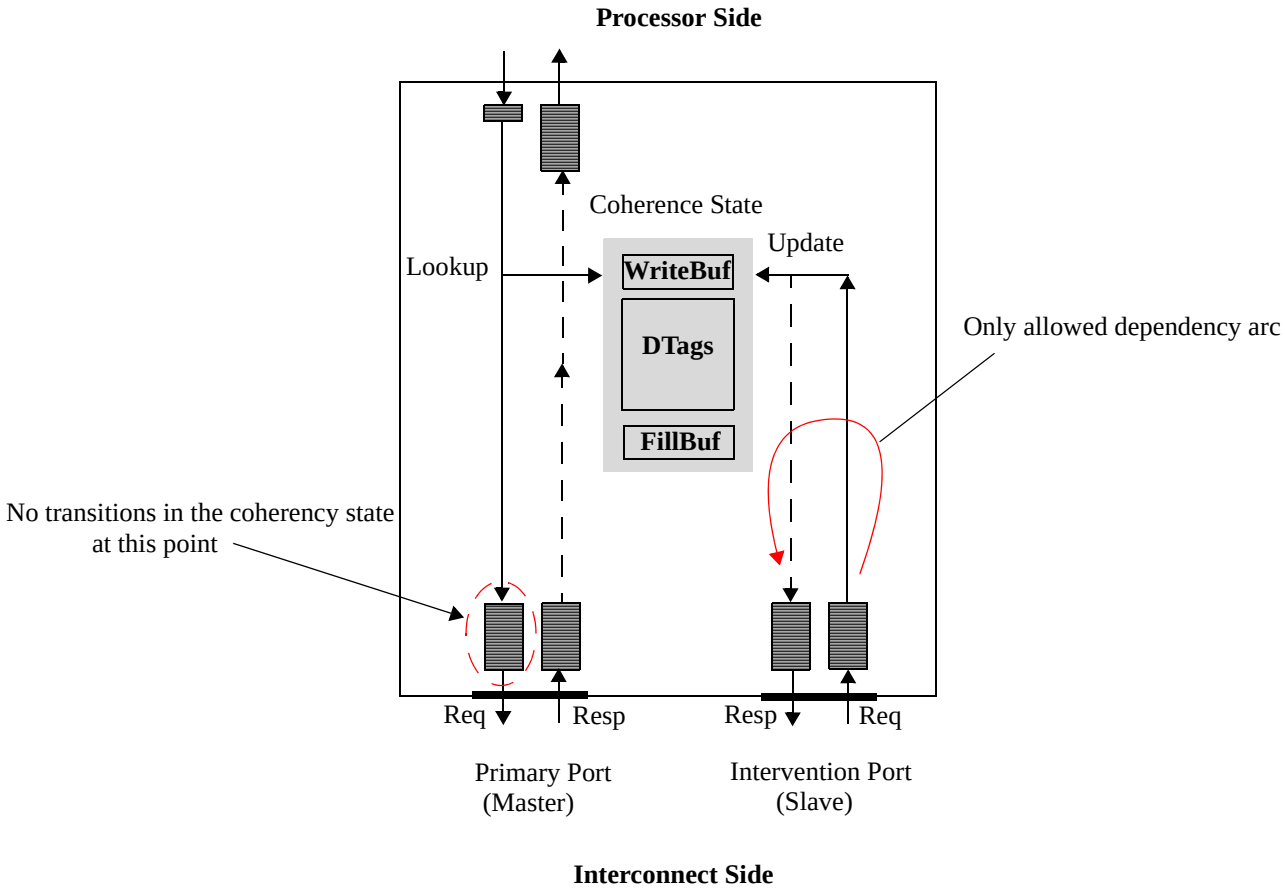
- Next, a self-intervention request is expected on the processors intervention port which causes the cache state to be updated and acknowledged in an intervention response. The only dependency arc is shown on the right side of Figure 2.3.

- Finally, the conflicts left during the interval from the self-intervention to the primary port response due to conflicting requests on the intervention port need to be resolved by the coherence protocol. The intervention port is assumed to be drained in a FIFO fashion in the basic model.

A *basic* implementation needs to accept intervention requests in a FIFO manner, such that all internal state transitions due to the intervention request on the intervention port are committed *before* the response to the intervention request is sent out. The primary port needs to also resolve all outstanding responses when and if it encounters a *conflicting* intervention request on the intervention port. A request conflicts with another if they address the same cache line, their order of execution matters, and at least one of them is a store. Higher performance implementations are possible, but they need to be reducible, systematically to this basic model. As an example, given a weak memory consistency model, one could use tagged, out-of-order intervention requests and responses rather than FIFO ordering.

For lower cost implementations the primary port and the intervention ports can be multiplexed onto the same physical interface. For even more complex implementations tags could be used for both the main and the intervention ports within each thread. All of these modifications can still be proven deadlock free as long as the implementation can be reduced systematically to this basic model.

Figure 2.3 Master Model



2.5 Coherent Slave Model

A diagram of a coherent slave model in the framework with the associated dependencies is shown in Figure 2.4 below. The slave model incorporates both snoopy and directory-based coherence schemes.

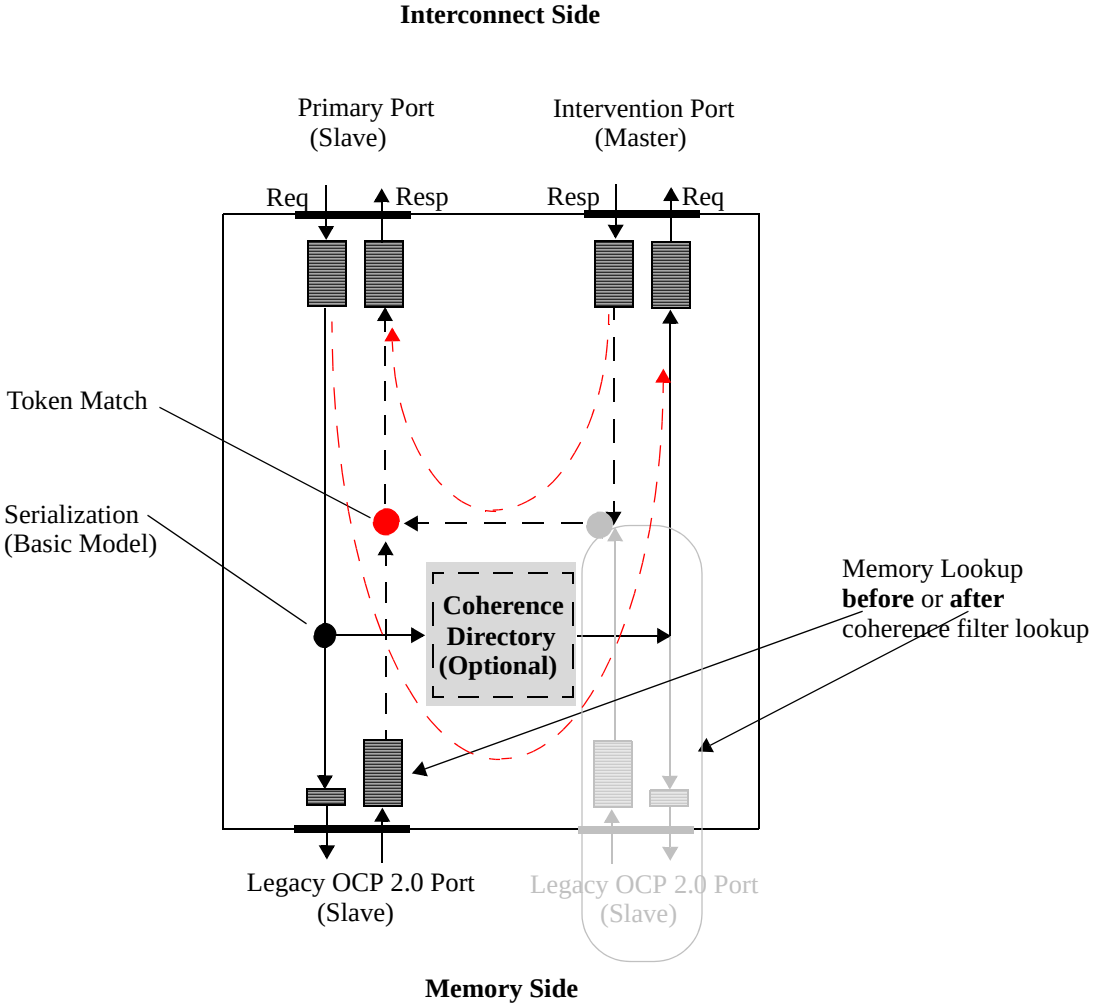
- Requests appear at the primary slave port already serialized by the interconnect in snoopy schemes. They are serialized in the slave in directory-based schemes. For the basic policy it is assumed that the directory, if present, is checked in parallel with the main memory. The intervention port is a master port on the slave agent. Its behavior varies between snoopy and directory schemes as follows.

- In the case of a *directory scheme*, after the lookup in the directory an intervention request may be sent to the implicated agent(s). An intervention response is then expected back on the intervention port. This response is matched (“Token Match” in the diagram) with the original request and a response sent back to the original initiator though the primary port. Also shown, shaded, is the variant policy where a memory access is initiated after the directory is looked up. This would be the policy if the directory lookup was very quick and the implementor wanted to avoid a speculative lookup of main memory. It is also possible in a directory scheme for the directory to be embedded in the interconnect, in which case the phases just described are still valid except for the fact that the directory is accessed via the interconnect, potentially leading to a simpler memory slave.

- In the case of the *snoopy scheme*, the directory which acts as a filter is not present in the slave. The serialization is still done in the interconnect. The master intervention port does not send any transactions out as they are sent directly from the interconnect. *In fact one could use a legacy memory slave without any master intervention port in a snoopy coherent system without modification*.

The two allowed dependency arcs between the intervention and primary port are shown in the figure below. All queues are assumed to be drained in a FIFO fashion in the basic model. This basic model assumes in-order intervention and then response. It is possible that for complex implementations tags could be used for the intervention ports in order to provide out-of-order operation. It is also possible for a higher latency implementation to provide a primary port response after the directory check, but before the intervention response. This reduces the serialization latency due to the intervention phase.

Figure 2.4 Slave Model



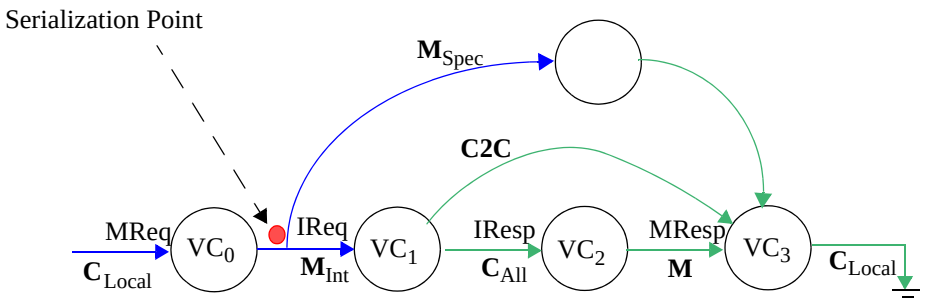
2.6 Coherent Interconnect Model

The design of the interconnect must ensure that the coherence protocol is deadlock-free. To this end, the communication can be modelled as a *Static Dependence Graph* with the interconnect modelled as virtual channels, each of which provides independently flow-controlled, in-order, unidirectional communication between two endpoints. In this graph

an arc from node *x* to node *y (x->y)* indicates the dependence of the resource *x* on the availability of the resource *y*. If these graphs are free of any cycles, then the system is provably deadlock free. Such a dependence graph is given in Figure 2.5 below.

The figure shows the (static) dependence graph for the basic coherence model including the interconnect. Four independent (one-way) channels with 2 channels for the primary port and 2 channels for the intervention port are assumed. Memory requests from a cache are sent through VC0 between the cache agent's primary port and the coherent memory subsystem. These requests are transformed into intervention requests which are sent through VC1 between the coherence subsystem and the cache intervention port. Intervention responses VC2 is the return path from the cache intervention port to the coherence subsystem, and VC3 is the return path from the coherence subsystem to the cache's primary port. Once past the serialization point, which may be in the memory agent or in the interconnect, as described in Section 2.5, memory read values can be sent back speculatively (the M_{Spec} arc) in parallel with the cache intervention request (the M_{Int} arc). The C2C arc is an optional transaction for cache-to-cache transfers, where a cache responds to an intervention request with both an intervention response and a memory value response. Request arcs are shown in blue while response arcs are shown in green. The graph is acyclic and hence deadlock free. The implementor is free to choose less or more channels as long as it can be reduced to this configuration. These channels are assumed to be in-order in the basic model and no ordering relationship between these channels is assumed or required.

Figure 2.5 Interconnect Model



Basic 4 Virtual Channel Static Dependence Graph

VC Virtual Channel

M Coherent Memory Subsystem

$\mathbf{C}_{Local}$ Cache of original requestor

$\mathbf{C}_{All}$ All caches within the coherence domain

The routing in the interconnect is address-based for the primary port requests and it is agent-based (for directory schemes) or broadcast (for snoopy schemes) for the intervention port. The responses for both intervention and primary ports are always routed back to the initiator in the basic model. It is possible to do 3-party forwarding (in directory schemes) and cache-cache transfers (in snoopy schemes). Tags can be used on both sets of channels to allow out-of-order responses. In snoopy schemes where transactions are not tagged the interconnect is also responsible for accumulating the intervention responses and returning them in-order. For directory schemes the interconnect could also hold the directory, in which case the interconnect is responsible for looking up the requests and then generating the proper requests and interventions to memory and the initiators respectively.

Various topologies can be used for the interconnect, from a single segment bus to a multistage switched network. The framework provides a deadlock free coherence protocol over 4 virtual channels. Implementations can use networks with almost no ordering guarantees as long as the implementation can be reduced/refined to the 4 channel

model described above. As an example the physical channels may be only 2, but it should be able to layer the 4 abstract virtual channels of the framework on top of the 2.

2.7 Coherent Transactions

Interaction between coherent caches and memories under the framework is done using coherent request/response transactions. Their encoding may vary with the choice of interconnect. This section describes their signal-independent semantics.

2.7.1 Coherent Requests

Table 2.2 describes the set of coherent requests defined by the coherency framework. Potentially all of these transactions can be observed at the primary master port. However the set visible at the intervention port of a coherent master depends on the system, for example the location of the directory. For a pure snoopy scheme these transactions will be "reflected" by the interconnect and broadcast to all the intervention ports in the system, so the entire set of transactions emanating at the master port will be observed at all of the intervention ports. In directory-based schemes, the intervention port might observe a reduced set of transactions.

Write-type transactions may be *collapsed* at the intervention port, such that they provide no data to the intervention port, consuming only the intervention port bandwidth necessary to place the transaction in the global order.

Table 2.2 Coherent Requests

Name	Description
CohReadOwn	Requests a writable copy of the cache line from the system. Usually generated for processor store misses, or Caching DMA store misses.
CohReadShare	Requests a readable copy of the cache line from the system. Usually generated for processor load misses or coherent DMA reads.
CohReadDiscard	Indicates to the system that the addressed line is leaving the coherence domain. This can be used by the system to avoid some unnecessary transactions (e.g.: E->M). An example use is by coherent I/O (DMA controllers).
CohReadShareAlways	Similar to CohReadShare, but indicates further that this agent will never write to this cache line. Typically generated by Instruction cache misses.
CohUpgrade	Requests ownership of a shared cache line from the system. It is usually generated for processor store hits to shared cache lines.
CohWriteBack	Transfers ownership of a cache line size of data back to a outer level of the cache hierarchy. It is usually generated as a side effect of processor load and store misses when dirty lines are victimized. [This transaction does not need to be broadcast even in a snoopy system, other than being sent back as a self-intervention.]

Table 2.2 Coherent Requests

Name	Description
CohCopyBack	Indicates to the system that the addressed line needs to be flushed from the system if in a dirty state (e.g.: M or O). The cache line can be retained in a valid state. An example use is by processor cache management instructions.
CohCopyBackInval	Indicates to the system that the addressed line needs to be flushed from the system if in a dirty state (e.g.: M or O) and also invalidated. An example use is by processor cache management instructions.
CohInvalidate	Indicates to the system that the addressed line needs to be purged from the system irrespective of its ownership status. When compared to CohWriteInvalidate, the difference is that no data transfer takes place. An example use is by some coherent DMA controllers which wish to separate the write path and also by processor cache management instructions.
CohWriteInvalidate	Injects new, possibly *sub cache line* data into a coherent system by either invalidating a cache line from the system and updating its value at the system memory location(full cache line case), or by merging the sub cache line data with the rest of the associated cache line data (sub cache line case)
CohCompletionSync	Indicates to the system that it needs to send a response back to the requestor as soon as this transaction has moved past a specific processing stage in the memory hierarchy. An example use is by an IO adapter to help flush coherent writes into the coherent system. Another example use is by the processor to flush writes to memory for non-coherent IO. No address information is sent in the request phase for this transaction. Implementation specific bits are used to indicate the processing stage within the system. These processing stages are chosen to guarantee ordering and/or visibility properties are met if the transaction has moved past such stages. The chosen processing stages are implementation specific.

2.7.2 Coherent Responses

Responses to coherent requests contain an *install state* for the cache line referenced. This is the state (see Table 2.1) that the requestor should assign to the affected line at the end of the transaction.

Slave agents responding to coherent requests that require change to cached state at the slave must indicate the completion status of that change in the response to the request.

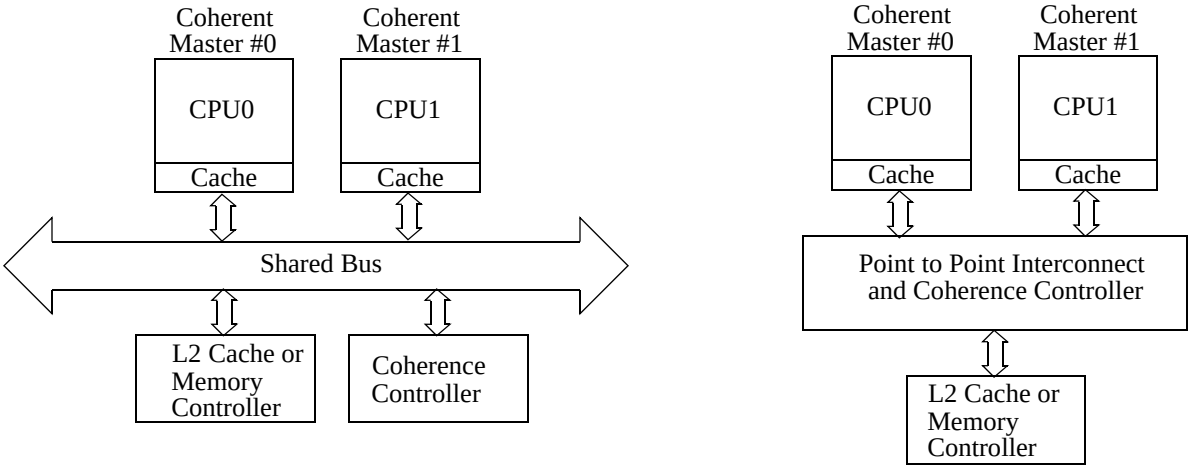
Coherent write transactions may, or may not have a main port response. Such a response may simplify implementations, but if ownership transfer of a cache line happens at self-intervention, the main port response is not necessary.

Chapter 3

Snoopy Coherence Protocol

This chapter describes the MESI coherence protocol used by coherent MIPS processor cores and interconnect. In a shared-memory multiprocessor system, different processors can share, access and modify data stored in main memory. Most modern processors are equipped with caches to improve memory access performance, and if the data being shared is also being cached, care must be taken that each processor receives the most up-to-date value of the shared data. Memory coherence protocols define when and how updates to shared locations are propagated to other processors. The protocol is a broadcast snoop based MESI protocol, that supports cache to cache transfers, with out-of-order data return and in-order intervention response. Out-of-order data return is supported to avoid stalling transactions in flight in case of uneven device latencies. Future revisions may use a MOESI protocol by including an additional 'Owned' state.

Figure 3.1 presents a high level block diagram of two coherent systems, with two coherent master agents with caches, coherent interconnect and a slave module (L2 cache or memory). It is possible to replace the L2 cache module with a memory module if an L2 cache is not required. Ports that do not support coherent transactions are termed 'legacy ports'. Each master agent contains a coherent data cache , an instruction cache that may or may not be coherent, and some buffers that hold transitory data lines (i.e., cache misses, upgrade requests, etc.). The slave can be a pure legacy slave with a hierarchy of caches (e.g. L3 cache) between it and memory, or it can directly be the memory controller. The behavior of the L2 cache based slave is well defined on the memory side. All coherent memory operations that require a memory transaction must be globally visible in the same order to all master agents. The precise order is defined by the memory consistency model (see Chapter 4).



(a) Coherent System with shared bus interconnect (b) Coherent System with point-to-point interconnect

Figure 3.1 Snoop-based MESI System - Block Diagram

3.1 Snoopy Cache Coherence Protocol

The MESI cache coherence protocol allows a coherent cache line to be in one of four states - Modified (M), Exclusive (E), Shared (S) or Invalid (I). Table 3.1 provides a brief explanation of when a cache line can be found in a particular state.

Table 3.1 Explanation of MESI states

State	Meaning
Modified (M)	Cache holds modified copy of data, main memory has a stale copy, no other cache has a copy
Exclusive (E)	Cache holds an unmodified copy of data, main memory still has valid copy, no other cache has a copy
Shared (S)	Cache holds an unmodified copy of the data, main memory has a valid copy, other caches may also have a copy
Invalid (I)	Cache does not hold the data, or cache line has been invalidated

3.1.1 Protocol Rules

3.1.1.1 Read by local processor

A read from the local processor can be satisfied by the cache (in case of a hit) if the line is in any state except Invalid (I). If the line is in the I state, a cache miss occurs, and the line must be fetched from the memory system. Depending on the Cacheability and Coherency Attributes (CCA) of the request, the fetched line may be installed as exclusive (E) or shared (S). For more details about CCAs, see Section 3.3.1, "Cacheability and Coherency Attributes."

If a cache miss occurs, the read request is propagated to the other caches within the coherent domain as a remote read request.

3.1.1.2 Write by local processor

A write from the local processor can be processed by the cache if the line is either in the Exclusive State (E), or the Modified (M) state.

If the line is in the I state, a cache miss occurs and a request must be made to fetch the line in exclusive state (i.e., no other processor/cache may have a copy of the data). In this case, the write request is propagated to the other caches within the coherent domain as a remote write request.

If the line is in the Shared (S) state, an upgrade request must first be sent out to convert the line to the ‘E’ state. Once the upgrade request is satisfied, the write can proceed.

3.1.1.3 Eviction from cache

A line can be evicted from the cache without any action being taken if it is in the ‘E’ state or the ‘S’ state.

If the line is in the ‘M’ state, it must first be written back to memory. Once the processor/cache receives a confirmation that the line has been written to memory, it can be discarded by the cache.

3.1.1.4 Remote Read Request

On a remote read request, the cache responds with the state of the cache line, and is required to take action in the following cases:

(a) If the line is in the E state, the line is downgraded to the S state

(b) If the line is in the M state, the cache must send the latest copy of the data. This data is sent to the other processor, and is used to update main memory. The state of the line in the local cache is then downgraded to the S state.

3.1.1.5 Remote Write Request

On a remote write request, if the line is in the S or E state, the cache responds with the state of the cache line and invalidates the local copy. If the line is in the M state, it must send an updated copy of the data to the requesting processor, and invalidate its own copy.

3.1.1.6 MESI State Transitions

The state transition diagram for the protocol is given in Figure 3.2.

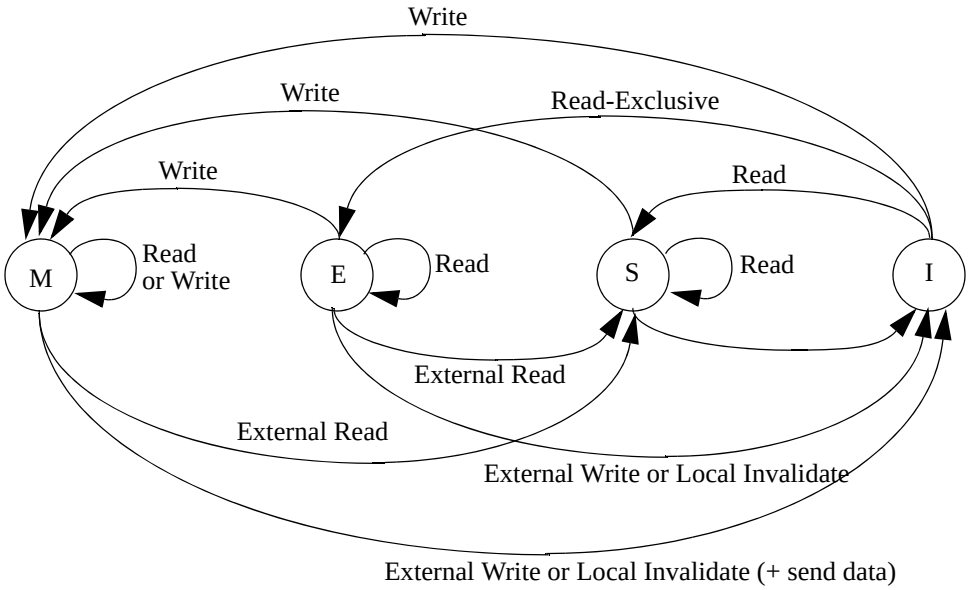


Figure 3.2 MESI State Transition Diagram

Note: (1) The transition from I->S or I->E depends on the Cacheability and Coherency Attribute (CCA) being used in addition to the global state of the cache line. If a CCA value of 4 is used, the system attempts to install the line in ‘E’ state if there are no sharers. For more details about CCAs, see Section 3.3.1, "Cacheability and Coherency Attributes.".

3.1.2 Establishing a Global Order

Since multiple CPUs can issue coherent read/write requests simultaneously, there must be some mechanism by which all read/write requests can be ordered, and this ordering must be made visible to each CPU. The self-intervention mechanism, described in Section 2.4.1, "Intervention," is used to order a CPUs requests with respect to requests from other CPUs. The new state of a cache line takes effect/can be visible only after the cache receives the self-intervention request.

3.2 Instruction Set Interactions with Cache Coherence

In this section, we describe features of the MIPS ISA that are affected by, or have an effect, on a coherent multiprocessor system.

3.2.1 CACHE instructions

The MIPS Privileged Resource Architecture defines various flavors of CACHE instructions (cache ops). Most of these are either targeted to specific caches or they have index semantics which are only usable by low level management routines.

CACHE instructions that use HIT type semantics are updated to have two flavors:

- If the address uses a non-coherent CCA, the behavior is as previously defined.
- If the address used uses a coherent CCA, the CACHE instruction may be *globalized.* This means the instruction affects all caches in the coherence domain at a particular level - L1, L2 or L3. This is done by broadcasting the command to all appropriate caches within the coherence domain. This globalized behavior with coherent CCAs is optional.

CACHE instructions that target non-coherent data caches (such as non-coherent L2/L3) , or use index semantics are unchanged in a coherent system.

If there are multiple levels of coherent caches in the system, it is still implementation specific whether L2 CACHE instructions operate on L1 caches first before operating on the L2 cache. Similarly, it is still implementation specific whether L3 CACHE instructions first operate on L1 caches and/or then operate on L2 caches before operating on the L3 cache. If multiple levels of caches are to be affected by one CACHE instruction, all of the affected cache levels must be processed in the same manner - either all affected cache levels use the globalized behavior or all affected cache levels use the legacy non-globalized behavior.

Table 3-1 lists the optional coherent behavior for HIT-type CACHE instructions.

Table 3-1 CACHE Instruction operations optionally affected by coherence

Code field in CACHE Instruction (20:18)	Name	Optional Coherent Behavior
0b100	Hit Invalidate	Operates on all instruction or data caches (of the specified level - I/D/S/T) in the coherence domain. Will invalidate a matching cache line without a write-back.
0b101	Hit WriteBack Invalidate	Operates on all data caches (of the specified level - D/S/T) in the coherence domain. A dirty line is first written back to memory before being invalidated. All matching cachelines are invalidated
0b110	Hit WriteBack	Operates on all data caches (of the specified level - D/S/T) in the coherence domain. A dirty line is written back to memory. All matching cachelines are left valid.

CACHE instructions can also be used to lock cache lines (i.e., prevent them from participating in the replacement/eviction algorithm). The most common usage model envisioned is to keep critical instructions or data in the cache to maintain performance or remove variability due to cache misses. This would require the process that uses the locked cache lines to be tied to that processor. This implies that there should be no coherent requests to a locked line,

or that the lock will take precedence over coherence. It is implementation specific whether cache-locking takes precedence over the snoopy MESI protocol or vice-versa.

3.2.2 Instruction Cache Management

In single CPU systems, SYNCI instructions are used to synchronize all caches to make writes to the instruction stream effective.

L1 instruction caches in a coherent multiprocessor system can be managed in one of these choices:

- Instruction Cache is coherently maintained by hardware. For this choice, the SYNCI instruction that uses a coherent CCA should be equivalent to a NOP.
- Instruction Cache is non-coherent and instead maintained by software. For this choice, the CACHE instructions targeting the instruction cache and SYNCI instructions that use coherence CCAs may optionally be globalized and made visible at all instruction caches.

3.2.3 Prefetch Instruction

The MIPS ISA defines two types of prefetch instructions (PREF and PREFX). Prefetch instructions that use coherent CCAs cause coherent transactions to be generated. Hence, a prepare-for-store prefetch operation results in a coherent request for exclusive ownership of the cache line. Since the instructions generate coherent transactions, a prefetch instruction issued on one core can cause data to be evicted from the cache in another core.

3.2.4 LoadLinked and Store Conditional Instructions

For a discussion on how LoadLinked and Store Conditional instructions are used as synchronization primitives, please refer to Section 4.6.1 “Atomic memory accesses using LoadLinked and StoreConditional”.

3.2.5 SYNC Instruction

In order to program a cached coherency system with a weakly ordered memory, system software needs some ordering guarantees at various points in a concurrent program. This is provided by barrier instructions. In the MIPS architecture, the barrier instructions use the SYNC opcode.

All cached coherent and uncached coherency attributes (CCAs) follow the ordering and/or completion rules specified by these SYNC instructions.

3.2.5.1 Completion Barriers

Previously, only a heavy weight barrier instruction was available to MIPS programmers: SYNC. This heavy weight type of barrier requires that all memory transactions listed in the program before the SYNC to have been completed before any memory transaction listed after the SYNC to be executed. Barriers that require instruction completion are known as completion barriers.

The completion of load instructions means that their target registers are written with the requested data. The completion of store instructions means that their data values have become globally visible. The completion of a SYNC instruction means that it no longer stalls subsequent memory instructions from being processed.

3.2.5.2 Ordering Barriers

With this Coherency specification, a lighter weight type of barrier is introduced. This type of barrier does not require instruction completion but rather only requires the ordering of how the memory requests become visible to the cache/memory hierarchy. This potentially reduces how many CPU cycles the barrier instruction must stall for before it completes. This type of barrier is known as an ordering barrier.

The term “Older” means that these instructions appear before the SYNC instruction in program order. The term “Younger” means that these instructions appear after the SYNC instruction in program order.

When a younger memory instruction is ordered against an older memory instruction, there are two requirements:

1. The older memory instruction must reach a stage in the load/store datapath after which no instruction re-ordering is possible before the younger memory instruction reaches the same stage in the load/store datapath.

2. If the older memory instruction generates a memory request to external memory and the younger memory instruction generates a memory request to external memory, the memory request belonging to older instruction must be globally visible before the memory request belonging to the younger instruction becomes globally visible.

There is no requirement on instruction completion ordering, for example hit-under-miss behavior is allowed (younger instruction can hit in a cache and complete before an older instruction which had earlier missed in the cache and must wait for pending data).

The following is an example that shows the behavior of these lighter-weight ordering barrier.

Let ExternalMemoryRequest* represent a memory request to external memory that is generated for any of the following reasons:

- Load to a cached address that misses in all caches within the processor core.

- Store to a cached address that misses in all caches within the processor core.

- Load to an uncached address.
- Store to an uncached address.

For the following code sequence, ExternalMemoryRequest2 is not allowed to happen before ExternalMemoryRequest1.

```
ExternalMemoryRequest1
SYNC_MB
ExternalMemoryRequest2
```

As a further enhancement, the Acquire and Release barrier types are used to minimize the memory orderings that must be maintained and still have software synchronization work.

3.2.5.3 Instruction Encodings for Barriers

Both the legacy SYNC and the newer types of barriers are encoded using the same major opcode. The different barrier types are differentiated using the SType field of the SYNC instruction. The SType field spans bits 10:6 of the instruction encoding.

The rest of this document uses the syntax SYNC(*value*) where *value* is the binary encoded number in the SType field of the SYNC instruction.

All multiprocessor implementations are required to implement the legacy SYNC(0). On MP implementations that do not implement all of the architecturally defined barrier types, the unimplemented barrier types must behave as SYNC(0).

Vendor-specific or implementation-specific barriers may be specified using the SType values of 0x1-0x3, 0x5 - 0xF.

SType values of 0x14 - 0x1F are reserved by MIPS Technologies for future extensions of the architecture. Neither Vendor-specific nor implementation-specific barriers types should use this range of SType values.

The following table summarizes the different Memory Barrier types.

Table 3.2 Memory Barrier Types

SType Field, eg. Instruction Encoding [10:6]	Name	Older instructions which must reach the load/store ordering point before the SYNC instruction completes.	Younger instructions which must reach the load/store ordering point only after the SYNC instruction completes.	Older instructions which must be completed when the SYNC instruction completes	Compliance
0x0	SYNC or SYNC(0)	Loads, Stores	Loads, Stores	Loads, Stores	Required
0x4	SYNC_WMB or SYNC(4)	Stores	Stores		Optional
0x10	SYNC_MB or SYNC(16)	Loads, Stores	Loads, Stores		Optional
0x11	SYNC_ACQUIRE or SYNC(17)	Loads	Loads, Stores		Optional
0x12	SYNC_RELEASE or SYNC(18)	Loads, Stores	Stores		Optional
0x13	SYNC_RMB or SYNC(19)	Loads	Loads		Optional
0x1-0x3, 0x5-0xF					Implementation-Specific and Vendor Specific Sync Types
0x14 - 0x1F	RESERVED				Reserved for MIPS Technologies for future extension of the architecture.

For the purposes of these barrier types, SC instructions are assumed to be a “Store”. LL instructions behave as a “Load”. PREF and PREFX instructions are not affected by memory barriers.

3.3 Priviledged Architecture interactions with Coherence

3.3.1 Cacheability and Coherency Attributes

The MIPS Privileged Resource Architecture (PRA) [*see MIPS document MD00090: MIPS32 Architecture for Programmers Volume III: The MIPS32 Privileged Resource Architecture*] defines CCA (Cacheability and Coherency Attribute) bits that can be specified per virtual memory page. These CCA bits can be used to control whether or not a given memory page is cacheable, or within the coherence domain.

Table 3.3 summarizes the available Cacheability and Coherency Attributes.

Table 3.3 Cacheability and Coherency Attributes

CCA Value	Architectural Definition	MIPS Technologies Inc. Historical Usage	Compliance
0x0			Optional, Implementation-specific
0x1			Optional, Implementation-specific
0x2	Uncached		Required
0x3	Cacheable, Non-Coherent		Required
0x4		Cacheable, Coherent, Install as Exclusive	Optional, Implementation-specific
0x5		Cacheable, Coherent, Install as Shared	Optional, Implementation-specific
0x6			Optional, Implementation-specific
0x7		Uncached, Accelerated	Optional, Implementation-specific

Historically, MIPS processor cores have used CCA values of 0x4 (Cacheable, Coherent, Writeback, Write Allocate, read install in Exclusive), and 0x5 (Cacheable, Coherent, Writeback, Write Allocate, read install in Shared) for coherent transactions. However, this is not mandated by the architecture. For a more complete listing of historic uses and descriptions of CCA values, refer to the *MIPS Architecture For Programmers Manual, Volume III*.

The coherence model does not define any ordering between cached and uncached (including uncached-accelerated) operation name spaces. See the Section 3.2.5 “SYNC Instruction” for the SYNC definitions and Section 4.6.2 “Memory Barriers” to see how they can be used to create specific load/store orderings.

3.3.2 TLB Coherence

In the MIPS ISA, TLBs are managed through software instructions. Each core in the multiprocessor system is expected to maintain its own virtual to physical page mappings in the TLB. If one core modifies the mapping in some way, there must be a mechanism by which the TLBs on the other cores can be made aware of the change. Since modifying memory mappings is a privileged operation, it is expected that the operating system will handle TLB coherence through the use of inter processor interrupts (IPI). For more details on inter processor interrupts, refer to *MD00577 Multiprocessor Platform Specification*.

3.3.3 Errors and Exceptions

Coherent transactions may result in these type of errors:

- cache errors on remote cores (ie, not the requesting core)
- bus errors occurring on remote cores
- situations in which a cache line is in the ‘M’ or ‘E’ state on one core while also being in the ‘M’, ‘E’ or ‘S’ states on any other core.

These situations should result in an error or exception. The exact mechanism by which these errors are reported and handled is implementation dependent.

3.4 I/O Coherence

I/O transactions are defined to be transactions sourced by external masters other than the coherent CPUs. Transactions generated by a DMA controller or external PCI/PCI-Express master are some examples.

Hardware I/O Coherence is where such external transactions automatically update the appropriate cachelines within the coherence domain, most probably invalidating cachelines affected by such I/O transactions (without software explicitly invalidating the cachelines).

Software I/O Coherence is where software has to explictly invalidate the appropriate cachelines after such external transactions.

It is strongly recommended that any MIPS multiprocessor implementation support Hardware I/O Coherence. However, there might be I/O devices that do not support coherence. Both classes of IO devices should be supported.

Hardware I/O coherence may be supported in one of the following ways:

3. Placing Coherent IO devices on a separate interconnect from that used by non-Coherent IO.

4. Using a single system interconnect for both Coherent and non-Coherent IO and differentiating between them by using Coherent and non-Coherent interconnect transaction types. The use of different transaction types can be chosen through using IO TLBs, which would create both coherent and non-coherent mappings of system memory.

If Hardware I/O coherence is not supported by an implementation, the implementation must supply globalized CACHE instructions to decrease the overhead of Software I/O coherence.

Chapter 4

Memory Consistency, Ordering and Synchronization

4.1 Definitions

Globally Visible - A point in time when a data value that is updated by the local processor is accessible by all non-local processors within the system.

Program Order - The order in which assembler-level instructions are listed in compiled/assembled program binaries.

Execution Order - The order in which the processor actually issues the listed instructions to its execution units**.**

Visibility Order - The order in which data values which are updated by the processors within the system are made accessible to the processors.

Store Atomicity - Whether the updated value from a store operation is made accessible to all processors within the system simultaneously.

Local Processor - the processor which executed the instruction of interest.

4.2 Execution Order Behavior

The following statements are meant to summarize the allowed behavior during instruction execution:

1. All processors are self-consistent. This means that the final results of executed usermode instructions are the same as if they were executed on a sequential one-instruction-at-a-time uniprocessor.

2. Uniprocessor Data Dependencies are always maintained. These include read-after-write, write-after-write and write-after-read hazards.

3. Uniprocessor Control Dependencies are always maintained. Any instruction which depends on a conditional branch is not allowed to modify architectural state unless the branch taken/not-taken decision is resolved and the instruction is on the code path which must be executed.

4. Loads not using any of the Uncached Cacheability and Coherency Attributes (CCA=2, 7) can be executed speculatively and in any order.

5. Stores can be buffered.

6. Stores to system memory are not done speculatively. Memory is only written if the instruction is committed to the architectural state (retired/graduated).

7. Loads and Stores using the Uncached coherency (CCA=2) are always executed in program order.

8. All Loads and Stores follow the ordering and completion rules of the SYNC instruction(s). (See following section on Memory Barriers).

9. Data prefetches (PREF and PREFX opcodes) never modify the contents of architectural state, only the cache hierarchy. The Prepare for Store flavor of prefetch is an exception to this rule.

4.2.1 Memory Operation Relaxations allowed for Execution Order.

The MIPS Architecture allows *weak ordering* of memory transactions.

Specific implementation may follow stronger ordering models that more closely mimic program ordering.

The memory barrier instructions (SYNC, etc.) can be used to prevent bypassing when so desired.

4.2.1.1 Younger Load can bypass an older store with different cacheblock address.

Rationale for this relaxation:

- Allows for Store Buffers and Write Buffers with FIFO behavior.
- Allows for hit-under-miss instruction state committal with non-blocking caches.
- Allows for lazy/opportunistic writebacks

4.2.1.2 Younger Store can bypass an older store with different cacheblock address.

Rationale for this relaxation:

- Allows for hit-under-miss instruction state committal with non-blocking caches.
- Allows for non-FIFO behavior for writes buffers - multiple write buffers, write coalescing/merging.

4.2.1.3 Younger Store can bypass an older load with different cacheblock address.

Rationale for this relaxation:

- Allows for hit-under-miss instruction state committal with non-blocking caches.

4.2.1.4 Younger Load can bypass an older load with different cacheblock address.

Rationale for this relaxation:

- Allows for hit-under-miss instruction state committal with non-blocking caches.

4.3 Visibility Order Behavior

Within any single processor, execution order must obey all data and control dependencies. However, the resulting execution order is not guaranteed to be the same as the order in which the resulting memory transactions become visible to other processors.

Some examples of when the visibility order can be different from the execution order:

- Buffered writes within the local processor are allowed to appear to the other processors in a non-FIFO order.
- Non-dependent loads which miss in the local non-blocking cache hierarchy are allowed to appear to the other processors in an order that is different from the local execution order.

Stores using the Uncached coherency (CCA=2) are always made visible in program order.

Loads and Stores must maintain the ordering and completion semantics specified by the different types of the SYNC instruction.

Store Serialization: When a store operation is made visible to any non-local processor, the memory location can not be modified until any previous modification to that location has been acknowledged by all processors. Acknowledgement means that either the processor has the updated value or has modified its cacheline state accordingly. This means that the newer store can not proceed until the older store is globally visible.

Store Atomicity: When a store operation is made visible to any non-local processor, no subsequent read is allowed to receive the new value until all processors have acknowledged the modification. Again, acknowledgement means that either the processors have updated their values or have modified their cacheline state accordingly. This means the newer load to the location can not proceed until the older store is globally visible.

4.3.1 Memory Operation Relaxations allowed for Visibility Order.

4.3.1.1 Read local store early

A store value that is generated by the local processor is allowed to be visible sooner within the local processor then when the new value is visible to other non-local processors.

The memory barrier instructions (SYNC, etc.) can be used to prevent this bypassing when so desired.

Rationale for this relaxation:

- Allows for forwarding of store data from store/write buffers.

4.4 Illegal Memory Transaction Sequences

To help describe the desired behaviors, memory transaction sequences which are not allowed by the architecture are now listed.

4.4.1 Notation used in Sequences

Thread 1	Thread 2
instr1 - first instruction in thread1	instr4 - 1st instruction in thread2
instr2 - 2nd instruction in thread1	instr5 - 2nd instruction in thread2
instr3 - 3rd instruction in thread1	

The execution of Thread2 could have occurred before, after or during the execution of Thread1.There is no assumed ordering between Thread1 and Thread2 other than what is deduced by the results from load instructions.

Each thread can represent a separate processor core or VPE or TC.

Please refer to Section 3.2.5 “SYNC Instruction” for a description of the completion memory barrier - SYNC(0) that is used in these example sequences.

4.4.2 Illegal Sequence 1

Initially variable X = 1.

Thread 1	Thread 2
instr1 - Store a value of 2 to variable X	instr2 - Loads variable X and gets value of 2
	instr3 - Loads variable X and gets value of 1

Analysis: Once a processor receives an updated value, it must not see an older value. Memory transactions are not allowed to appear to go back in time.

4.4.3 Illegal Sequence 2

Initially variable X = 1.

Thread 1	Thread 2
instr1 - Store a value of 2 to variable X	instr2 - Store a value of 3 to variable X
	instr3 - Loads variable X and gets value of 2
	instr4 - Loads variable X and gets value of 3

Analysis:

Statement1 - The value received by instr3 says that instr1 was the last visible store to variable X.

Statement2 - The value received by instr4 indicates the opposite; that instr2 was the last visible store to variable X.

Statement3 - The Read-after-Read data dependency hazard requires instr4 to be executed after instr3.

Both Statement1 and Statement2 can’t be both true.This is a contradiction and thus this sequence is not allowed.

4.4.4 Illegal Sequence 3

Initially variable X = 1.

Thread 1	Thread 2	Thread 3
instr1 - Store a value of 2 to variable X	instr3 - Store a value of 3 to variable X	instr4 - Loads variable X and gets value of 3
instr2 - Loads variable X and gets value of 3		instr5 - Loads variable X and gets value of 2

Analysis:

Statement1 - The value received by instr2 implies that instr3 was the last visible store to variable X.

Statement2 - The value received by instr4 also implies that the store from instr3 is globally visible.

Statement3 - The value received by instr5 implies that instr1 was the last visible store to variable X.

Statement 3 contradicts Statements1&2 on which store is the last visible one to variable X. Thus this sequence is not allowed.

4.4.5 Illegal Sequence 4

Initially variables X and Y = 1. X and Y are different cacheblock addresses.

Thread 1	Thread 2
instr1 - Store a value of 2 to variable X	instr4 - Load variable Y and get value of 2
instr2 - Sync(0)	instr5 - Sync(0)
instr3 - Store a value of 2 to variable Y	instr6 - Load variable X and get value of 1

Analysis:

Statement1 - The value received by instr4 implies that instr3 was the last visible store to variable Y. This also implies that instr1 happened even earlier due to ordering required by the SYNC instruction in instr2.

Statement2 - The value received by instr6 implies that either:

Statement2a) instr1 was not executed yet OR

Statement2b) The Sync in instr2 did not make the store value of instr1 globally visible

Statement2a) is a contradiction with Statement1 and thus this sequence is not allowed. Memory transactions can not appear to travel back in time.

Statement2b) is not allowed by the architecture and thus this sequence is not allowed. All Processors have to implement SYNC properly.

Since the writes to X and Y are non-dependent within Thread1 and the reads to X and Y are non-dependent within Thread2, both threads need the memory barriers to ensure that these register values are precluded.

4.4.6 Illegal Sequence 5

Initially variables X and Y = 1. X and Y are different cacheblock addresses.

Thread 1	Thread 2
instr1 - Store a value of 2 to variable X	instr4 - Store a value of 2 to variable Y
instr2 - Sync(0)	instr5 - Sync(0)
instr3 - Load variable Y and get value of 1	instr6 - Load variable X and get value of 1

Analysis:

Statement1 - The SYNC in instr2 requires that the store value of instr1 is globally visible before the execution of instr3.

Statement2 - The value received by instr3 implies that the store value of instr4 is globally visible after the store value of instr1 is globally visible due to Statement1.

Statement3 - The SYNC in instr5 requires that the store value of instr4 is globally visible before the execution of instr6.

Statement4 - The value received by instr6 implies that the store value of instr1 is globally visible after the store value of instr4 is globally visible due to Statement3.

Statement2 and Statement4 contradict each other and thus this sequence is not allowed.

The code sequences for Sequence 5 are used to provide mutual exclusion in Dekker’s Algorithm. For mutual exclusion to be provided, these results must not happen (instr 3 and instr 6 both receive stale data values).

4.4.7 Illegal Sequence 6

Initially variables X and Y = 1. X and Y are different cacheblock addresses.

Thread 1	Thread 2	Thread 3	Thread 4
instr1 - Store a value of 2 to variable X	instr2 - Store a value of 2 to variable Y	instr3 - Load variable X and get value of 2	instr6 - Load variable Y and get value of 2
		instr4 - Sync(0)	instr7 - Sync(0)
		instr5 - Load variable Y and get value of 1	instr8 - Load variable X and get value of 1

Analysis: This scenario is very similar to Illegal Sequence 5.

Statement1 - The values received by instr(3-5) implies that the store value from instr2 was globally visible after the store value from instr1 is globally visible.

Statement2 - The values received by instr(6-8) implies that the store value from instr1 was globally visible after the store value from instr2 is globally visible.

Statements 1and 2 contradict each other and thus this sequence is not allowed.

4.4.8 Illegal Sequence 7

Initially variable X = 1.

Thread 1	Thread 2	Thread 3	Thread 4
instr1 - Store a value of 2 to variable X	instr2 - Store a value of 3 to variable X	instr3 - Load variable X and get value of 2	instr6 - Load variable X and get value of 3
		instr4 - Sync(0)	instr7 - Sync(0)
		instr5 - Load variable X and get value of 3	instr8 - Load variable X and get value of 2

Analysis:

Statement1 - The values received by instr(3-5) implies that the store value from instr2 was globally visible after the store value from instr1 is globally visible.

Statement2 - The values received by instr(6-8) implies that the store value from instr1 was globally visible after the store value from instr2 is globally visible.

Statements 1 and 2 contradict each other and thus this sequence is not allowed.

4.4.9 Illegal Sequence 8

Initially variable X = 1.

Thread 1	Thread 2
instr1 - Store a value of 2 to variable X	instr4 - Store a value of 3 to variable X
instr2 - Load variable X and get value of 2	instr5 - Load variable X and get value of 3
instr3 - Load variable X and get value of 3	instr6 - Load variable X and get value of 2

Analysis:

Statement1 - The value received by instr(5,6) implies that instr1 was the last globally visible write to X.

Statement2 - The value received by instr(2,3) implies that instr4 was the last globally visible write to X.

Statement2 contradicts Statement1 and thus this sequence is not allowed.

4.4.10 Illegal Sequence 9

Initially variables X and Y = 0. X and Y are different cacheblock addresses.

Thread 1	Thread 2	Thread 3
instr1 - Store a value of 1to variable X	instr2 - Load variable X	instr5 - Load variable Y
	instr3 - Branch back to instr2 if instr2 received value of 0	instr6 - Branch back to instr5 if instr5 received value of 0
	instr4 - Stores a value of 1 to variable Y	instr7 - Loads variable X and gets value of 0

Analysis:

Statement1 - Due to instruction dependency, for instr4 to be executed, the store value from instr1 must be visible to Thread2.

Statement2 - Due to instruction dependency, for instr7 to be executed, the store value from instr4 must be visible to Thread3.

Statement3 - The Store Atomicity requirement states that Thread3 would see the instr1 store value at the same time as Thread2.

The value received by instr7 implies either:

Statement4a - that instr1 has not executed yet when instr7 is executed OR

Statement4b - the store value from instr1 isn’t visible yet to Thread3.

Statement4a breaks the instruction dependencies of Thread2 and Thread3 and thus this sequence is not allowed.

Statement4b contradicts Statements 3 and thus this sequence is not allowed.

4.4.11 Illegal Sequence 10

Initially variables X, Y, Z= 0. X, Y, and Z are all different cacheblock addresses.

Thread 1	Thread 2	Thread 3
instr1 - Store a value of 1 to variable X	instr4 - Load variable Y	instr7 - Load variable Z
instr2 - Sync(0)	instr5 - Branch back to instr4 if instr4 received value of 0	instr8 - Branch back to instr7 if instr7 received value of 0
instr3 - Store a value of 1 to variable Y	instr6 - Stores a value of 1 to variable Z	inst9 - Loads variable X and gets value of 0

Analysis:

Statement1 - For instr6 to be executed, the store value from instr3 must be globally visible.

Statement2 - For instr9 to be executed, the store value from instr6 must be globally visible.

Statement3 - Due to the store atomicity requirement, that means the store value from instr3 must also be visible to Thread3 when instr9 is executed since it is visible to Thread2.

Statement4 - Due to the SYNC in instr2, the store value from instr1 must also be visible to Thread3 when instr9 is executed.

Statement5 - The value received by instr9 implies that the store value from instr1 is not yet visible to Thread3.

Statement5 contradicts with Statements 4 and thus this sequence is not allowed.

4.4.12 Illegal Sequence 11

Initially word variable X= 0x01000001. The CPUs and memory are Little-Endian.

Thread 1	Thread 2
instr1 - Store a byte value of 2 to variable X	instr4 - Store a byte value of 2 to variable X+3
instr2 - Sync(0)	instr5 - Sync(0)
instr3 - Load word variable X and get value of 0x01000002	instr6 - Load word variable X and get value of 0x02000001

Analysis: This scenario is very similar to Illegal Sequence 5, but for overlapping memory locations.

Analysis:

Statement1 - The SYNC in instr2 requires that the store value of instr1 is globally visible before the execution of instr3.

Statement2 - The value received by instr3 implies that the store value of instr4 is globally visible after the store value of instr1 is globally visible due to Statement1.

Statement3 - The SYNC in instr5 requires that the store value of instr4 is globally visible before the execution of instr6.

Statement4 - The value received by instr6 implies that the store value of instr1 is globally visible after the store value of instr4 is globally visible due to Statement3.

Statement2 and Statement4 contradict each other and thus this sequence is not allowed.

4.5 Legal Memory Transaction Sequences

The notation is the same as that used in the previous section.

4.5.1 Legal Sequence 1

Initially variables X and Y = 1. X and Y are different cacheblock addresses.

Thread 1	Thread 2
instr1 - Store a value of 2 to variable X	instr3 - Load variable Y and get value of 2
instr2 - Store a value of 2 to variable Y	instr4 - Load variable X and get value of 1

Analysis -

Implementations are allowed to reorder instr2 ahead of instr1, since the stores are to different cacheblock addresses, so it is possible instr4 can receive the updated value while instr3 does not.

Implementations are allowed to reorder instr4 ahead of instr3, since the loads are to different cacheblock addresses, so instr4 can be executed before the store value of instr1 is globally visible.

4.5.2 Legal Sequence 2

Initially variables X and Y = 1. X and Y are different cacheblock addresses.

Thread 1	Thread 2
instr1 - Store a value of 2 to variable X	instr3 - Load variable Y and get value of 2
instr2 - Store a value of 2 to variable Y	instr4- Sync(0)
	instr5 - Load variable X and get value of 1

Analysis -

Implementations are allowed to reorder instr2 ahead of instr1, since the stores are to different cacheblock addresses, so it is possible instr3 can receive the updated value while instr5 does not.

To preclude these load values, Thread1 needs to have a memory barrier placed between the store instructions. Refer to Section 4.4.5 “Illegal Sequence 4”.

4.5.3 Legal Sequence 3

Initially variables X and Y = 1. X and Y are different cacheblock addresses.

Thread 1	Thread 2
instr1 - Store a value of 2 to variable X	instr4 - Load variable Y and get value of 2
instr2 - Sync(0)	instr5 - Load variable X and get value of 1
instr3 - Store a value of 2 to variable Y	

Analysis -

Implementations are allowed to reorder instr5 ahead of instr4, since the loads are to different cacheblock addresses, so it is possible instr5 can receive the updated value while instr4 does not.

To preclude these load values, Thread2 needs to have a memory barrier placed between the load instructions. Refer to Section 4.4.5 “Illegal Sequence 4”.

4.5.4 Legal Sequence 4

Initially variables X and Y = 1. X and Y are different cacheblock addresses.

Thread 1	Thread 2
instr1 - Store a value of 2 to variable X	instr3 - Store a value of 2 to variable Y
instr2 -Load variable Y and get value of 1	instr4 - Load variable X and get value of 1

Analysis -

Within each thread, the store and load instructions use different cacheblock addresses. For that reason, implementations are allowed to issue the loads ahead of the stores.

Processors are allowed to buffer writes. There can be a time delay between when the write is first buffered and when the write is made visible to non-local processors. There is no requirement that either store has been made globally visible when instr2 executes nor when instr4 executes.

To force either store to be globally visible before the subsequent load instruction is executed, a SYNC(0) would be placed after the store instruction.

4.5.5 Legal Sequence 5

Initially variables X and Y = 1. X and Y are different cacheblock addresses.

Thread 1	Thread 2
instr1 - Store a value of 2 to variable X	instr4 - Store a value of 2 to variable Y
instr2 - Load variable X and get value of 2	instr5 - Load variable Y and get value of 2
instr3 - Load variable Y and get value of 1	instr6 - Load variable X and get value of 1

Analysis -

For Thread1, instr3 can be issued ahead of the other instructions since instr3 doesn't use the same cacheblock address. Similarly for Thread2 with instr6.

Processors are allowed to buffer writes. There can be a time delay between when the write is first buffered and when the write is made visible to non-local processors. There is no requirement that either store has been made globally visible when instr3 executes nor when instr6 executes. Processors are allowed to forward data from the write buffer to subsequent loads within the local processor before the store is globally visible.

4.5.6 Legal Sequence 6

Initially variable X = 1.

Thread 1	Thread 2
instr1 - Store a value of 2 to variable X	instr3 - Store a value of 3 to variable X
instr2 - Load variable X and get value of 2	instr4 - Load variable X and get value of 3

Analysis -

Processors are allowed to buffer writes. There can be a time delay between when the write is first buffered and when the write is made visible to non-local processors. There is no requirement that either store has been made globally visible when instr2 executes nor when instr4 executes. Processors are allowed to forward data from the write buffer to subsequent loads within the local processor before the store is globally visible.

Since there is no pre-ordained ordering between the times when the store values of instr1 and instr3 becoming globally visible, instr2 gets its value as if instr1 was the last globally visible store to X at that time. Similarly instr4 gets its value as if instr3 was the last globally visible store at that time. Once both stores are globally ordered, the instruction which received the stale value can be considered to have executed before the latter store in the global order.

4.5.7 Legal Sequence 7

Initially word variable X = 0x00000000. The CPUs and memory are Little-Endian.

Thread 1	Thread 2
instr1 - Store a byte value of 0x01 to variable X	instr3 - Store a byte value of 0x02 to variable X+3
instr2 - Load word from variable X and get value of 0x00000001	instr4 - Load word variable X and get value of 0x02000000

Analysis -

Processors are allowed to buffer writes. There can be a time delay between when the write is first buffered and when the write is made visible to non-local processors. There is no requirement that either store has been made globally visible when instr2 executes nor when instr4 executes. Processors are allowed to forward data from the write buffer to subsequent loads within the local processor before the store is globally visible.

Warning to implementors: Though allowed by the architecture, this specific case has the undesirable effect of having Thread1 and Thread2 seeing different store orderings to the over-lapping memory location. This would not be the behavior once either of the stores is globally visible. To maintain a more uniform execution behavior, some implementations might want to avoid this case.

4.5.8 Legal Sequence 8

Initially word variable X = 0x00000000. The CPUs and memory are Little-Endian.

Thread 1	Thread 2
instr1 - Load word from variable X and get value of 0x000000000.	instr5 - Load word from variable X and get value of 0x000000000
instr2 - Store a word value of 0xAAAAAAAA to variable X	instr6 - Store a byte value of 0x55 to variable X
instr3 - Load word from variable X and get value of 0xAAAAAAAA	instr7 - Load word from variable X and get value of 0x00000055
instr4 - Load word from variable X and get value of 0xAAAAAA55	instr8 - Load word from variable X and get value of 0xAAAAAA55

Analysis -

Processors are allowed to buffer writes. There can be a time delay between when the write is first buffered and when the write is made visible to non-local processors. There is no requirement that either store has been made globally visible when instr3 executes nor when instr7 executes. Processors are allowed to forward data from the write buffer to subsequent loads within the local processor before the store is globally visible.

Warning to implementors: Though allowed by the architecture, this specific case has the undesirable effect of having Thread1 and Thread2 seeing different store orderings to the over-lapping memory location. This would not be the behavior once either of the stores is globally visible. To maintain a more uniform execution behavior, some implementations might want to avoid this case.

4.5.9 Legal Sequence 9

Initially variables X and Y = 1. X and Y are different cacheblock addresses.

Thread 1	Thread 2
instr1 - Store a value of 2 to variable X	instr4 - Load variable Y and get value of 2
instr2 - Load variable X and get value of 2	instr5 - Load variable X and get value of 1
instr3 - Store instr2 result to variable Y.	

Analysis -

For Thread1, Read-after-Write and Write-after-Read data dependencies ensure that instr2 is executed after instr1 and instr3 is executed after instr2.

There is no requirement that either store has been made globally visible when instr4 executes nor when instr5 executes.

The global visibility order is allowed to be different from Thread1’s execution order.

4.6 Synchronization Primitives

4.6.1 Atomic memory accesses using LoadLinked and StoreConditional

When processors want to share a data-structure or hardware register, they are often accessed in this manner:

```
Arbitrate for ownership of shared resource
Critical Section - Access shared resource
Release ownership of shared resource
```

The intention is to give exclusive access to the shared resource. However, on older computer architectures, even the arbitration step required exclusive access to system resources. For example, load/store commands which are atomic would lock the system interconnect to memory. This would cause the system to act serially by precluding concurrent memory transactions when any processor was arbitrating for the shared resource. In systems with large number of processors, this loss of concurrency can be noticeable.

To allow concurrency and sharing to occur at the same time, the LoadLinked and StoreConditional instruction pairs are used to lock specific memory locations instead locking hardware resources such as system interconnects/busses.These two instructions allows software to check to see if it can read and then modify a memory location in an atomic manner without actually stopping other processors in the system. This is done by checking for conflicting accesses by other processors (or threads) instead of physically locking the memory system. Another nice feature of this scheme is that it does not require the more specialized atomic/locking memory transactions that are part of other architectures.

The check for atomicity between the LL and SC instructions is done this way:

When the LL instruction is executed, an internal state bit called the LLBit is set within the processor. The target address of the LL instruction is also saved in the LLAddr COP0 register.

The local processor snoops to see if there is any memory transaction from another processor to the LLAddr memory location during the time between the LL instruction is executed and when the SC instruction is executed. If there was, the LLBit is cleared by the local processor. If the LLBit is cleared, the SC instruction will fail and not write the semaphore. This is the mechanism to ensure atomicity across multiple processors.

The local processor will also clear the LLBit if there was a context switch which happened between the LL instruction and the SC instruction execution times. This is to prevent another software thread from modifying the target address without the current thread knowing about it. This is done by the ERET instruction always clearing the LLBit. This is the mechanism to ensure atomicity across multiple software threads/programs within one processor.

Since the processor snoops for the modifications for the LLAddr memory location, the local processor does not have to re-fetch the memory location to ensure it has the up-to-date value for the memory location every time the LL/SC instructions are executed. This greatly reduces the amount of memory traffic that would have occurred if multiple processors were arbitrating for ownership. In fact, the LL/SC instructions are meant to be used only with cached addresses. Behavior when using uncached addresses for LL/SC operations is undefined by the architecture.

The following sections show how atomic primitives can be implemented using the LL and SC instructions. The register a0 holds the address of the memory location to be modified.

4.6.1.1 Spin Locks

A Spin lock is used to give exclusive access to a shared resource to the winner of the lock. The requestor keeps polling the status of the lock, hence the name “spin”. In the example, if the semaphore has a value of zero, it means the resource is currently not used while any non-zero value means the resource is in use.

```
LockLoop:     ll     t0, 0(a0)        // Read the semaphore.
              bne    t0, 0, LockLoop  // If non-zero, someone else already
              li     t0, 0x1          // set semaphore, so try again.
              sc     t0, 0(a0)        // Try writing semaphore.
              beq    t0, 0, LockLoop  // Check if LL-SC sequence was atomic
              nop                     // if not, try again.
                                      // Need memory barrier here, see next section.
```

For completeness, the code for the Semaphore unlock is given below:

```
UnLock:       sw     r0, 0(a0)        // Need memory barrier before this store,
                                      // see next section.
```

4.6.1.2 Fetch and Op

Fetch&Op is a class of primitives that allows multiple processors to act concurrently. For example, Fetch&Increment with Fetch&Decrement can be used to implement a counting semaphore which gives access to more than one processor at a time (for example, give access to an array of memory locations). They are also useful in large inteconnection networks where the intermediate nodes can combine semaphore modifications from multiple processors (eg. combining networks).

```
Loop:         ll     t0, 0(a0)        // Read the semaphore.
              OP     t0, t0, t1       // Do something to semaphore value.
              sc     t0, 0(a0)        // Try writing semaphore.
              beq    t0, 0, Loop      // Check if LL-SC sequence was atomic
              nop                     // if not, try again.
                                      // Need memory barrier here, see next section.
```

Counting Semaphore

Here’s an example where a Fetch&Decrement and Fetch&Increment are used to implement a counting semaphore. The semaphore (pointed by register a0), is set to an initial value N. The semaphore allows N processors to enter their critical sections. When the (N+1)th processor checks the semaphore, the first branch instruction makes it spin-wait until one of the previously requestors has incremented the semaphore.

```
Loop1:        ll     t0, 0(a0)        // Read the counter.
              blez   t0, Loop1        // Check if counter==0,
              nop                     // if not, try again.
              addi   t1, t0, -1       // Decrement value.
              sc     t1, 0(a0)        // Try writing counter.
              beq    t1, 0,Loop1      // Check if LL-SC sequence was atomic
              nop                     // if not, try again.
                                      // Need memory barrier here, see next section.
              << Critical Section>>
```

```
                                    // Need memory barrier here see,next section.
Loop2:      ll      t0, 0(a0)       // Read the counter.
            addi    t1, t0, 1       // Increment value.
            sc      t1, 0(a0)       // Try writing counter.
            beq     t1, 0, Loop2    // Check if LL-SC sequence was atomic
            nop                     // if not, try again.
                                    // Need memory barrier here, see next section.
```

4.6.1.3 Compare and Swap

Some older computer architectures have an atomic compare and swap instruction in which a memory location is compared with a register value. If there is a match, a second register value is written to the memory location. The following code example shows how an atomic compare and swap operation can be implemented with LL and SC instructions. The register t1 holds the expected original value of the memory location while the register t2 holds the replacement value.

```
Loop:       ll      t0, 0(a0)       // Read memory location.
            bne     t0, t1, Exit    // Check for Match, if not quit.
            or      t0, t2, t2      // Get 2nd value.
            sc      t0, 0(a0)       // Try writing location.
            beqz    t0, Loop        // Check if LL-SC sequence was atomic
            nop                     // if not, try again.
Exit:                               // Need memory barrier here, see next section.
```

4.6.2 Memory Barriers

New flavors of memory barriers are introduced with this specification. Among the new additions are ordering barriers which are lighter-weight than the pre-existing completion barriers. Barriers with either acquire or release semantics are also introduced. Please refer to Section 3.2.5 "SYNC Instruction".

Some code examples are now given when memory barriers are necessary.

In the examples, the lighter-weight sync types are shown but the heavier weight sync(0) can always be used. The lighter weight type is preferred for performance reasons.

4.6.2.1 Acquiring and Releasing Locks

For this example, the state of the lock must be globally visible before the code within the critical section is executed.

```
    Lock(a0)
                    // Don't start Critical Section until Lock is acquired
    sync_acquire    //or sync(0)
<< Critical Section >>
                    // Critical Section must complete before releasing lock
    sync_release    //or sync(0)
Unlock(a0)
```

See the previous section on Spin Locks for the code that implements Lock() and Unlock(). It would be good practice that include the barriers within the Lock/Unlock routines/macros.

Real world examples of using locks include

- data structures like linked-lists being shared by multiple threads
- IO devices being shared by multiple threads.

4.6.2.2 Producer and Consumer

In this example, the producer is writing both the data for the consumer as well as a producer-is-finished/consumer-can-begin flag. The producer has to ensure that the data/parameters is globally visible before the flag otherwise the consumer might read stale values.

Since the required ordering is between two stores (the last data store) and the store to the flag, the write ordering memory barrier is sufficient.

```
<<Update new data/parameters in system memory>>
sync_wmb                // or sync(0)
<<Update flag to say new data/parameters are ready to be consumed>>
```

Real world examples of this include:

- One processor thread setting up Programmed IO parameters for another processor thread that is going to do the programmed IO.
- A processor setting up descriptors in system memory for a DMA engine and writing a flag to indicate that the descriptor is ready.

Another scenario where the consumer might need a memory barrier is for FIFO buffers:

Producer Code:

```
Append buffer with new data
sync_wmb                              // or sync(0)
Update Tail pointer for buffer
```

Consumer Code:

```
Read oldest data from buffer
sync_mb                               // or sync(0)
Update Head pointer for buffer
```

Notice the Producer needs to order two stores while for the consumer its a load and store which must be ordered. For that reason, the producer can use the Store->Store memory barrier.

4.6.2.3 IO Device Registers with Side-Effects

Since writes to IO devices can have side-effects, when IO devices are shared among multiple CPUs, completion memory barriers are necessary to prevent interleaving of stores from multiple CPUs.

Here’s an example where the ADDR register selects which internal resource of the IO device will be accessed. The data for the internal resource is accessed by the DATA register. In this case, writing the ADDR register has the side-effect for the meaning of the DATA register.

Thread 1:

```
Lock(IO_Device)
Uncached Write IO_Device.ADDR = Transmit_Data
Uncached Write IO_Device.DATA = 0x1
sync(0)
UnLock(IO_Device)
```

Thread2:

```
Lock(IO_Device)
Uncached Write IO_Device.ADDR = Configuration
Uncached Write IO_Device.DATA = 0x7
sync(0)
UnLock(IO_Device)
```

The stores are done with the Uncached coherency/CCA=2. These are guaranteed by the architecture to be executed in program order within one processor.

If the completion memory barriers were missing, it would be possible that the two writes to the ADDR registers are done first before the writes to the DATA register and therefore the data can be written to the wrong internal resource of the IO device. For example, IO_Device.ADDR=Transmit_Data, then IO_Device.ADDR=Configuration then IO_Device.DATA=0x1.

The completion memory barriers are needed as the write data values need to be seen by the external IO device before ownership of the device is passed to the next requesting thread.

4.6.2.4 Dekker's Algorithm

Dekker's algorithm is a popular software algorithm used to provide mutual exclusion when there is no hardware support for atomic read-modify-write operations. Each processor sets it's own `*_wants_resource` variable when it desires to access the shared resource. It checks the equivalent variables for the other processors to decide if it has acquired the access rights to the resource. An additional variable `whose_turn` is used to preclude starvation.

The following pseudo-code example shows the algorithm for a system with two processors - P0 and P1. The code for P0 is shown, the code for P1 would swap the variables `P0_wants_resource` for `P1_wants_resource`; the value `P0_turn` for `P1_turn`.

```
P0_wants_resource=1
sync(0)
while (P1_wants_resource==1) {
    if (whose_turn != P0_turn) {
        P0_wants_resource = 0
        sync(0)
        while (whose_turn != P0_turn) {
               // waiting for P1 to update whose_turn
        }
        P0_wants_resource=1//P1 is now done
        sync(0)
    }
    // waiting for P1 to be done
}
<<Critical Section>>
sync_release ( or sync(0) )
whose_turn = P1_turn;
P0_wants_resource = 0;
sync(0)
```

The sync instructions after the stores to the *P0_wants_resource* flag and the *whose_turn* variable are necessary to make the store values globally visible. Otherwise, it is possible that both processors see stale data values and both enter the critical section simultaneously. In this case only the completion barrier will suffice as the local stores must be made visible to the other processor immediately.

Note to software engineers: This algorithm is historically used on systems that do not have synchronization primitives. Since the MIPS architecture does have these primitives (like LL/SC), this algorithm is presented as only an example where memory barriers would be needed, not as a solution to software synchronization.

4.6.2.5 Lamport's Bakery Algorithm

Lamport's Bakery algorithm is another software algorithm used to provide mutual exclusion when there are multiple requestors and there is no hardware support for atomic read-modify-write operations.

This algorithm is somewhat similar to what happens when boarding an commercial airliner. The analogy is when the passenger gets to their seat, it represents when the passenger gets the service/resource they are requesting.

The algorithm models a class scheme where some passengers have higher priority than others. For the airline those who paid for first-class or business class seats are in different classes from those who only paid for economy class. In a software project, this priority might be used for differentiate high priority threads/services from lower priority threads/services.

Besides their priority, each passenger receives their boarding group number. The boarding group number is first-come, first served. Those passengers who arrive early enough to be in one boarding group number gets to board the plane before passengers who arrived later and were assigned another boarding group number. In a software project, these boarding groups can represent threads/services at the same priority level but are differentiated by how long they have been waiting.

The algorithm deviates from the airplane boarding analogy in that the algorithm gives precedence to the boarding group number over the priority class.

The array InQueue[0:N-1] represents all of the possible passengers. If InQueue[x]==0 then it means passenger x did not show up in the airport and does not desire to board the plane. If InQueue[x]> 0 then that value is passenger x's position in the queue. A lower position in the array means higher priority (eg. Location 0 represents the highest priority passenger, location 1 represents the next highest priority passenger).The array GettingBoardingPass[0:N-1] is used to prevent loads from getting stale values of the InQueue array - eg. passenger Y is checking if passenger X is waiting in line while at the same time passenger X is getting his/her position in the queue.

The following pseudo-code is written for passenger i:

```
GettingBoardingPass[i] = 1
sync_wmb                // or sync(0)
InQueue[i] = << currently available boarding group number>>
sync_wmb                // or sync(0)
GettingBoardingPass[i] = 0
sync(0)
for (j=0; j < N; j++) {
    while ( GettingBoardingPass[j]) {
        // wait for other passenger to get their position in queue
    }
    while (InQueue[j]> 0) && (( InQueue[j] < InQueue[i]) |
           ((Inqueue[j] == InQueue[i]) & ( j < i) ) ){
        // waiting for passengers who are in the airport AND
        // 1. in an earlier boarding group OR
        // 2. in my boarding group but has higher priority
    }
}
<<Critical Section>>
sync_release            // or sync(0)
InQueue[i] = 0;
sync(0)
```

The updates to the GettingBoardingPass and InQueue arrays must be made globally visible in program order. Notice the use of the sync_wmb which can be used to order the sequence of stores and only the last store has to be followed by a completion memory barrier.

Note to software engineers: This algorithm is historically used on systems that do not have synchronization primitives. Since the MIPS architecture does have these primitives (like LL/SC), this algorithm is presented as only an example where memory barriers would be needed, not as a solution to software synchronization.

4.6.2.6 Lamport's Fast Mutual Exclusion Algorithm

Lamport derived a minimum sequence of regular load/store instructions which can detect mutual exclusion without special hardware primitives. Each processor in the system runs this minimum sequence of operations to arbitrate for shared resources.

This sequence uses two variables in shared memory - X and Y. The minimum sequence of instructions is:

1. write X
2. then read Y
3. then write Y
4. then read X.

Mutual exclusion is detected if the loads gets back the expected values. If the loads gets different values from the expected values then some other processor has run its version of this sequence and there are conflicting stores to either the X or Y shared variables.

The following pseudo-code example shows the algorithm for Processor i. The initial value for Y is 0. It doesn't matter what the initial value is for X. Mutual Exclusion is detected if the first load of Y gets the value of 0 and the first load of X gets the value of i.

```
Start:    X=i                 // 1. write X
          sync_mb             // or sync(0), needed to enforce write->read order
          if (Y != 0) {       // 2. read Y
             goto Start
          }
          Y = i               // 3. write Y
          sync_mb             // or sync(0), needed to enforce write->read order
          if (X != i) {       // 4. read X
             if (Y != i) {
                goto Start
             }
          }
          <<Critical Section>>
          sync_release        // or sync(0)
          Y = 0;              // release the shared variables
```

The transactions (1 followed by 2) and (3 followed by 4) must be ordered by the memory barriers as the second transaction in the pair uses a different cacheblock address than the first transaction. The ordering memory barriers are sufficient as the transaction ordering that must be maintained are within each processor. The transactions (2 followed by 3) do not need a memory barrier as the ordering is enforced by a WAR anti-dependency since the same address is used.

This pseudo-code example is meant to show examples where memory barriers are necessary and omits additional code in the algorithm that further aids in detecting contention.

Note to software engineers: This algorithm is historically used on systems that do not have synchronization primitives. Since the MIPS architecture does have these primitives (like LL/SC), this algorithm is presented as only an example where memory barriers would be needed, not as a solution to software synchronization.

4.6.2.7 Memory accessed with multiple Cacheability and Coherency Attributes

A memory location is first written with one Cacheability and Coherency Attribute (CCA), using virtual address 1. Subsequently, the same memory location is read back with a different CCA using virtual address 2. To ensure that the younger load sees the updated store value, a memory barrier is required.

```
<<Update physical address 0x20.0100 using VA=0x8820.0100 (cached address)>>
                                    (mapped by the TLB as write-thru).
sync_mb                 // or sync(0)
<<Read physical address 0x20.0100 using VA=0xa020.0100. (uncached address)>>
```

A real world example of this might be user mode software updating a hardware frame buffer and a kernel-mode device driver later reading the data to do further transforms.

Assuming the write and subsequent read are within the same processor, the ordering memory barrier is sufficient.

4.6.2.8 Context Switching

In a multiprocessor system, if a thread can be migrated from one processor to another, a sync(0) instruction needs to be executed by the original processor before the thread is made run-able on another processor. This is to ensure that all previous stores generated by the thread are visible to that other processor.

As mentioned in the previous section, if a memory location is accessed with one Cacheability and Coherency Attribute while in user mode and a different CCA while in kernel mode, a sync(0) is needed to ensure that the kernelmode software sees all of the previous store values created by the usermode code.

4.6.3 Implicit Memory Barriers

The MIPS architecture does not specify any implicit memory barriers.

- Entrance and exit from exception/error modes do not require implicit barriers.
- An earlier version of the architecture required that the LL/LLD/SC/SCD instructions implement implicit memory barriers. This is no longer required.
- It is implementation-specific if the CACHE instruction (and which operations) implement an implicit memory barrier.

4.7 Memory Coherency and Instruction Caches

In the MIPS architecture, the instruction caches are allowed to be non-coherent. For that reason, software must handle the case when code is to be modified but that code is already resident in the caches.

4.7.1 A processor modifying code for another processor

For this case:

1. All of the updated code must be made visible to any subsequent instruction fetches, probably with a SYNC(0) instruction following the last store operation that is updating the code.

2. Any stale copies of the code must be invalidated from the entire cache hierarchy on any processor which will execute the new code, using CACHE Hit_Invalidate_[I,S,T] or SYNCI instructions.

3. The recipient executes a JR.HB or JALR.HB instruction to access the updated code. These instructions are necessary to prevent any prefetching of the stale instructions while the invalidates are happening. Only then can the recipient processor start executing from the memory location holding the updated code.

Some implementations might choose to make the CACHE and SYNCI instructions broadcast the cache invalidations commands to all processors within the system. For such implementations, steps 1 and 2 can be done with the code sequence shown with the SYNCI instruction specification in the MIPS32/64 Instruction Set specification.

Other implementations that do not broadcast the cache invalidates can signal an interrupt so that each processor invalidates their local caches accordingly.

4.8 Requirements for the rest of the system.

4.8.1 IO Device Access

The Uncached Cacheability and Coherency Attribute (CCA=2) is the primary way IO devices are accessed in the MIPS architecture. Since the MIPS architecture does not separate the virtual memory map into IO device spaces and memory spaces, memory transactions using CCA=2 are always strongly ordered. This means:

- Younger memory accesses using CCA=2 are never allowed to be executed before older memory accesses using CCA=2. eg. No bypassing is allowed.
- Loads and Stores using CCA=2 are never speculative.

For this reason, the rest of the system must maintain these properties for processor initiated uncached accesses. This means that the system interconnect must not reorder uncached transactions once the transactions have left the processor subsystem on their way to the IO device.

4.8.2 System-level synchronization

In some systems, when a memory transaction has left the processor subsystem, it is still not guaranteed that the updated value is visible to all IO devices. For such systems, some mechanism that ensures such visibility would be helpful in implementing synchronization between the processor subsystem and the IO devices.

Some possibilities:

- A memory mapped register that forces the system interconnect to complete previously outstanding transactions before the register value is returned to the reader
- A memory mapped register that forces a specific IO device to complete previously outstanding transactions before the register value is returned to the reader.

Chapter A

Extensions to the OCP Port for Cache Coherent Systems

This appendix summarizes the mapping from the coherence framework to the Open Core Protocol. Further details of OCP extensions for cache coherence protocols may be found in the OCP-IP document *OCP Coherence Extensions.*

A.1 Additional Ports and Signals for Coherence

A.1.1 Augmented Main Memory Port

The main port of the master is augmented with additional signals which are detailed in Table A.1. New coherence transactions are also added to the transaction set and these are described in Section A.2 "New set of Coherence Transactions". This port also can accept a new response from the interconnect.

Table A.1 Augmented Main Memory Port Signals

<table>
<tr><th>Signal Name</th><th colspan="4">Description</th></tr>
<tr><td>MCohCmd</td><td colspan="4">A non 0x0 value of this signal indicates that this request is a coherent request as opposed to a legacy OCP transaction.</td></tr>
<tr><td>MCmd[4:3]</td><td colspan="4">The OCP MCmd signal group is extended by two bits, to allow the extended coherent command set to be encoded.</td></tr>
<tr><td>SCohState</td><td colspan="4">This signal indicates the install state which the requestor is expected to mark the cache line when the data is filled into the cache. It is part of the response phase, and is passed back to the master with any response to a coherent request. For legacy requests, this signal is a don't care. The encoding of this signal field is detailed below.
<table>
<tr><th>SCohState</th><th>Install State</th><th>Mnemonic</th><th>Description</th></tr>
<tr><td>0x0</td><td>Invalid</td><td>I</td><td>Cache line not present</td></tr>
<tr><td>0x1</td><td>Shared</td><td>S</td><td>Cache line in more than 1 agent with read only capability</td></tr>
<tr><td>0x2</td><td>Modified</td><td>M</td><td>Only cache line in system with the latest data</td></tr>
<tr><td>0x3</td><td>Exclusive</td><td>E</td><td>Exclusive cached copy</td></tr>
<tr><td>0x4-0x5</td><td>Reserved</td><td>-</td><td>-</td></tr>
<tr><td>0x6</td><td>Owned</td><td>O</td><td>Cached in more than 1 agent and this agent has the latest copy while memory has a stale copy</td></tr>
<tr><td>0x7</td><td>Migratory</td><td>T</td><td>Indicates to the requestor that the cache line is migratory</td></tr>
</table></td></tr>
</table>

Table A.1 Augmented Main Memory Port Signals

<table>
<tr><th>Signal Name</th><th colspan="4">Description</th></tr>
<tr><td>SResp</td><td colspan="4">A new response, “OK”, is added to the legacy responses. This response is used by both the intervention port as well as the system to indicate to the master an ack without a data transfer. All the encodings are shown below for convenience.</td></tr>
<tr><td></td><td>SResp</td><td>Response</td><td>Mnemonic</td><td>Description</td></tr>
<tr><td></td><td>0x0</td><td></td><td>NULL</td><td>Legacy Response</td></tr>
<tr><td></td><td>0x1</td><td>Data Valid</td><td>DVA</td><td>Legacy Response</td></tr>
<tr><td></td><td>0x2</td><td>Request Failed</td><td>FAIL</td><td>Legacy Response</td></tr>
<tr><td></td><td>0x3</td><td>Error Response</td><td>ERR</td><td>Legacy Response</td></tr>
<tr><td></td><td>0x4</td><td>Ack without Data Transfer</td><td>OK</td><td>New Response</td></tr>
</table>

A.1.2 Added Intervention Port

A read-only Intervention port is added to coherent agents which have cache state. This port is a *dual* to the main port, meaning that if the main port is a master port this port is a slave port and vice-versa. This port is required of all coherent agents which cache memory locations, which is generally the case with master agents but may be the case with slave agents if they maintain cache state, e.g. a directory. Coherent masters which do not cache data can dispense with the intervention port, but it should be noted that I/O adapters which do not cache memory lines in the conventional sense may still require an intervention port to support partial-line writes. All transactions on the intervention ports receive a response, which is used to send cacheline state to the coherency master.

There is an OCP port option where Write type transactions are *collapsed* at the master’s intervention port, so that Single Request Multiple Data (SRMD) write type transactions neither send data on the master port nor provide any data to the intervention port. Instead the data is sent out as a response to the self-intervention on the intervention port. This option aids in having local writebacks be ordered among external coherent traffic.

Table A.2 shows the intervention port signals. In accordance with OCP convention, signals prefixed by M are driven by the Master and signals prefixed by S are driven by the Slave. Note that the intervention channel is driven by coherent memory agents and received by the processor agents - in that sense they are the reverse of the main port.

Table A.2 Intervention Port Signals (OCP signals assumed)

Group	Allowed Signals	Comments
Basic	Clk, MAddr, **MCmd**, MRespAccept, SCmdAccept, SData, **SResp**	This group of signals are needed for a *basic* OCP port and are identical to their main port counterparts except for the lack of MData, MDataValid signals which are not allowed because the intervention port is read-only. **MCmd** and **SResp** are augmented with extra encoding(s).
Simple	SDataInfo, MReqInfo, SRespInfo	Only these three signals from the *simple* group of the OCP signal group are allowed. Others in this group, i.e. MAddrSpace, MByteEn, MDataByteEn, MDataInfo are not.

Table A.2 Intervention Port Signals (OCP signals assumed)

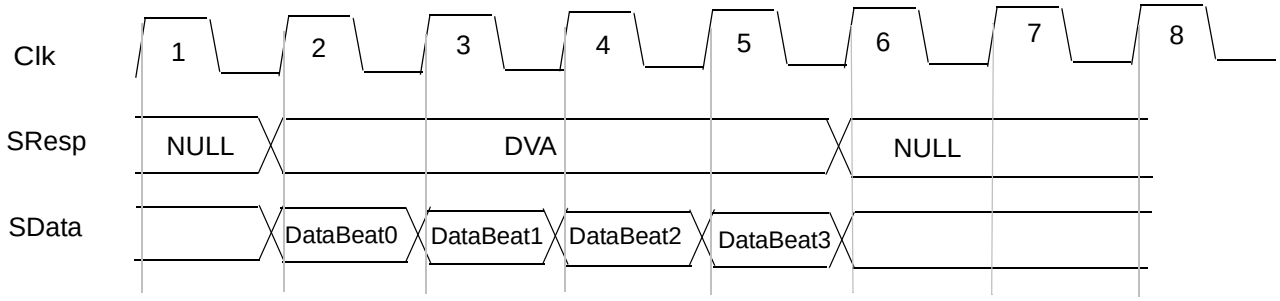
Group	Allowed Signals	Comments
Burst	MAtomicLength, MBurstLength, MBurstPrecise, MBurstSingleReq, SRespLast, ***SDataLast***	This group of signals from the burst group (for burst transactions) have the following fixed parameters: MAtomicLength = Cache line size, MBurstLength = Cache line size, MBurstPrecise = 1, MBurstSeq = {INCR, XOR, WRAP} only, MBurstSingleReq = 1. MDataLast, MReqLast are not allowed. Signals in italics have been added to allow a split intervention response.
Thread	MThreadBusy, MThreadID, SThreadBusy, SThreadID, ***SDataThreadID***, ***MDataThreadBusy***	These group of signals allow multiple threads to be implemented on a port. MConnID, MDataThreadID and SDataThreadBusy are not allowed due to this being a read-only port. Signals in italics have been added to allow a split intervention response.
Tags	MTagID, STagID, MTagInOrder, STagInOrder, ***SDataTagID***	This group of signals allow multiple tags to be implemented on a port. MDataTagID is not allowed due to this being a read-only port. Signals in italics have been added to allow a split intervention response
Coherence		This is a new group of signals added for coherency.
	SCohState	Required Signal. It is used to indicate the install state of a cache line. It is part of the intervention response phase. Its encodings are exactly the same as described in Table A.3 for the Main port signal with the same name.
	MDataAccept	This is available in the case of a split intervention response to flow control return data on an intervention request in this cycle.
	SDataValid	This is available in the case of a split intervention response to indicate the availability of data in this cycle.
	MReqSelf	Required Signal. This is available only on the intervention port as an input. It is valid when an MCmd is not IDLE. It indicates to the intervention slave that this intervention request is a result of a main port request which originated from the master port of this agent (And is hence a self-intervention). This concept of self-intervention is critical as it allows the agents to enforce a global order in a coherent system.
	MCohID	Optional (Used for Directory based scheme). It specifies the target of the request. It is valid when an MCmd is not IDLE. It is used at a *master intervention port* to indicate the target of a intervention request.
	MCohFwdID	Optional (Used for Directory based scheme). It specifies the target for a three party transaction. It is valid when an MCmd is not IDLE. It is used at a *master intervention port* to signal to the target that a three party transaction is required and this is the address of the originating request.
	SCohID	Optional (Used for Directory based scheme). It specifies the target of the intervention response. It is valid when an MCmd is not IDLE. It is used at a *slave intervention port*.
Sideband	SReset_n, MReset_n	These signals are required.

To allow for a *split intervention response*, signals shown in italics in Table A.2 have been added to the intervention port. A split intervention allows for a faster response throughput when the latency to access data is longer than it is to supply the state of the cache line. As shown in Figure A.1 (a) a hit on an intervention results in blocking the interven-

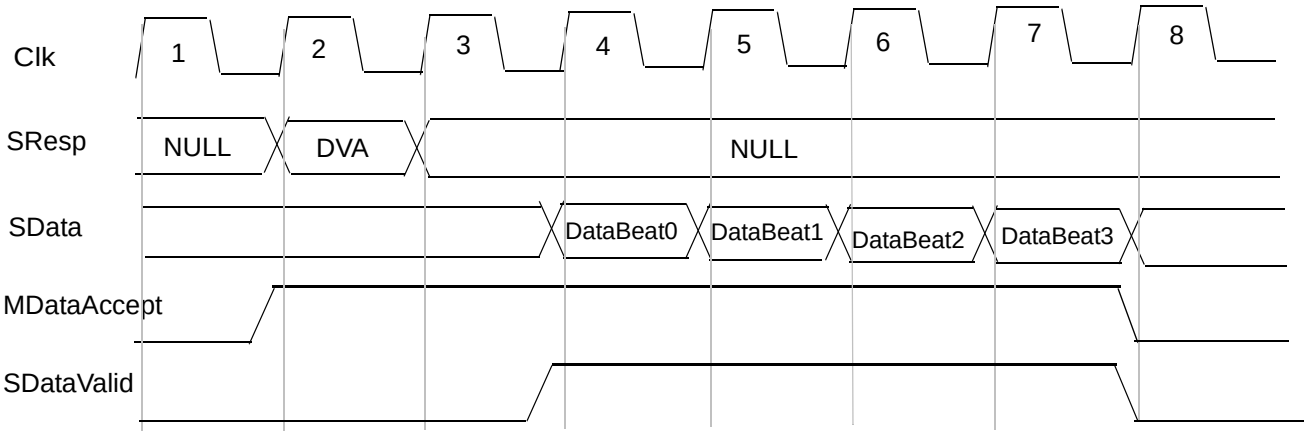
tion pipeline for 3 more cycles, while in (b) due to the split nature of the intervention response, the response pipeline flows independantly of the data response in case an access hits.

Figure A.1 Split Intervention Response

(a) Without Split Intervention Response



(b) With Split Intervention Response



A.2 New set of Coherence Transactions

The set of new transactions is listed in the Table A.3 below. The older OCP transactions are termed *legacy* transactions and need to be supported. The new transactions are briefly described below. All of these transactions can only be in the SRMD format. Coherent Write transactions on the main port may or may not have a response (Non-posted or Posted). All of these transactions can potentially be observed at the master port. However the set visible at the intervention port of a coherent master depends on the system, for example the location of the directory. For a pure snoopy scheme these transactions will be reflected by the interconnect and broadcast to all the intervention ports in the system, in which case the entire set of transactions emanating at the master port will be observed at the interven-

tion port. For a scheme with a dual set of tags hidden in the interconnect (or a pure directory scheme) a reduced set of these transactions could appear on the intervention port.

Table A.3 Extended Coherence Transactions

{MCohCmd, MCmd[4:0]}	Command	Mnemonic (Main / Intervention) Port	Request Type
{0x0, Dont'care}	Legacy	Legacy	Legacy
{0x1, 0x8}	**CohReadOwn**	C_RO/I_RO	Coherent Cached Read to Own - used for CPU store misses.
{0x1, 0x9}	**CohReadShare**	C_RS/I_RS	Coherent Cached Read to Share - used for CPU load misses.
{0x1, 0xA}	**CohReadDiscard**	C_RD/I_RD	Coherent Cached Read to Discard - used for reads by IO devices without caches.
{0x1, 0xB}	**CohReadShareAlways**	C_RSA/I_RSA	Coherent Cached Read to Share Always - used for instruction fetches from coherent icaches and for reads by IO devices with caches.
{0x1, 0xC}	**CohUpgrade**	C_UPG/I_UPG	Coherent Cached Upgrade - used to change cache-line state to Modified from Exclusive for CPU store hits.
{0x1, 0xD}	**CohWriteBack**	C_WB/I_WB	Coherent Cached Write Back - used for CPU evictions of dirty cachelines.
{0x1, 0xE}	**Reserved**	-	-
{0x1, 0xF}	**Reserved**	-	-
{0x1, 0x10}	**CohCopyBack**	C_CB/I_CB	Coherent Cached Copyback - used for cache maintenance.
{0x1, 0x11}	**CohCopyBackInval**	C_CBI/I_CBI	Coherent Cached Copyback and Invalidate - used for cache maintenance.
{0x1, 0x12}	**CohInvalidate**	C_INV/I_INV	Coherent Cached Invalidate - used for flushing caches for IO writes.
{0x1, 0x13}	**CohWriteInvalidate**	C_WINV/I_WINV	Coherent Cached Write Invalidate - used to update memory with IO write data and flusing caches of stale data.
{0x1, 0x14}	**CohCompletionSync**	C_CCS/I_CCS	Coherent Completion Sync - used to enforce transaction ordering
{0x1, 0x15}	**MessageSend**	MSG/I_MSG	Message Send (Can be used to send packetized Interrupts)
{0x1, 0x16}	**Reserved**	-	-
{0x1, 0x17}	**Reserved**	-	-

The CohWriteBack transaction appears only as a self-intervention and need not be broadcast, even in a snoopy scheme.

The CohWriteInvalidate transaction has *copyback* semantics with respect to caches at or below it in the hierarchy when it is sub-cache-line sized. When it is cache line sized, it has *invalidate* semantics.

Chapter C

Revision History

In the left hand page margins of this document you may find vertical change bars to note the location of significant changes to this document since its last release. Significant changes are defined as those which you should take note of as you use the MIPS IP. Changes to correct grammar, spelling errors or similar may or may not be noted with change bars. Change bars will be removed for changes which are more than one revision old.

Please note: Limitations on the authoring tools make it difficult to place change bars on changes to figures. Change bars on figure titles are used to denote a potential change in the figure itself. Certain parts of this document (Instruction set descriptions) are references to Architecture specifications, and the change bars within these sections indicate alterations since the previous version of the relevant Architecture document.

Revision	Date	Description
00.95	November 20,2007	First version with spin-off of MP Platform Specification.
00.96	December 5, 2007	Clean-up for EA.
00.97	April 28, 2008	Chapter 3 - Added example to further explain expectations of lighter-weight barriers. Clearer explanation of lighter-weight Syncs. Added SYNC_RMB.
00.98	June 04, 2008	All Chapters - review comments Chapter 4 - added some more illegal & legal sequences.
01.00	June 25, 2008	* Clean-up for GA. * Chapter 3 - make SYNC description match up to VolumeII-BIS
01.01	September 14, 2015	Branded document.