Cache Organization, Operation, and Coherency

11

This chapter describes in detail the cache memory: its place in the R4000 memory organization, individual operations of the primary and secondary caches, cache interactions, and an example of a cache coherency request cycle. The chapter concludes with a description of R4000 processor synchronization in a multiprocessor environment.

This chapter uses the following terminology:

- The primary cache may also be referred to as the P-cache.
- The secondary cache may also be referred to as the S-cache.
- The primary data cache may also be referred to as the D-cache.
- The primary instruction cache may also be referred to as the I-cache.

These terms are used interchangeably throughout this book.

11.1 Memory Organization

Figure 11-1 shows the R4000 system memory hierarchy. In the logical memory hierarchy, caches lie between the CPU and main memory. They are designed to make the speedup of memory accesses transparent to the user. Each functional block in Figure 11-1 has the capacity to hold more data than the block above it. For instance, physical main memory has a larger capacity than the secondary cache. At the same time, each functional block takes longer to access than any block above it. For instance, it takes longer to access data in main memory than in the CPU on-chip registers.

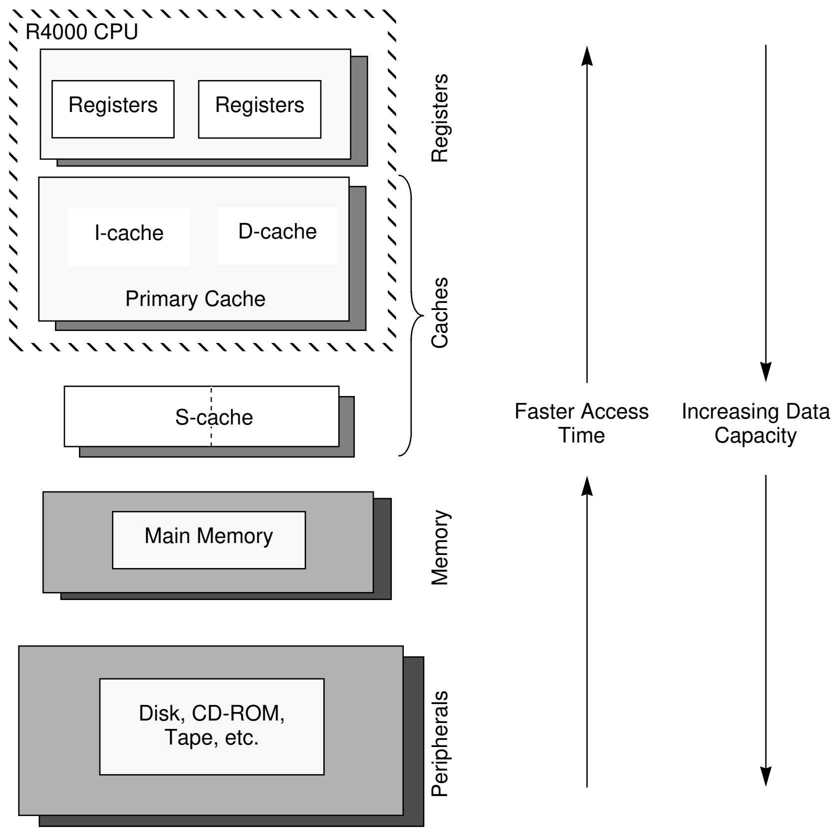


Figure 11-1 Logical Hierarchy of Memory

The R4000 processor has two on-chip primary caches: one holds instructions (the instruction cache), the other holds data (the data cache). Off-chip, the R4000 processor supports a secondary cache on the R4000SC and MC models.

11.2 Overview of Cache Operations

As described earlier, caches provide fast temporary data storage, and they make the speedup of memory accesses transparent to the user. In general, the processor accesses cache-resident instructions or data through the following procedure:

1. The processor, through the on-chip cache controller, attempts to access the next instruction or data in the primary cache.
2. The cache controller checks to see if this instruction or data is present in the primary cache.
   - If the instruction/data is present, the processor retrieves it. This is called a primary-cache *hit*.
   - If the instruction/data is not present in the primary cache, the cache controller must retrieve it from the secondary cache or memory. This is called a primary-cache *miss*.
3. If a primary-cache miss occurs, the cache controller checks to see if the instruction/data is in the secondary cache.
   - If the instruction/data is present in the secondary cache, it is retrieved and written into the primary cache.
   - If the instruction/data is not present in the secondary cache, it is retrieved as a cache line (a block whose size set in the *Config* register; see the section titled Variable-Length Cache Lines in this chapter for available cache line lengths) from memory and is written into both the secondary cache and the appropriate primary cache.
4. The processor retrieves the instruction/data from the primary cache and operation continues.

It is possible for the same data to be in three places simultaneously: main memory, secondary cache, and primary cache. This data is kept consistent through the use of write back methodology; that is, modified data is not written back to memory until the cache line is replaced.

11.3 R4000 Cache Description

As Figure 11-1 shows, the R4000 contains separate primary instruction and data caches. Figure 11-1 also shows that the R4000 supports a secondary cache that can be split into separate portions, one portion containing data and the other portion containing instructions, or it can be a joint cache, holding combined instructions and data.

This section describes the organization of on-chip primary caches and the optional off-chip secondary cache. Table 11-1 lists the cache and cache coherency support for the three R4000 models.

Table 11-1 R4000 Cache and Coherency Support

R4000 Model	Support Primary Cache?	Support Secondary Cache?	Support Cache Coherency?
R4000PC	Yes	No	No
R4000SC	Yes	Yes	No
R4000MC	Yes	Yes	Yes

Figure 11-2 provides block diagrams of the three R4000 models:

- R4000PC, which supports only the primary cache
- R4000SC and R4000MC, which support both primary and secondary caches

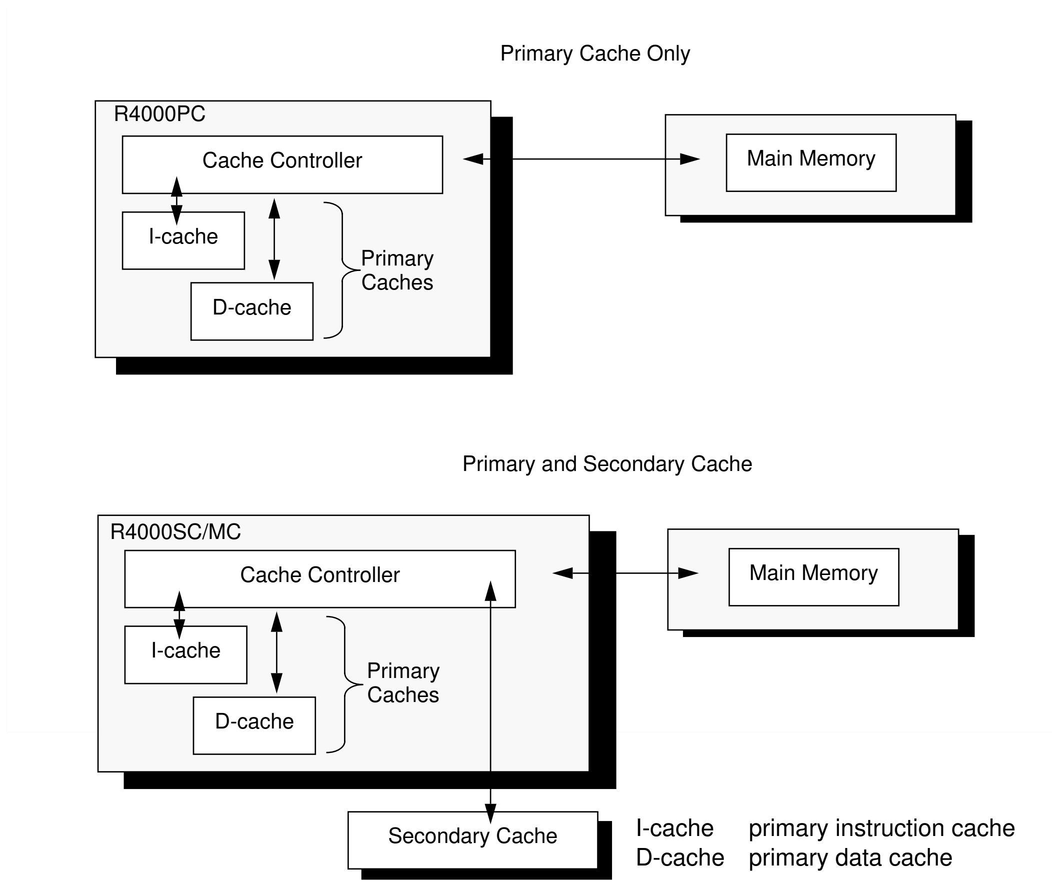


Figure 11-2 Cache Support in the R4000PC, R4000SC, and R4000MC

Secondary Cache Size

Table 11-2 lists the range of secondary cache sizes. The secondary cache is user-configurable at boot time through the boot-mode bits (see Chapter 9); it can be a joint cache, containing both data and instructions in a single cache, or split into separate data and instruction caches.

Table 11-2 Secondary Cache Sizes

Cache	Minimum Size	Maximum Size
Secondary Joint Cache	128 Kbytes	4 Mbytes
Secondary Split I-Cache	128 Kbytes	2 Mbytes
Secondary Split D-Cache	128 Kbytes	2 Mbytes

Variable-Length Cache Lines

A *cache line* is the smallest unit of information that can be fetched from the cache, and that is represented by a single tag.† A primary cache line can be either 4 or 8 words in length; a secondary cache line can be either 4, 8, 16, or 32 words in length. Primary cache line length is set in the *Config* register; see Chapter 4 for more information. Secondary cache line length is set at boot time through the boot-mode bits, as described in Chapter 9.

Upon a cache miss in both primary and secondary caches, the missing secondary cache line is loaded first from memory into the secondary cache, whereupon the appropriate subset of the secondary cache line is loaded into the primary cache.

The primary cache line length can never be longer than that of the secondary cache; it must always be less than or equal to the secondary cache line length. This means the secondary cache cannot have a 4-word line length while the primary cache has an 8-word line length.

Cache Organization and Accessibility

This section describes the organization of the primary and secondary caches, including the manner in which they are mapped, the addressing (either virtual or physical) used to index the cache, and composition of the cache lines. The primary instruction and data caches are indexed with a virtual address (VA), while the secondary cache is indexed with a physical address (PA).

† Primary and secondary cache tags are described in the following sections.

Organization of the Primary Instruction Cache (I-Cache)

Each line of primary I-cache data (although it is actually an instruction, it is referred to as data to distinguish it from its tag) has an associated 26-bit tag that contains a 24-bit physical address, a single valid bit, and a single parity bit. Byte parity is used on I-cache data.

The R4000 processor primary I-cache has the following characteristics:

- direct-mapped
- indexed with a virtual address
- checked with a physical tag
- organized with either a 4-word (16-byte) or 8-word (32-byte) cache line.

Figure 11-3 shows the format of an 8-word (32-byte) primary I-cache line.

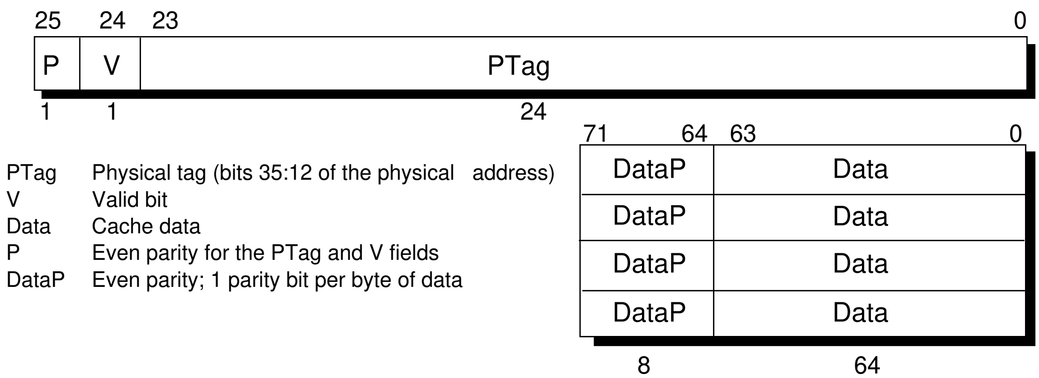


Figure 11-3 R4000 8-Word Primary I-Cache Line Format

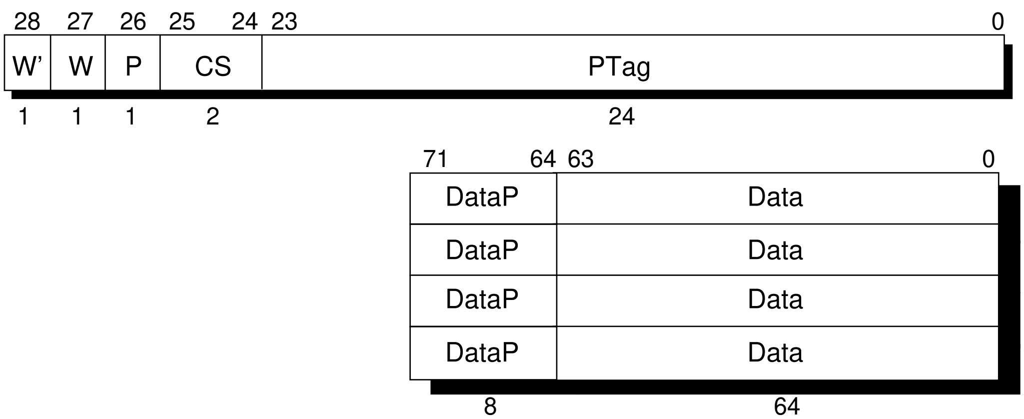
Organization of the Primary Data Cache (D-Cache)

Each line of primary D-cache data has an associated 29-bit tag that contains a 24-bit physical address, 2-bit cache line state, a write-back bit, a parity bit for the physical address and cache state fields, and a parity bit for the write-back bit. Byte parity is used on D-cache data.

The R4000 processor primary D-cache has the following characteristics:

- write-back
- direct-mapped
- indexed with a virtual address
- checked with a physical tag
- organized with either a 4-word (16-byte) or 8-word (32-byte) cache line.

Figure 11-4 shows the format of a 8-word (32-byte) primary D-cache line.



W' Even parity for the write-back bit
W Write-back bit (set if cache line has been written)
P Even parity for the PTag and CS fields
CS Primary cache state:
0 = Invalid in all R4000 configurations
1 = Shared (either Clean or Dirty) in R4000MC configuration only
2 = Clean Exclusive in R4000SC and MC configurations only
3 = Dirty Exclusive in all R4000 configurations
PTag Physical tag (bits 35:12 of the physical address)
DataP Even parity for the data
Data Cache data

Figure 11-4 R4000 8-Word Primary Data Cache Line Format

In all R4000 processors, the *W* (write-back) bit, not the cache state, indicates whether or not the primary cache contains modified data that must be written back to memory or to the secondary cache.

Accessing the Primary Caches

Figure 11-5 shows the virtual address (VA) index into the primary caches. Each instruction and data cache range in size from 8 Kbytes to 32 Kbytes; therefore, the number of virtual address bits used to index the cache depends on the cache size. For example, VA(12:4) accesses a 8-Kbyte page tag in a cache with a 4-word line (VA(12) addresses 8 Kbytes and VA(4) provides quadword resolution); similarly, VA(14:5) accesses an 8-word tag: VA(5) provides octalword access in a 32-Kbyte cache (VA(14) addresses 32 Kbytes).

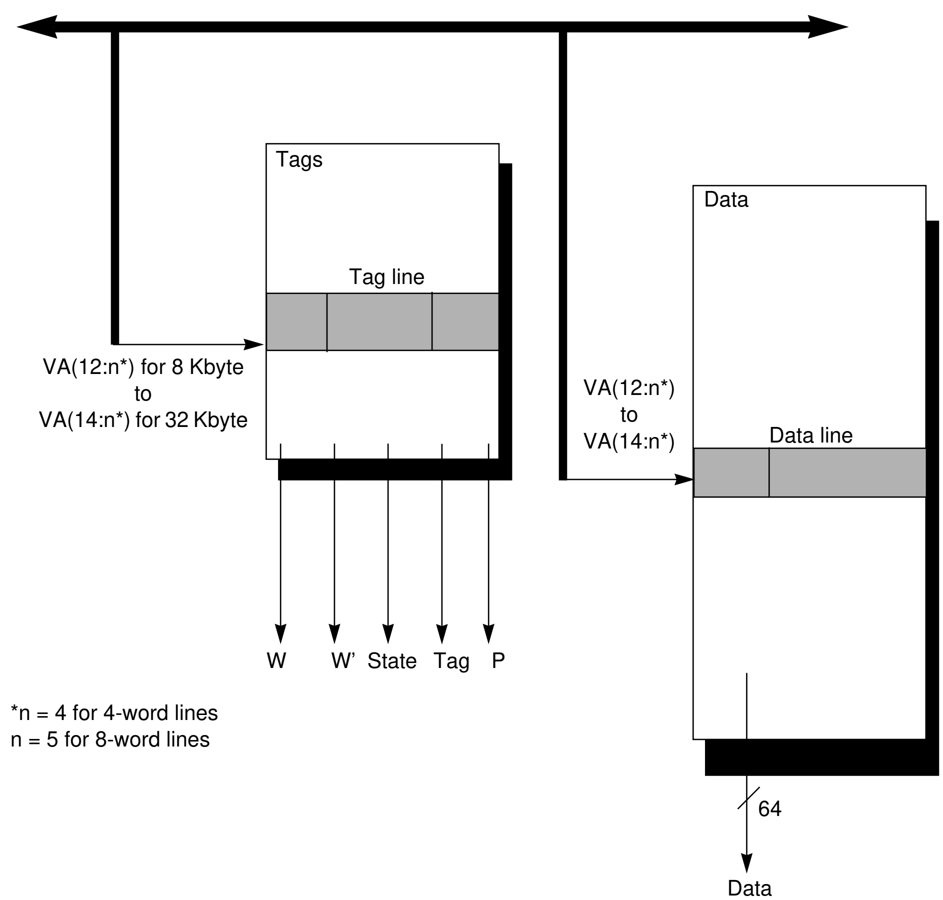


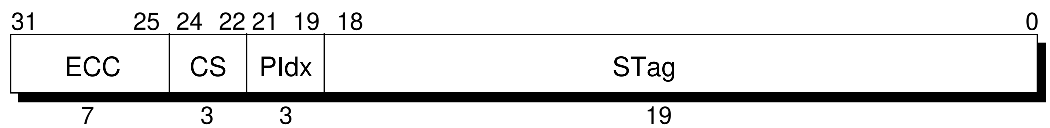
Figure 11-5 Primary Cache Data and Tag Organization

Organization of the Secondary Cache

Each secondary cache line has an associated 19-bit tag that contains bits 35:17 of the physical address, a 3-bit primary cache index, VA(14:12), and a 3-bit cache line state. These 25 bits are protected by a 7-bit ECC code.

The secondary cache is accessible to the processor and to the system interface; by setting the appropriate boot-mode bits, it can be configured at chip reset as a joint cache, or as separate I- and D-caches.

Figure 11-6 shows the format of the R4000 processor secondary-cache line. The size of the secondary cache line is set in the *SB* field of the *Config* register.



ECC ECC for secondary tag
CS Secondary-cache state
- 0 = Invalid
- 1 = reserved
- 2 = reserved
- 3 = reserved
- 4 = Clean Exclusive
- 5 = Dirty Exclusive
- 6 = Shared
- 7 = Dirty Shared

PIdx Primary cache index (bits 14:12 of the virtual address)
STag Physical tag (bits 35:17 of the physical address)

Figure 11-6 R4000 Secondary Cache Line Format

The R4000 processor secondary cache has the following characteristics:

- write-back
- direct-mapped
- indexed with a physical address
- checked with a physical tag
- organized with either a 4-word (16-byte), 8-word (32-byte), 16-word (64-byte), or 32-word (128-byte) cache line.

The secondary cache state (*CS*) bits indicate whether:

- the cache line data and tag are valid
- the data is *potentially* present in the caches of other processors (shared versus exclusive)
- the processor is responsible for updating main memory (clean versus dirty).

The *PIdx* field provides the processor with an index to the virtual address of primary cache lines that may contain data from the secondary cache line.

The *PIdx* field also detects a cache alias. Cache aliasing occurs when the physical address tag matches during a data reference to the secondary cache, but the *PIdx* field does not match in the virtual address. This indicates that the cache reference was made from a different virtual address than the one that created the secondary-cache line, and the processor signals a Virtual Coherency exception.

Accessing the Secondary Cache

Figure 11-7 shows the physical address (PA) index into the secondary cache. The secondary cache ranges in size from 128 Kbytes to 4 Mbytes, and the number of physical address bits used to index the cache depends upon the cache size. For instance, PA(16:4) accesses the tags in a 128-Kbyte secondary cache with 4-word lines; PA(21:5) accesses the tags in a 4-Mbyte secondary cache with 8-word lines.

The processor always uses PA(35:17) from the secondary cache, regardless of the S-cache size. This makes it important to initialize all secondary cache tag address bits with a valid physical address, regardless of the size of the S-cache.

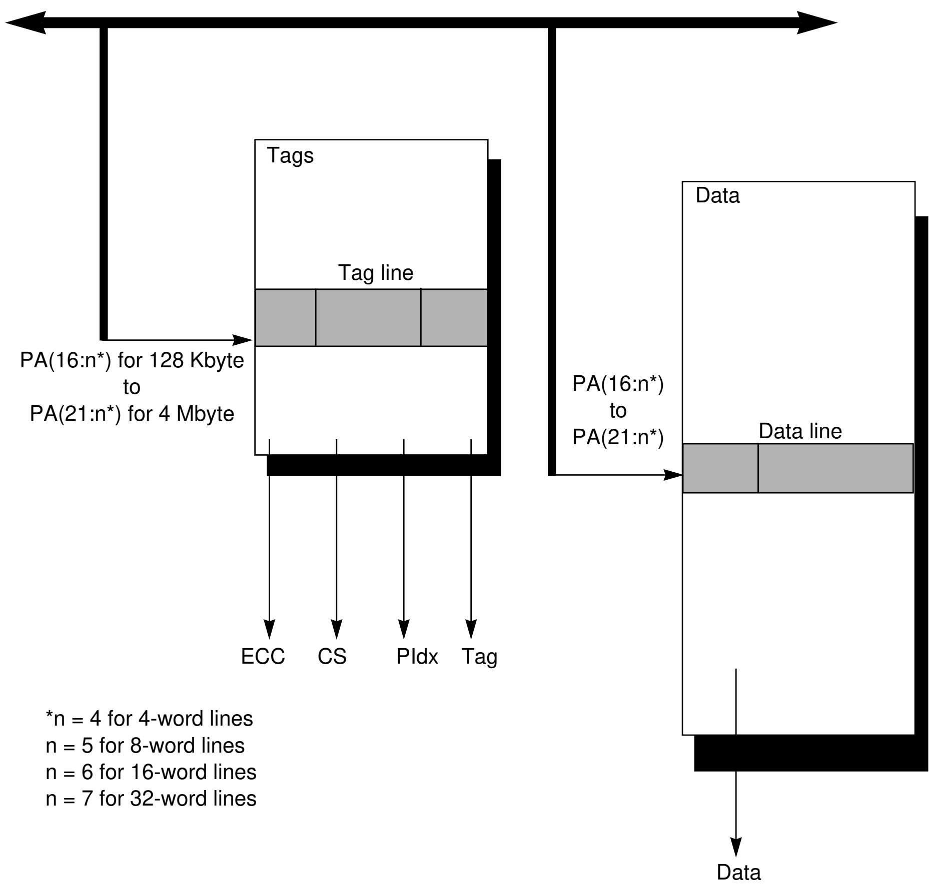


Figure 11-7 Secondary Cache Data and Tag Organization

11.4 Cache States

The four terms below are used to describe the *state* of a cache line:

- **Exclusive**: a cache line that is present in exactly one cache in the system is exclusive, and may be in one of the exclusive states.
- **Dirty**: a cache line that contains data that has changed since it was loaded from memory is dirty, and must be in one of the dirty or shared states.
- **Clean**: a cache line that contains data that has not changed since it was loaded from memory is clean, and may be in one of the clean states.
- **Shared**: a cache line that is present in more than one cache in the system.

Each primary and secondary cache line in the R4000 system is in one of the states described in Table 11-3. Table 11-3 also lists with the types of cache and the R4000 models in which the various states may be found.

Table 11-3 Cache States

Cache Line State	Description	Where the State is Used	Available on the Following R4000 Models
Invalid	A cache line that does not contain valid information must be marked invalid, and cannot be used. For example, a cache line is marked invalid if the same information, located in another cache, is modified. A cache line in any other state than invalid is assumed to contain valid information.	Primary or Secondary Cache	R4000PC R4000SC R4000MC
Shared	A cache line that is present in more than one cache in the system is shared.	Primary or Secondary Cache	R4000MC only
Dirty Shared	A dirty shared cache line contains valid information and can be present in another cache. This cache line is inconsistent with memory and is owned by the processor (see the section titled Cache Line Ownership in this chapter).	Secondary cache only	R4000MC only

Table 11-3 (cont.) Cache States

Cache Line State	Description	Where the State is Used	Available on the Following R4000 Models
Clean Exclusive	A clean exclusive cache line contains valid information and this cache line is not present in any other cache. The cache line is consistent with memory and is not owned by the processor (see the section titled Cache Line Ownership in this chapter).	Primary or Secondary Cache	R4000SC R4000MC
Dirty Exclusive	A dirty exclusive cache line contains valid information and is not present in any other cache. The cache line is inconsistent with memory and is owned by the processor (see the section titled Cache Line Ownership in this chapter).	Primary or Secondary Cache	R4000PC R4000SC R4000MC

Primary Cache States

Each primary data cache line is in one of the following states:

- invalid
- shared
- clean exclusive
- dirty exclusive

Each primary instruction cache line is in one of the following states:

- invalid
- valid

Secondary Cache States

Each secondary cache line is in one of the following states:

- invalid
- shared
- dirty shared
- clean exclusive
- dirty exclusive

Mapping States Between Caches

Secondary cache states correspond, or map, to primary cache states (this mapping is listed in Table 11-6, later on in this chapter). For example, the secondary cache shared and dirty shared states map to the primary cache shared state.

Therefore, when the primary cache line is filled from the secondary cache, the state of the secondary cache line is also mapped into the primary cache; in the case described above, the shared or dirty shared secondary state is mapped to the shared primary cache state.

As shown in Figure 11-8, a primary cache line in the R4000PC model can be in either an invalid or dirty exclusive state. In the R4000SC model, a primary cache line can be in the invalid, clean exclusive, or dirty exclusive state. In the R4000MC model, the primary cache line can be invalid, clean exclusive, dirty exclusive, or shared.

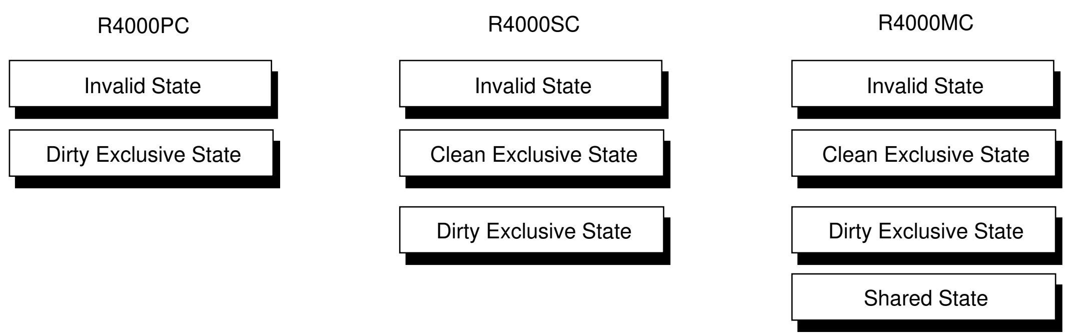


Figure 11-8 Primary Cache States Available to Each Type of Processor

11.5 Cache Line Ownership

A processor becomes the owner of a cache line after it writes to that cache line (that is, by entering the dirty exclusive or dirty shared state), and is responsible for providing the contents of that line on a read request. There can only be one owner for each cache line.

The ownership of a cache line is set and maintained through the rules described below.

- A processor assumes ownership of the cache line if the state of the secondary cache line is dirty shared or dirty exclusive.
- A processor that owns a cache line is responsible for writing the cache line back to memory if the line is replaced during the execution of a Write-back or Write-back Invalidate cache instruction. For *read responses* to a processor *coherent read request* (both of these terms are defined in Chapter 12) in which the data is returned in the dirty shared or dirty exclusive state, the cache state is set when the last word of read response data is returned. Therefore, the processor assumes ownership of the cache line when the last word of response data is returned.
- For processor coherent write requests, the state of the cache line changes to invalid if the cache line is replaced, or to either clean exclusive or shared if the cache line is retained (provided the cache line was written back to memory). In either case, the cache state transition occurs when the last word of write data is transmitted to the external agent. Therefore, the processor gives up ownership of the cache line when the last word of write data is transmitted to the external agent (Chapter 12 defines *external agent*).
- Memory always owns clean cache lines.
- The processor gives up ownership of a cache line when the state of the cache line changes to invalid, shared, or clean exclusive.

11.6 Cache Write Policy

The R4000 processor manages its primary and secondary caches by using a *write-back* policy; that is, it stores write data into the caches, instead of writing it directly to memory.† Some time later this data is independently written into memory. In the R4000 implementation, a modified cache line is not written back to memory until the cache line is replaced either in the course of satisfying a cache miss, or during the execution of a Write-back CACHE instruction.

If a primary cache line is in either the dirty exclusive or shared state and that cache line has been modified (the *W* bit is set), the processor writes this cache line back to memory (or the secondary cache, if it is present) when the line is replaced, either in the course of satisfying a cache miss or during the execution of a Write-back or Write-back Invalidate CACHE instruction.

If a secondary cache line is in either the dirty exclusive or dirty shared state, the processor writes this cache line back to memory when the line is replaced, either in the course of satisfying a cache miss or during the execution of a Write-back CACHE instruction.

Many systems, in particular multiprocessor systems, or systems employing I/O devices that are capable of DMA, require the system to behave as if the caches are always consistent both with memory and with each other. Schemes for maintaining consistency between more than one cache, or between caches and memory, are defined by the system cache coherency protocols (see the section titled Cache Coherency Overview later in this chapter). In the R4000 system, when the content of a cache line is inconsistent with memory, it is classified as dirty and is written back to memory according to the rules of the cache write-back policy.

When the processor writes a cache line back to memory, it does not ordinarily retain a copy of the cache line, and the state of the cache line is changed to invalid. However, there is one exception. The processor retains a copy of the cache line if a cache line is written back by the Hit Write-back cache instruction. The processor changes the retained cache line state to either clean exclusive if the secondary cache state was dirty exclusive before the write, or shared if the secondary cache state was dirty shared before the write. The processor signals this line retention during a write by setting **SysCmd(2)** to a 1, as described in Chapter 12.

† An alternative to this is a *write-through* cache, in which information is written simultaneously to cache and memory.

11.7 Cache State Transition Diagrams

The following sections describe the cache state diagrams that illustrate the cache state transitions for both the primary and secondary caches. Figures 11-9 and 11-10 are state diagrams of the primary and secondary caches, respectively.

When an external agent supplies a cache line, the initial state of the cache line is specified by the external agent (see Chapter 12 for a definition of an external agent). Otherwise, the processor changes the state of the cache line during one of the following events:

- A store to a clean exclusive cache line causes the state to be changed to dirty exclusive in both the primary and secondary caches.
- A store to a shared cache line—that is a line marked shared in the primary cache and either shared or dirty shared in the secondary cache—causes the processor to issue either an invalidate request or an update request (depending on the coherency attribute in the TLB entry for the page containing the cache line). And update page attribute causes an update request to be issued; a sharable page attribute causes an invalidate request to be issued.
    - Upon successful completion of an invalidate, the processor completes the store and changes the state of the cache line to dirty exclusive in both the primary and secondary caches.
    - Upon successful completion of an update, the processor completes the store and changes the state of the cache line to shared in the primary cache and dirty shared in the secondary cache if dirty shared mode is enabled. Dirty shared mode is programmable through the boot-time mode control interface (see Chapter 9 for a description of boot mode bits). If dirty shared mode is not enabled, the state of the primary and secondary caches are left in a shared state, after successful completion of an update.
- A store to a dirty exclusive line remains in a dirty exclusive state.

These state diagrams do not cover the initial state of the system since the initial state is system dependent.

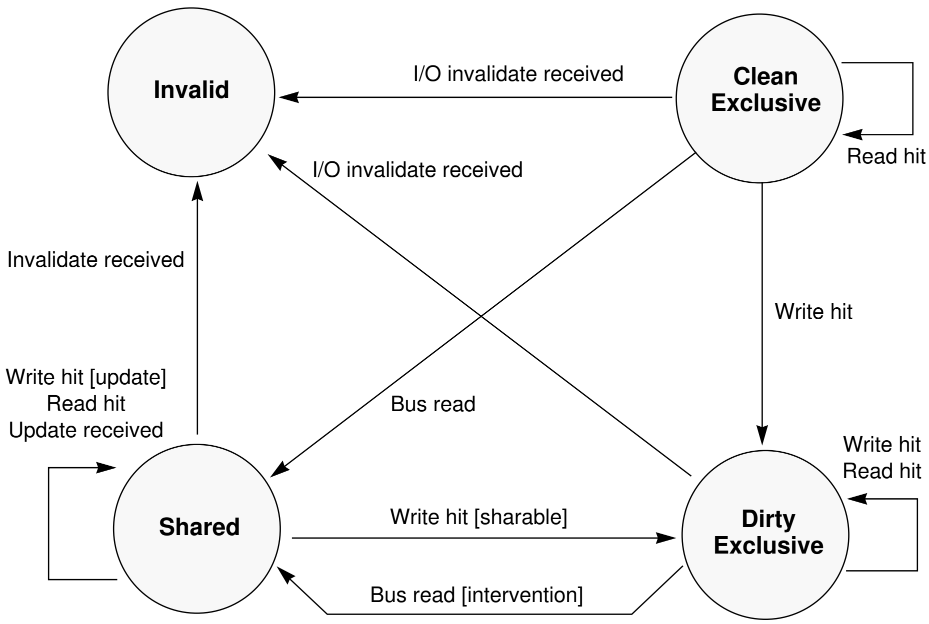


Figure 11-9 Primary Data Cache State Diagram

If the system is in no-secondary-cache mode, the cache state provided by the system is ignored, and the primary data cache state is set to dirty exclusive.

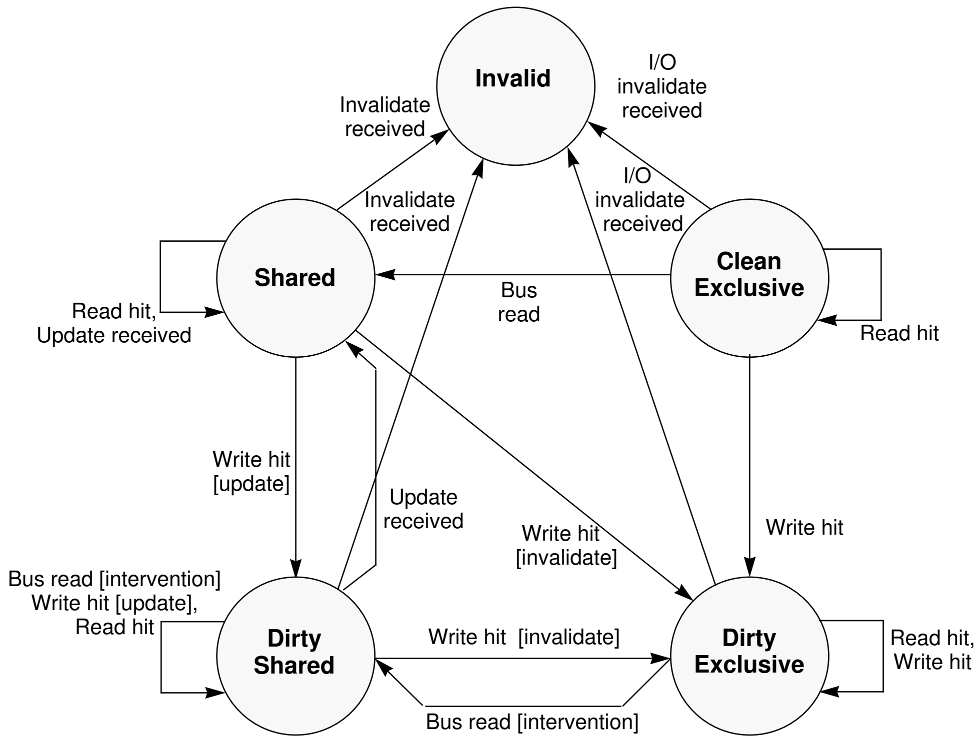


Figure 11-10 Secondary Cache State Diagram

The state of a secondary cache line is provided by the external agent and is set as follows:

Case 1. If the cache line is not present in another cache, it should be loaded in the clean exclusive state.

Case 2. If the cache line is retained by another cache and the state of the line in that cache remains shared or dirty shared, the line should be loaded in the shared state.

Case 3. If the cache line is retained by another cache and the cache relinquishes ownership to the processor making the read request, the line should be returned in the dirty shared state.

Case 4. If the cache line is retained by another cache and ownership is relinquished to memory, the line should be loaded in the shared state.

Case 5. If the cache line is relinquished by another cache and ownership is transferred to the processor making the read request, the line should be loaded in the dirty exclusive or dirty shared state.

For case 1, if the refill occurs on a store miss, the processor changes the cache line state to dirty exclusive. For each of the remaining cases listed above, the R4000 processor passes the state received from the external agent to the secondary cache.

The invalid state is never used for a refill. Software, however, should initialize the secondary cache to the invalid state after the system is powered up.

11.8 Cache Coherency Overview

Systems using more than one R4000MC processor must have a mechanism to maintain data consistency throughout a multi-cache, multiprocessor system. This mechanism is called a cache coherency protocol.

Cache Coherency Attributes

Cache coherency attributes are necessary to ensure the consistency of data throughout the multitude of caches that can be present in the multiprocessor environment.

Bits in the translation look-aside buffer (TLB) control coherency on a per-page basis. Specifically, the TLB contains 3 bits per entry that provide five possible coherency attributes; they are listed below and described more fully in the following sections.

- *uncached* (R4000PC, R4000SC, R4000MC)
- *noncoherent* (R4000PC, R4000SC, R4000MC)
- *sharable* (R4000MC only, with secondary cache)
- *update* (R4000MC only, with secondary cache)
- *exclusive* (R4000MC only, with secondary cache)

Only uncached or noncoherent attributes can be used by an R4000PC or an R4000SC processor.

Table 11-4 summarizes the behavior of the processor on load misses, store misses, and store hits to shared cache lines for each of the five coherency attributes listed above. The following sections describe in detail the five coherency attributes.

Table 11-4 Coherency Attributes and Processor Behavior

Attribute	Load Miss	Store Miss	Store Hit Shared
Uncached	Main memory read	Main memory write	NA
Noncoherent	Noncoherent read	Noncoherent read	Invalidate †
Exclusive	Coherent read exclusive	Coherent read exclusive	Invalidate †
Sharable	Coherent read	Coherent read exclusive	Invalidate
Update	Coherent read	Coherent read	Update

† These should not occur under normal circumstances.

Uncached

Lines within an *uncached* page are never in a cache. When a page has the uncached coherency attribute, the processor issues a doubleword, partial-doubleword, word, or partial-word read or write request directly to main memory (bypassing the cache) for any load or store to a location within that page.

Noncoherent

Lines with a *noncoherent* attribute can reside in a cache; a load or store miss causes the processor to issue a noncoherent block read request to a location within the cached page.

Sharable

Lines with a *sharable* attribute must be in a multiprocessor environment (using the R4000MC), since shared lines can be in more than one cache at a time. When the coherency attribute is *sharable*, the processor operates as follows:

- a coherent block read request is issued for a load miss to a location within the page, or
- a coherent block read request that requests exclusivity is issued for a store miss to a location within the page.

In most systems, coherent read requests require snoops or directory checks, and noncoherent read requests do not.† Cache lines within the page are managed with a *write invalidate* protocol; that is, the processor issues an invalidate request on a store hit to a shared cache line.

Update

Lines with an *update* coherency attribute must be in a multiprocessor environment and can reside in more than one cache at a time. When the coherency attribute is update, the processor issues a coherent block read request for a load or store miss to a location within the page. Cache lines within the page are managed with a *write update* protocol; that is, the processor issues an update request on a store hit to a shared cache line.

† A coherent read that requests exclusivity implies that the processor functions most efficiently if the requested cache line is returned to it in an exclusive state, but the processor still performs correctly if the cache line is returned in a shared state.

Exclusive

Lines with an *exclusive* coherency attribute must be in a multiprocessor environment. When the coherency attribute is exclusive, the processor issues a coherent block read request that requests exclusivity for a load or store miss to a location within the page.

Cache lines within the page are managed with a write invalidate protocol.

> **NOTE:** Load Linked-Store Conditional instruction sequences must ensure that the link location is not in a page managed with the exclusive coherency attribute.

Cache Operation Modes

The R4000 processor supports the following two *cache modes*:

- secondary-cache mode (R4000MC and R4000SC models; for R4000MC all five cache coherency attributes described above are applicable, and for R4000SC only uncached and noncoherent coherency attributes are applicable)
- no-secondary-cache mode (only uncached and noncoherent coherency attributes are applicable).

Secondary-Cache Mode

In its *secondary-cache* mode, an R4000MC model provides a set of cache states and mechanisms that implement a variety of cache coherency protocols. In particular, the processor simultaneously supports both the write-invalidate and write-update protocols.

No-Secondary-Cache Mode

A processor in *no-secondary-cache* mode supports the uncached and noncoherent coherency attributes. These two attributes are described in the section titled Cache Coherency Attributes in this chapter.

Strong Ordering

Cache-coherent multiprocessor systems must obey ordering constraints on stores to shared data. A multiprocessor system that exhibits the same behavior as a uniprocessor system in a multiprogramming environment is said to be *strongly ordered*.

An Example of Strong Ordering

Given that locations X and Y have no particular relationship—that is, they are not in the same cache line—an example of strong ordering is as follows:

1. At time **T**, Processor A performs a store to location X and at the same time processor B performs a store to location Y.
2. At time **T+1**, Processor A does a load from location Y and at the same time processor B does a load from location X.

For the system to be considered strongly ordered, either processor A must load the new value of Y, or processor B must load the new value of X, or both processors A and B must load the new values of Y and X, respectively, under all conditions.

If processors A and B load old values of Y and X, respectively, under any conditions, the system is not strongly ordered.

Testing for Strong Ordering

Table 11-5 shows the algorithm for testing strong ordering.

Table 11-5 Algorithm for Testing Strong Ordering

Time	Processor A	Processor B
T	Store to location X	Store to location Y
T+1	Load from location Y	Load from location X

For this algorithm to succeed, stores must have a global ordering in time; that is, every processor in the system must agree that either the store to location X precedes the store to location Y, or vice versa. If this global ordering is enforced, the test algorithm for strong ordering succeeds.

Restarting the Processor

Strong ordering requires precise control of a processor restart. Specifically, after completion of a processor coherency request, the system must ensure the completion of any cache state changes before allowing a processor restart.

The following sections describe processor restarts in a strong-ordered system after a processor coherency request.

Restart after a Coherent Read Request

Unless a processor invalidate or update request is unacknowledged after a coherent read request, the processor restarts (if sequential ordering is enabled) after the last word in the block has been transmitted to the processor.

Any external requests that must be completed before the read request is finished must be issued to the processor before the read response is issued.

Restart after a Coherent Write Request

The processor restarts after the coherent write request is completed. That is, the processor restarts after the last doubleword of data associated with the write request has been transmitted to the external agent, *unless* a processor read request is pending,† or a processor invalidate or update request is unacknowledged.

Restart after an Invalidate or Update Request

Following an invalidate or update request, the processor restarts after the external agent asserts **IvdAck*** or **IvdErr***, *unless* a processor read request is pending or the processor is processing an external request when either **IvdAck*** or **IvdErr*** is asserted.

If either **IvdAck*** or **IvdErr*** is asserted during or after the first cycle that the external agent asserts **ExtRqst***, the processor accepts the external request and completes any cache state changes associated with the external request before restarting.

† That is, present but not yet executed.

If either **IvdAck*** or **IvdErr*** is asserted before, but not asserted during or after the first cycle that the external agent asserts **ExtRqst***, the processor restarts before beginning the external request.

External requests must be completed before a processor invalidate or update completes. They can be completed, provided the processor receives an asserted **ExtRqst*** by the external agent either before or during the same cycle **IvdAck*** or **IvdErr*** is asserted.

11.9 Maintaining Coherency on Loads and Stores

Cache coherency protocols maintain data consistency throughout a multiprocessor environment. Table 11-6 lists the coherency effects of load and store operations on primary and secondary cache states in a multiprocessor environment (using an R4000MC processor).

Table 11-6 R4000MC Data Cache Coherency States

Primary Cache States	Secondary Cache States	Action on Load	Action on Store
Invalid	Any	Miss	Miss
Shared	Shared Dirty Shared	None	Read secondary tag. If the coherency algorithm is Update on Write, then send update and set the secondary cache state to Dirty Shared. If the coherency algorithm is Invalidate on Write, then send invalidate and set the primary and secondary cache states to Dirty Exclusive.
	Dirty Exclusive	None	Set the primary cache state to Dirty Exclusive.
Clean Exclusive	Clean Exclusive	None	Set the primary and secondary cache states to Dirty Exclusive.
	Dirty Exclusive	None	Set the primary data cache state to Dirty Exclusive.
Dirty Exclusive†	Dirty Exclusive	None	None

† The dirty exclusive primary state allows the primary cache to be written without a secondary access.

11.10 Manipulation of the Cache by an External Agent

Just as the processor accesses caches, so too can an external agent examine and manipulate the state and content of the primary and secondary caches through invalidate, update, snoop, and intervention transactions.

These transactions are described in the following sections. Encodings of these request transactions are given in Chapter 12.

Invalidate

An invalidate request causes the processor to change the state of the specified cache line to invalid in both the primary and secondary caches.

Update

An update request causes the processor to write the specified data element into the specified cache line, and either change the state of the cache line to shared in both the primary and secondary caches, or leave the state of the cache line unchanged, depending on the nature of the update request. An external agent can issue updates to cache lines that are in either the exclusive or shared states without changing the state of the cache line (see the **SysCmd(3)** bit description in Chapter 12).

> **NOTE:** If there is an update to a line in the primary instruction cache, the line in the secondary cache is updated and the primary instruction cache line is invalidated.

Snoop

A snoop request to the processor causes the processor to return the secondary cache state of the specified cache line.

At the same time, the processor atomically† sets the state of the specified cache line in both the primary and secondary caches according to the value of the **SysCmd(2:0)** bits, which define cache state change, and are supplied by the external agent.

† An atomic operation is one that cannot be split, or portions of it deferred. In this case, the processor sets the state of both secondary and primary caches in an indivisible action; it cannot set the state of one cache line, allow another process to interrupt, and then complete the first process by setting the state of the remaining cache line.

Intervention

An intervention request causes the processor to return the secondary cache state of the specified cache line and, under certain conditions related to the state of the cache line and the nature of the intervention request, the contents of the specified secondary cache line.

At the same time, the processor atomically sets the state of the specified cache line in both the primary and secondary caches according to the value of the **SysCmd(2:0)** bits which define cache state change, and are supplied by an external agent.

11.11 Coherency Conflicts

The R4000MC processor must handle competing coherency conflicts that arise from the processor and an external source. This section describes how coherency conflicts arise and how they are handled. A system model illustrates the implications of coherency conflicts in a multiprocessor environment; a coherent read request cycle is described at the end of this section.

Figure 11-11 shows the R4000MC processor issuing processor coherency requests and accepting external coherency requests.

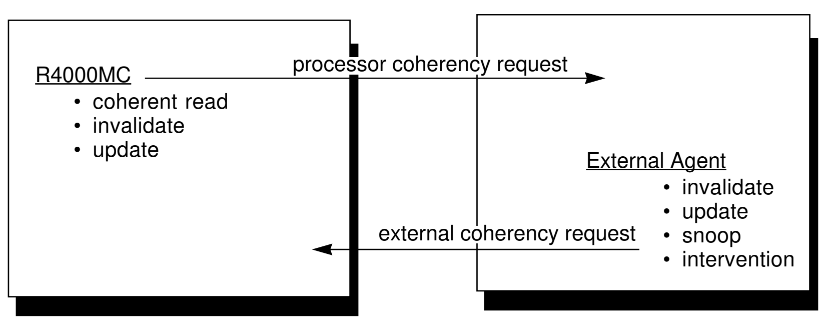


Figure 11-11 Coherency Requests: Processor and External

The R4000MC processor issues the following processor coherency requests:

- processor coherent read requests
- processor invalidate requests
- processor update requests

The R4000MC processor accepts the following external coherency requests:

- external invalidate requests
- external update requests
- external snoop requests
- external intervention requests

How Coherency Conflicts Arise

Because of the overlapped nature of the system interface, it is possible for an external coherency request to target the same cache physical address as a pending processor read request, an unacknowledged processor invalidate, or an update request. The processor does not contain the comparison mechanism necessary to detect such conflicts; instead, it uses the secondary cache as a point of reference to determine suitable coherency actions, and only checks the state of the secondary cache at specific times.

Processor Coherent Read Requests

When the processor wants to service either a store or load cache miss for a page that has a coherent page attribute in the TLB (meaning the data passed back and forth should follow a defined multiprocessor coherency scheme), a coherent read request is used.

Conflicting external coherency requests cannot affect the behavior of the processor for pending processor coherent read requests. The processor only issues read requests for a range of physical addresses not currently in the cache; consequently, an external coherency request that targets the same physical address range will not find this physical address range in the cache. In such a case, the processor simply discards any external coherency requests that conflict with a pending processor coherent read request.

Processor Invalidate or Update Requests

For processor invalidate or compulsory update requests, a *cancellation* mechanism indicates a conflict. For example, if an external coherency request is submitted while a processor invalidate or compulsory update request has been issued but not yet acknowledged, the conflict is resolved when the external agent cancels the processor invalidate or compulsory update.

Cancellation is accomplished by setting the cancellation bit in the command for the coherency request [**SysCmd(4)**]. The processor, upon receiving an external coherency request with the cancellation bit set, considers its invalidate or update request to be acknowledged and cancelled. The processor again accesses the secondary cache to determine whether to reissue the invalidate or update request, or to issue a read request.

An external agent can only assert the cancellation bit during an unacknowledged processor invalidate or unacknowledged compulsory update request. If an external coherency request is issued with the cancellation bit set, and there is no unacknowledged processor invalidate or update request pending, the behavior of the processor is undefined.

If an external coherency request is issued with the cancellation bit set when a processor update request remains potential—in other words, while a processor read request is currently pending—the behavior of the processor is undefined.

Processor potential update requests cannot be cancelled. Potential updates are always issued with processor read requests and become compulsory only after the response to the processor read request is returned in one of the shared states.

External Coherency Requests

If an external agent issues an external coherency request that conflicts with an unacknowledged processor invalidate or update request, without setting the cancellation bit, the system will operate in an undefined manner. In this case, the processor has no indication of the conflict and does not reevaluate the cache state to determine the correct action; it simply waits for an acknowledge to its invalidate or update request as it would for any invalidate or update request.

It is not possible for external coherency requests to conflict with processor write requests, since the processor does not accept external requests while a processor write request is in progress.

Tables 11-7 and 11-8 summarize the interactions between processor coherency requests and conflicting external coherency requests, organized by processor state. These two tables show the processor in one of the following states:

Idle: no processor transactions are pending.

Read Pending: a processor coherent read request has been issued, but the read response has not been received.

Potential Update Unacknowledged: a processor update request has been issued while a processor coherent read request is pending but not yet acknowledged. By definition, therefore, the response to the coherent read request has not been received.

Invalidate or Update Unacknowledged: a processor invalidate or update request has been issued but has not yet been acknowledged. By definition, no coherent read request is pending.

Table 11-7 Summary of Coherency Conflicts: Invalidate and Update

Processor State	Conflicting External Coherency Request			
	Invalidate	**Invalidate with Cancel**	**Update**	**Update with Cancel**
Idle	NA	Undefined	NA	Undefined
Read Pending	OK	Undefined	OK	Undefined
Potential Update Unacknowledged	OK	Undefined	OK	Undefined
Invalidate or Update Unacknowledged	OK†	OK	OK†	OK

† This can cause incorrect system operation and normally should not be allowed to occur.

Table 11-8 Summary of Coherency Conflicts: Intervention and Snoop

Processor State	Conflicting External Coherency Request			
	Intervention	**Intervention with Cancel**	**Snoop**	**Snoop with Cancel**
Idle	NA	Undefined	NA	Undefined
Read Pending	OK	Undefined	OK	Undefined
Potential Update Unacknowledged	OK	Undefined	OK	Undefined
Invalidate or Update Unacknowledged	OK†	OK	OK†	OK

† This can cause incorrect system operation and normally should not be allowed to occur.

System Implications of Coherency Conflicts

The constraints that the processor must place on the handling of coherency conflicts have certain implications on the design of a multiprocessor system using the R4000MC model. These constraints and their implications are described in this section.

System Model

To describe the implications of a coherency conflict, this section uses a system model that is snooping, split-read, and bus-based; I/O is not considered in this model.

The system model used in this example has the following components:

- Four processor subsystems, each consisting of an R4000MC processor, a secondary cache, and an external agent (shown in Figure 11-12). The external agent communicates with the R4000MC processor, accepting processor requests and issuing external requests. Likewise, the system bus issues and receives bus requests.
- A memory subsystem that communicates with main memory and the system bus.
- A system bus that has the following characteristics:
  - It is a multiple master, request-based, arbitrated bus. When an agent wishes to perform a transaction on the bus, it must request the bus and wait for global arbitration logic to assert a grant signal before assuming mastership of the bus. Once mastership has been granted, the agent can begin a transaction.
  - It supports read transactions, read exclusive transactions, write transactions, and invalidate transactions.
  - It is a split-read bus. This means bus operations can separate a read request from the return of its data.
  - It is a snooping bus. All agents connected to the bus must monitor all bus traffic to correctly maintain cache coherency.
- All of the TLB pages in the system have either a noncoherent or a sharable coherency attribute. (Noncoherent data is not allowed; noncoherent page attributes are used for instructions only.)
- The sharable coherency attribute allows data to be shared between the four caches in the system by using a write invalidate cache coherency protocol.
- The secondary cache states used are invalid, shared, clean exclusive, and dirty exclusive; the dirty shared secondary cache state is not allowed.

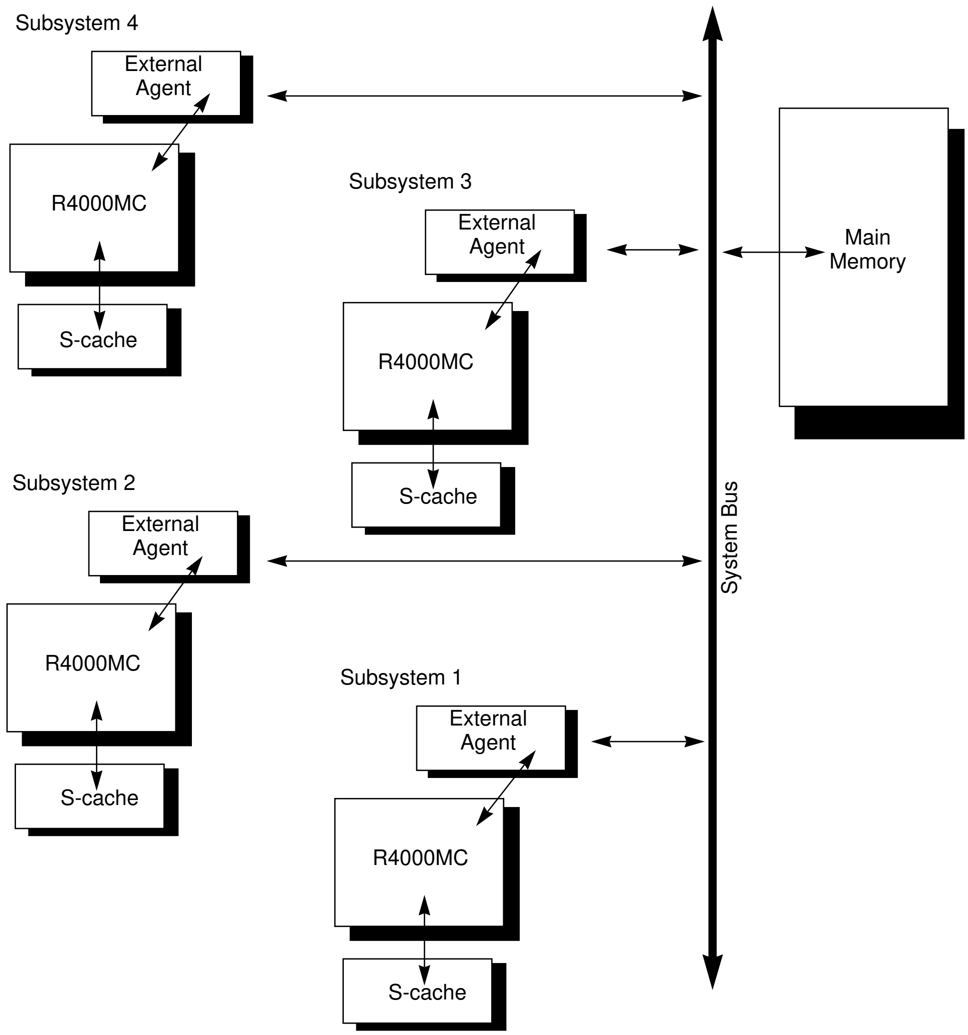


Figure 11-12 4-Processor System Illustrating Coherency Transactions

Given this system model, the following operations are described:

- loads and stores
- processor coherent read request and read response
- processor invalidate
- processor write

Load

A shown in Figure 11-12, when a processor misses in the primary and secondary caches on a load, the processor issues a read request. The subsystem external agent translates this to a read request on the bus. The returned data is loaded in either the clean exclusive or shared state, based on the shared indication returned with the read response data.†

Store

In this system model, when a processor misses in the primary and secondary caches on a store, it issues a read request with exclusivity; this is translated to a read exclusive on the bus and data is loaded in the dirty exclusive state.

When a processor hits in the cache on a store to shared data, it issues an invalidate request that must be forwarded to the system bus. Before the store can be completed and the state changed to dirty exclusive, the invalidate request must be acknowledged.

Processor Coherent Read Request and Read Response

In this system model, when one of the external agents observes a coherent read request on the system bus, it does not take immediate action. Instead, the external agent issues an intervention request to its processor during the read response. This is referred to as a *response complete* read protocol; that is, the read is complete after the read response has occurred.

At the end of the read response, each of the external agents in the system model indicate whether it was able to obtain the state of the cache line that is the target of the intervention; if successful, the external agent indicates either sharing or takeover. Takeover occurs when an external agent discovers that its processor has a dirty exclusive copy of the cache line that is the target of the read.

The read response is extended until all external agents have obtained the state of the cache line from their processors.

In this system model, the response from an external agent at the end of a read response depends on whether the read request was an ordinary read request or a read exclusive request. These are described in the following sections.

† The shared indication is the result of an intervention request to another processor, and is supplied by an external agent that is a part of the other three processor subsystems.

Ordinary Read Request

For an ordinary read request, an external agent indicates *shared* at the end of the read response if it finds that its processor has a copy of the requested cache line in the clean exclusive or shared state.

An external agent indicates both shared and takeover at the end of a read response if it finds that its processor has a copy of the requested cache line in the dirty exclusive state. Having indicated takeover, the external agent supplies the contents of the cache line (returned by the processor in response to the intervention request) over the bus to the read requester, and causes the processor to change the state of the cache line to shared. At the same time the cache line is supplied to the read requester, it is also written back to memory.

Read Exclusive Request

For a read exclusive request, an external agent never indicates shared at the end of the read response, regardless of the state the cache line is in. Instead, the cache line must be in one of the following states:

- If the current state of the cache line is clean exclusive or shared, the external agent changes the state of the cache line to invalid.
- If the current state of the cache line is dirty exclusive, the external agent indicates takeover but not shared. Having indicated takeover, the external agent supplies the contents of the cache line to the read requester, and the processor changes the state of the cache line to invalid. While the cache line is supplied to the read requester, it is also written back to memory.

Processor Invalidate

In this system model, an invalidate request is considered complete as soon as it appears on the system bus. When an external agent observes an invalidate request on the system bus, it reacts as if the invalidate has changed the state of all caches at that instant.

Processor Write

In this system model, an external agent takes no action in response to a write request on the bus.

Handling Coherency Conflicts

Coherency conflicts are examined and resolved based on the current state of the processor. Referring to Figure 11-12, the following conflicts and their resolutions are described in this section:

- coherent read conflicts
- coherent write conflicts
- invalidate conflicts

Coherent Read Conflicts

External coherency requests that conflict with pending processor coherent read requests can be issued to the processor without affecting processor behavior. In the system model shown in Figure 11-12, no conflict detection is performed by the external agent for processor coherent read requests; if an external intervention request or invalidate request is forwarded to the processor that is in conflict with a pending processor coherent read request, it does not affect the processor cache since the targeted cache line is, by definition, absent from the cache. The processor effectively discards the conflicting external intervention request, responding with an invalid indication for the targeted cache line. Similarly, the processor discards a conflicting external invalidate request since the targeted cache line is not present and therefore invalid.

For pending processor coherent read requests, conflict detection could be added to a system similar to the one shown in Figure 11-12. In such a case, when the external agent sees a read response on the bus that conflicts with a pending processor coherent read request, the external agent does not issue an intervention request to the processor. Rather, it simply reacts as if an intervention request has been completed and the cache line is not present in the processor cache.

Similarly, when an external agent sees an invalidate request on the bus that conflicts with a pending processor coherent read request, it does not forward the invalidate request to the processor since the targeted cache line is absent from the processor cache. This scheme for conflict detection on processor coherent read requests could reduce the number of external intervention and invalidate requests issued to the processor. However, since the intervention and invalidate requests that would otherwise be issued to the processor cannot result in any state modification within the processor (since the targeted cache line is not present in the cache), conflict detection for processor coherent read requests is not necessary.

Coherent Write Conflicts

As soon as a write request has been issued to the external agent, the external agent becomes responsible for the cache line. No conflicts are possible with a processor write request; however, the external agent must manage ownership of the cache line while it is waiting to acquire mastership of the system bus so that it can forward the write request. The external agent is responsible for the cache line from the time the issue cycle of the write request completes until the write request is forwarded to the system bus.

If the response to a coherent read request conflicts with a waiting processor write request, or with a processor write request that is transmitting data, the external agent detects the conflict and does not issue an intervention request to the processor. Instead, it reacts as if an intervention request has been completed and the line is in the dirty exclusive state. The external agent indicates takeover and supplies the read data to the read requester itself without disturbing the processor. After providing the read data to the read requester, the external agent must discard the write request if the read request was a read exclusive. In fact, the external agent can ignore the write request for either type of read, since processor-supplied read data is also written back to memory.

It is not possible for an invalidate request, or a write request that conflicts with a waiting processor write request, to appear on the system bus; before a processor write request can be issued, the state of the processor cache line must be set to dirty exclusive.

Invalidate Conflicts

From the time the processor issues an invalidate request until that request is acknowledged, any external coherency request issued to the processor that conflicts with the unacknowledged invalidate must include a cancellation.

In the model system shown in Figure 11-12, an acknowledge for the invalidate is sent to the processor as soon as the invalidate is forwarded to the system bus. Therefore, while the external agent is waiting to become a bus master to forward the invalidate request, the external agent must detect, by using comparators, any external coherency request that conflicts with the unacknowledged invalidate. If a conflict is detected, the external agent must not forward the invalidate request to the system bus; instead, it must rescind the invalidate request and submit the conflicting external request to the processor, with a cancellation for the invalidate request.

If the response to a coherent read request conflicts with a waiting unacknowledged processor invalidate request, the external agent detects this conflict and does not forward the processor invalidate request to the bus. Instead, it discards the processor invalidate request and issues to the processor an intervention request that includes a cancellation. The processor then reevaluates its cache state and either reissues the invalidate request or issues a coherent read request.

If an invalidate request appears on the bus while the external agent has a processor invalidate request waiting, and the external agent detects the conflict, the external agent does not forward the processor invalidate request. Instead, it discards the processor invalidate request and issues an external invalidate request that includes a cancellation to the processor. The processor then reevaluates its cache state and either reissues the invalidate request or issues a coherent read request.

It is not possible for a write request that conflicts with a waiting processor invalidate request to appear on the system bus. To issue an invalidate request, the state of the cache line must be shared with every cache in the system that contains the line.

Sample Cycle: Coherent Read Request

This section describes a multiprocessor system within which a coherent read request cycle† services a secondary cache load miss. The system has two processors, P_A and P_B, and two external agents linked to these processors, external agent A (E_A) and external agent B (E_B). The external agents connect the processors to a system bus. Each of the processors has its own secondary cache.

The sample cycle follows the steps below (these steps are also numbered in Figures 11-13, 11-14, and 11-15):

1. Processor B has a load miss within a sharable page.
2. Processor B issues a *coherent read request* (CRR) through E_B.
3. The CRR is placed on the bus.

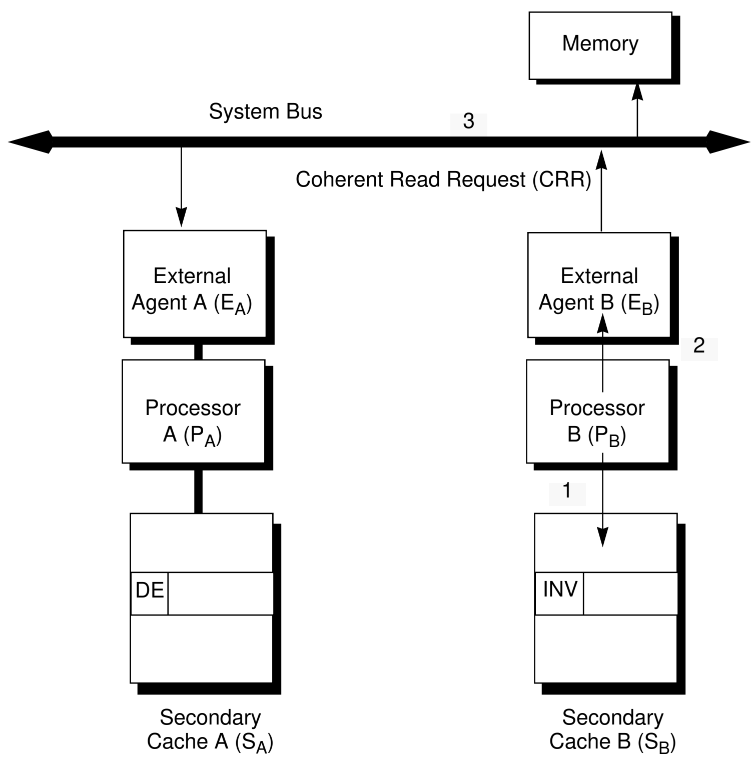


Figure 11-13 Cache Load Miss Cycle: Coherent Read Request

† Request Cycles are described in Chapter 12.

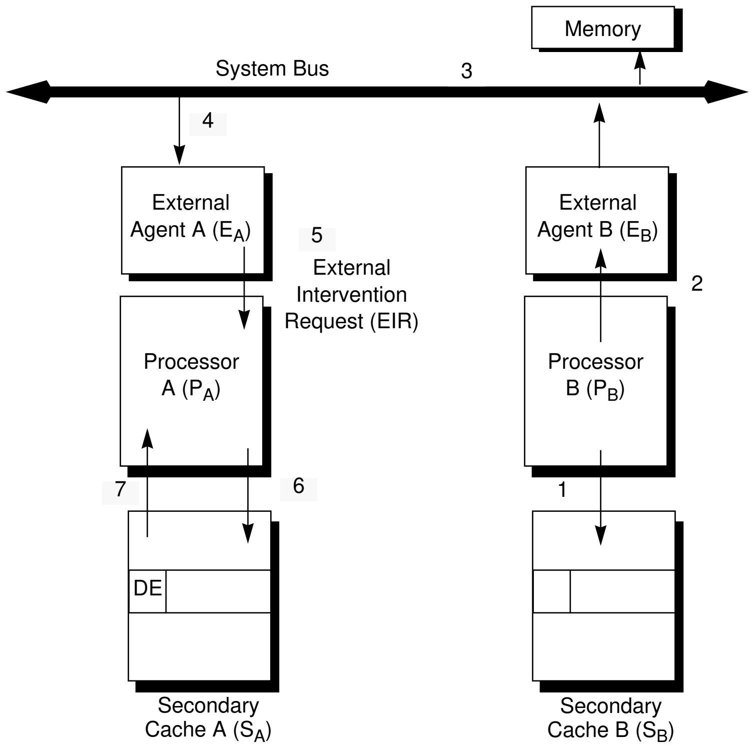


Figure 11-14 Cache Load Miss Cycle: External Intervention

4. As shown in Figure 11-14, external agent E_A reads the CRR from the bus.
5. To service this CRR, E_A issues an external intervention request (EIR) to processor A, P_A.
6. P_A receives the EIR and examines its secondary cache, S_A.
7. Depending on the type of intervention request—based on the state of the *SysCmd(3)* bit—one of the following actions is taken:
   - If the cache line in S_A is in the dirty exclusive state, the entire cache line is returned.
   - Otherwise, P_A just returns the state of the secondary cache line.

   In Figure 11-14 the retrieved data is in the dirty exclusive state (DE), servicing a load miss, when the state of cache line S_A goes from dirty exclusive to dirty shared (DS),† indicating P_A is owner of the line.

† Assuming DS mode is enabled.

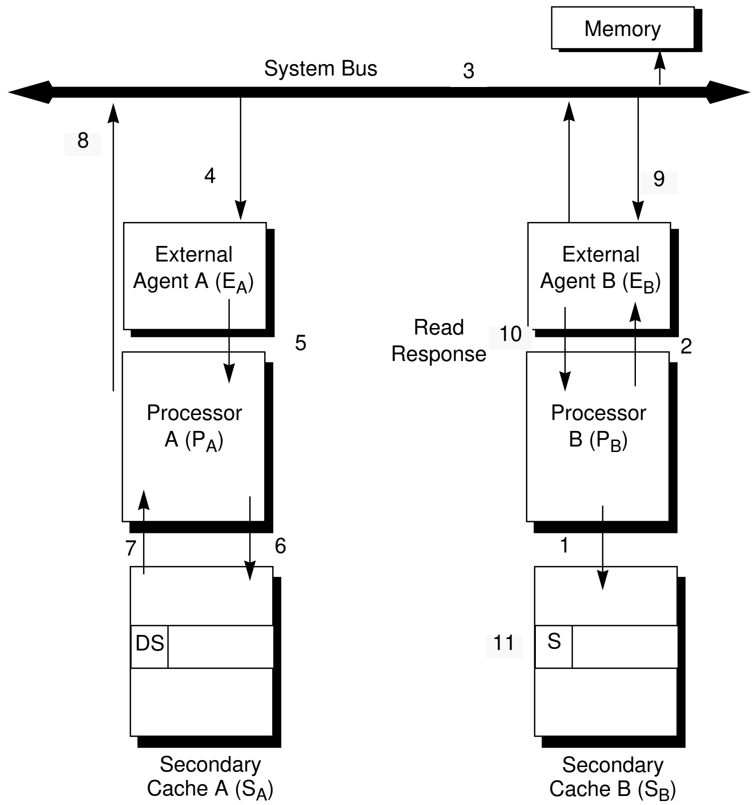


Figure 11-15 Cache Load Miss Cycle: Read Response

8. Figure 11-15 shows the cache state and cache data returned from P_A, through E_A to the bus.
9. This cache state and data are returned to E_B.
10. E_B issues a read response to P_B.
11. P_A remains owner of the cache line.

11.12 R4000 Processor Synchronization Support

In a multiprocessor system, it is essential that two or more processors working on a common task execute without corrupting each other's subtasks. Synchronization, an operation that guarantees an orderly access to shared memory, must be implemented for a properly functioning multiprocessor system. Two of the more widely used methods are discussed in this section: test-and-set, and counter.

Test-and-Set (Spinlock)

Test-and-set† uses a variable called the *semaphore*, which protects data from being simultaneously modified by more than one processor.

In other words, a processor can lock out other processors from accessing shared data when the processor is in a *critical section*, a part of program in which no more than a fixed number of processors is allowed to execute. In the case of test-and-set, only one processor can enter the critical section.

Figure 11-16 illustrates a test-and-set synchronization procedure that uses a semaphore; when the semaphore is set to 0, the shared data is unlocked, and when the semaphore is set to 1, the shared data is locked.

† Test-and-set is sometimes referred to as *spinlock*.

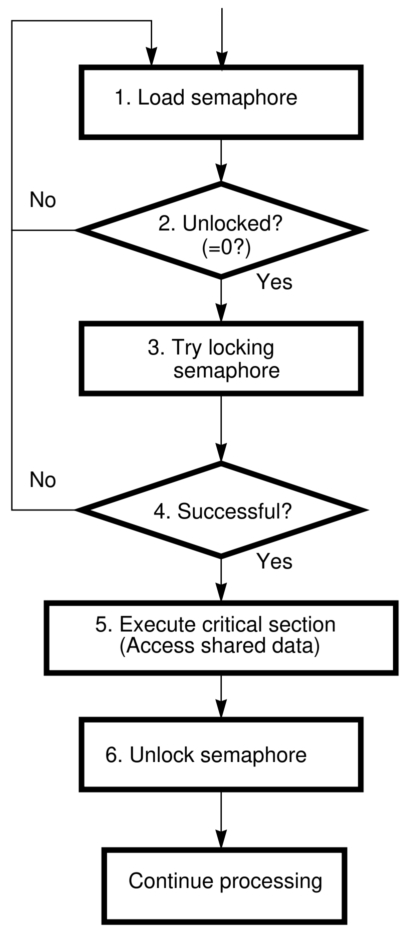


Figure 11-16 Synchronization with Test-and-Set

The processor begins by loading the semaphore and checking to see if it is unlocked (set to 0) in steps 1 and 2. If the semaphore is not 0, the processor loops back to step 1. If the semaphore is 0, indicating the shared data is not locked, the processor next tries to lock out any other access to the shared data (step 3). If not successful, the processor loops back to step 1, and reloads the semaphore.

If the processor is successful at setting the semaphore (step 4), it executes the critical section of code (step 5) and gains access to the shared data, completes its task, unlocks the semaphore (step 6), and continues processing.

Counter

Another common synchronization technique uses a *counter*. *A counter is* a designated memory location that can be incremented or decremented.

In the test-and-set method, only one processor at a time is permitted to enter the critical section. Using a counter, up to *N* processors are allowed to concurrently execute the critical section. All processors after the *N*th processor must wait until one of the *N* processors exits the critical section and a space becomes available.

The counter works by not allowing more than one processor to modify it at any given time. Conceptually, the counter can be viewed as a variable that counts the number of limited resources (for example, the number of processes, or software licenses, etc.). Figure 11-17 shows this process.

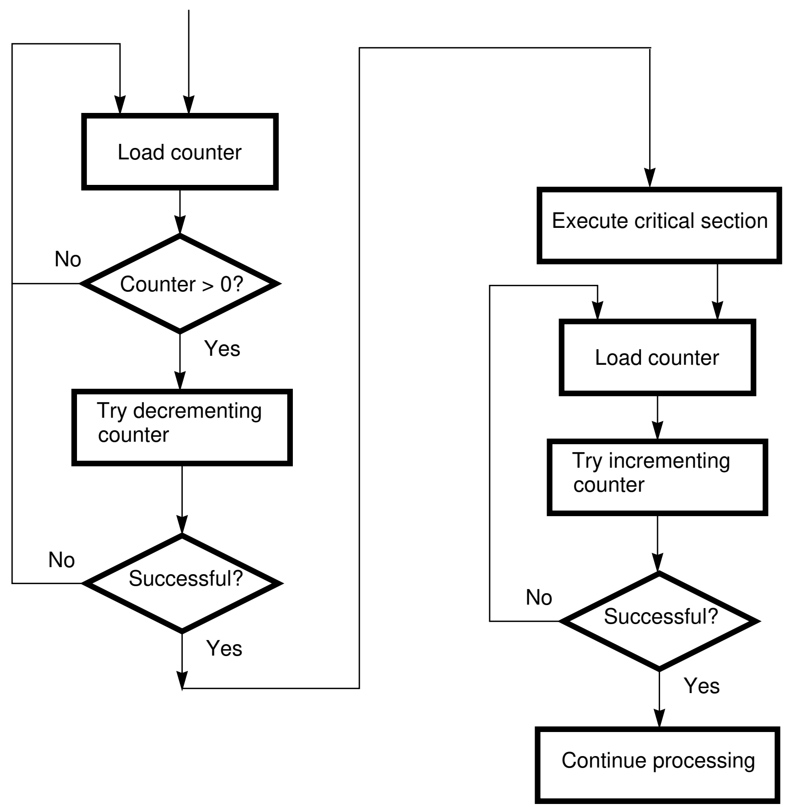


Figure 11-17 Synchronization Using a Counter

LL and SC

MIPS instructions *Load Linked* (LL) and *Store Conditional* (SC) provide support for processor synchronization. These two instructions work very much like their simpler counterparts, load and store. The LL instruction, in addition to doing a simple load, has the side effect of setting a bit called the *link bit*. This link bit forms a breakable link between the LL instruction and the subsequent SC instruction. The SC performs a simple store if the link bit is set when the store executes. If the link bit is not set, then the store fails to execute. The success or failure of the SC is indicated in the target register of the store.

The link is broken in the following circumstances:†

- if any external request (invalidate, snoop, or intervention) changes the state of the line containing the lock variable to invalid
- upon completion of an ERET (return from exception) instruction
- an external update to the cache line containing the lock variable

The most important features of LL and SC are:

- They provide a mechanism for generating all of the common synchronization primitives including test-and-set, counters, sequencers, etc., with no additional overhead.
- When they operate, bus traffic is generated only if the state of the cache line changes; lock words stay in the cache until some other processor takes ownership of that cache line.

† The most obvious case where the link is broken occurs when an invalidate to the cache line is the subject of the load. In this case, some other processor has successfully completed a store to that line.

Examples Using LL and SC

Figure 11-18 shows how to implement test-and-set using LL and SC instructions.

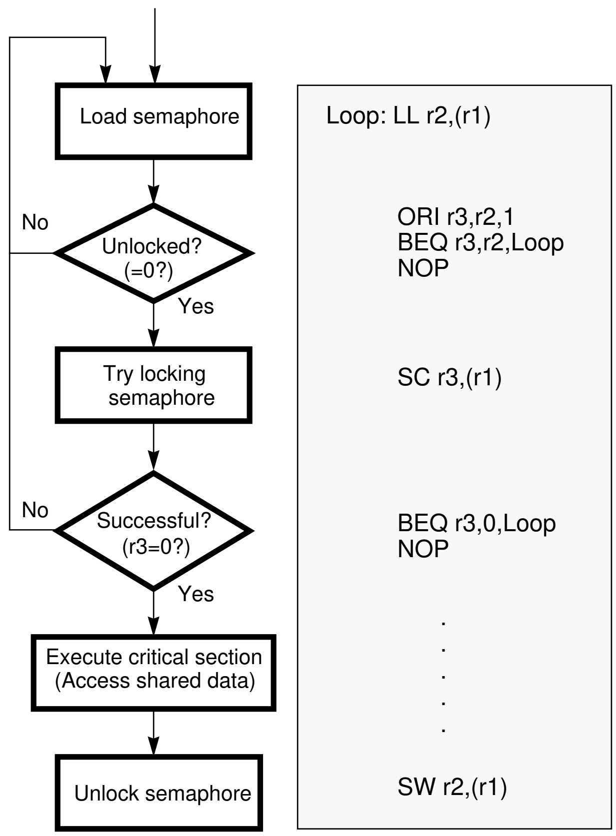


Figure 11-18 Test-and-Set using LL and SC

Figure 11-19 shows synchronization using a counter.

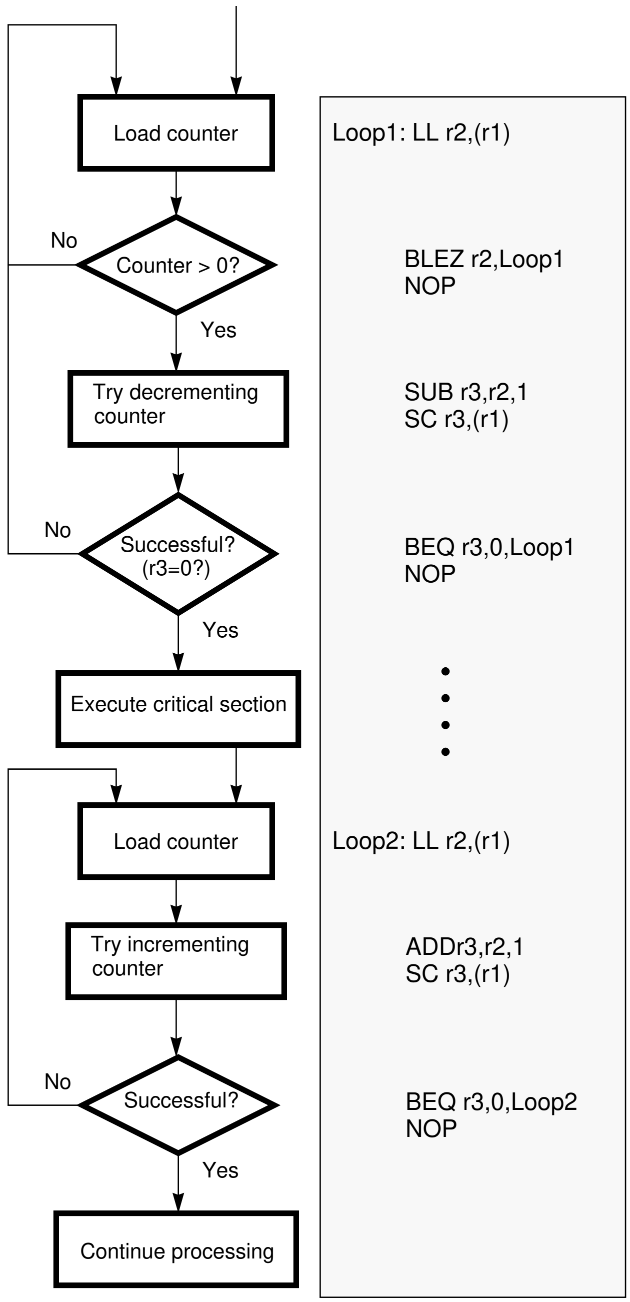


Figure 11-19 Counter Using LL and SC