



---

# *Implementing a Memory-Based Mutex and Barrier Synchronization Library*

Application Note

Tensilica, Inc.  
3255-6 Scott Blvd.  
Santa Clara, CA 95054  
(408) 986-8000  
Fax (408) 986-8919  
[www.tensilica.com](http://www.tensilica.com)

*July, 2007*

*Doc Number: AN07-092-00*

© TENSILICA, INC.



## Implementing a Memory-Based Mutex and Barrier Synchronization Library

© 2007 Tensilica, Inc.

Printed in the United States of America

All Rights Reserved

This publication is provided "AS IS." Tensilica, Inc. (hereafter "Tensilica") does not make any warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

Information in this document is provided solely to enable system and software developers to use Tensilica processors. Unless specifically set forth herein, there are no express or implied patent, copyright or any other intellectual property rights or licenses granted hereunder to design or fabricate Tensilica integrated circuits or integrated circuits based on the information in this document. Tensilica does not warrant that the contents of this publication, whether individually or as one or more groups, meets your requirements or that the publication is error-free. This publication could include technical inaccuracies or typographical errors. Changes may be made to the information herein, and these changes may be incorporated in new editions of this publication.

Tensilica is a registered trademark of Tensilica, Inc. The following terms are trademarks of Tensilica, Inc.: FLIX, OSKit, Sea of Processors, TurboXim, Vectra, Xenergy, Xplorer, and XPRES. All other trademarks and registered trademarks are the property of their respective companies.

### Notice

Tensilica, Inc. reserves the right to make changes to its products or discontinue any of its products or offerings without notice.

Tensilica warrants the performance of its products to the specifications applicable at the time of sale in accordance with Tensilica's standard warranty.

### Document Change History:

Published July 2007



## Contents

|   |   |   |
|---|---|---|
| 1 | Introduction .....                                      | 1 |
| 2 | The S32C1I Instruction .....                            | 1 |
| 3 | Managing Coherence.....                                 | 2 |
| 4 | Maintaining the Program Order of Memory Operations..... | 3 |
| 5 | LIBXMP: Xtensa Synchronization Library .....            | 5 |
| 6 | Example Code .....                                      | 5 |



## **Abstract**

Programming multiprocessor systems often requires that the tasks running on the processors be able to synchronize with each other and be able to safely access shared data. To enable multiprocessor synchronization and data sharing, the Xtensa architecture includes several multiprocessor synchronization instructions. This application note describes how the multiprocessor synchronization instructions can be used to implement a memory-based mutual exclusion (mutex) primitive. Included with this application note is the complete source code for a C/C++ library that implements an example mutex API as well as a barrier synchronization API.

# 1 Introduction

Programming multiprocessor systems often requires that the tasks running on the processors be able to synchronize with each other and be able to safely access shared data. To enable multiprocessor synchronization and data sharing via a shared memory, the Xtensa architecture includes several multiprocessor synchronization instructions.

This application note describes how multiprocessor synchronization primitives can be implemented on Xtensa processors. This document specifically discusses how a memory-based mutex (also referred to as a lock) primitive can be implemented in a multiprocessor system. Included with this application note is the complete source code for a C/C++ library that implements an example mutex and barrier API.

A *mutex* is useful for protecting shared data structures from concurrent modifications, and for implementing critical sections and monitors. A mutex has two possible states: unlocked (not owned by any core/thread), and locked (owned by one core/thread). A mutex can never be owned by two different threads simultaneously. A thread attempting to lock a mutex that is already locked by another thread is blocked until the owning thread unlocks the mutex. The thread that has locked a mutex is said to *own* the mutex.

A *barrier* is a synchronization primitive that requires a specified number of threads to wait at the barrier before any of the threads are allowed to proceed. A barrier can be used to synchronize multiple threads to ensure that all threads have completed a part of an application before proceeding to the next part.

## 2 The S32C1I Instruction

To enable multiprocessor synchronization, the Xtensa architecture provides the S32C1I instruction. This instruction performs a read-conditional-write operation which atomically stores to a memory location only if the current value of that memory location is equal to an expected value. For a complete description of S32C1I see the *Xtensa Instruction Set Architecture (ISA) Reference Manual*.

The following example code shows how to implement a mutex on a processor without data caches (we will discuss data caches in the following section). The `lock` variable is a pointer to an address in shared memory.

In the following code, the mutex is unlocked when `*lock` is 0, and is locked when `*lock` is 1. So to acquire the lock we must use S32C1I to check that the current value of `*lock` is 0 and atomically set it to 1. The code first sets the `SCOMPARE1` special register to 0 to indicate that the conditional store performed by the S32C1I instruction should only be allowed to succeed if the value of the memory location is 0. The code then repeatedly attempts to acquire the mutex by setting `*lock` to 1. If `*lock == SCOMPARE1` (that is, the mutex is not currently locked), the S32C1I instruction will set `*lock = 1v`. If `*lock != SCOMPARE1` (that is, the mutex is currently locked), S32C1I will not change the value of `*lock`. Regardless of the value of `*lock`, S32C1I will assign `1v` to the old value of `*lock`. Therefore, `1v` is assigned to 0 if, and only if, the mutex is not locked.

```
int *lock = <address of lock>;
int lv;
```

```
XT_WSR_SCOMPARE1(0);
do {
    lv = 1;
    XT_S32C1I(lv, lock, 0);
} while (lv != 0);
```

To unlock the mutex, the processor that owns the mutex simply needs to assign `*lock` to zero as shown in the following code.

```
*lock = 0;
```

The above code shows the basic implementation for mutex locking and unlocking. For an actual implementation, we must take into account data caching and the ordering of memory operations. These issues are described in the following sections.

### 3 Managing Coherence

When the shared memory holding a mutex is cacheable, the locking and unlocking code must explicitly manage the coherence of the data cache to ensure that writes to the mutex are made visible to other processors in the system, and that reads of the mutex access the system copy of the data and not a locally cached copy. The `S32C1I` instruction automatically manages cache coherence, so no explicit cache management is needed for memory locations accessed by that instruction. However, if there is additional data associated with the mutex that is not accessed by `S32C1I`, then we must explicitly manage its coherence, as shown in the following example.

Assume the mutex is composed of two data fields: the lock itself as in the previous section's example, and a field that holds the processor id of the core that currently holds the mutex or -1 if no core owns the mutex. The processor id field should be readable by processors that do not currently own the mutex. The following code implements a lock and unlock with explicit cache management operations required for this case.

```
#include <xtensa/config/core.h>
#if XCHAL_HAVE_S32C1I || XCHAL_HAVE_RELEASE_SYNC
#include <xtensa/tie/xt_sync.h>
#endif
...

int *lock = <address of lock>;
int *lock_pid = <address of lock processor id>;
int lv;

XT_WSR_SCOMPARE1(0);
do {
    lv = 1;
    XT_S32C1I(lv, lock, 0);
} while (lv != 0);

/* This point is reached when the thread owns the mutex. Set
   the 'lock_pid' value to this core's processor id and then flush
   that cache line so that the value is written to system memory
```

```
    and other processors can then access it. */
    *lock_pid = <processor id>;
    xthal_dcache_line_writeback(lock_pid);

    ...

    /* Done with the mutex, so set 'lock_pid' to -1 and then release
       mutex. We must explicitly flush both 'lock' and 'lock_pid' so
       that they are visible to other cores. */
    *lock_pid = -1;
    xthal_dcache_line_writeback(lock_pid);
    *lock = 0;
    xthal_dcache_line_writeback(lock);
```

For a core that does not currently own the mutex to read the value of `lock` or `lock_pid`, the core must first invalidate the cache line. This causes the core to read the current value of the variable from system memory instead of using the locally cached copy. The following code demonstrates how the `lock_pid` value can be read.

```
xthal_dcache_line_invalidate(lock_pid);
pid = *lock_pid;
```

Cache lines are invalidated and written-back using HAL calls provided by the Xtensa compile-time HAL. The compile-time HAL is described in the *Xtensa System Software Reference Manual*.

The example code assumes that both `lock` and `lock_pid` is assigned so that each is the only data object on a cache line. If this is not the case, then invalidating the cache line containing `lock` or `lock_pid` could incorrectly invalidate other data, leading to incorrect program execution.

## 4 Maintaining the Program Order of Memory Operations

The *program order* of memory load and store operations refers to the order of those operations as coded in the application. Maintaining program order for some memory operations is important to ensure correct behavior of synchronization operations. The C/C++ compiler can generate code that changes the order of load and store operations. For example, if the compiler determines that two memory operations access different memory locations, then the compiler may reorder those memory operations during instruction scheduling to improve application performance. The Xtensa hardware can also reorder memory operations that access different memory addresses. A correct implementation of mutex and barrier operations must explicitly enforce program memory ordering where it is required.

The following example demonstrates the potential problem. The example shows a portion of code at the end of a critical region and the code that unlocks the mutex guarding that region. Within the critical region, a value is stored to a variable shared between multiple processors, `shared_var`. Because the assignment to `shared_var` is guarded by a mutex, only one core at a time should be allowed to perform an assignment to that variable.

```
...

/* Mutex is locked at this point. Assign to shared variable. */
*shared_var = ...;

/* Unlock the mutex... */
*lock = 0;
xthal_dcache_line_writeback(lock);
```

As written in the above code, either the C/C++ compiler or the Xtensa hardware could potentially move the store to `*lock` so that it occurs before the store to `*shared_var`. If that occurs, other processors in the system may see the mutex unlocked before the store to `shared_var`. This would effectively move the store to `shared_var` outside the critical region protected by the mutex.

To prevent the C/C++ compiler and hardware from reordering memory operations, we can use the `flush_memory` pragma. The following code shows how the `flush_memory` pragma is placed to make sure that the store to `*shared_var` is seen by other processors in the system before the store to `*lock`.

```
...

/* Mutex is locked at this point. Assign to shared variable. */
*shared_var = ...;

/* Unlock the mutex... */
#pragma flush_memory
*lock = 0;
xthal_dcache_line_writeback(lock);
```

For Xtensa LX and LX2 processors, the `flush_memory` pragma is translated into a MEMW instruction. A more efficient implementation is possible for cores that implement the Multiprocessor Synchronization option described in the *Xtensa Instruction Set Architecture (ISA) Reference Manual*. For these cores, we can use the `no_reorder_memory` pragma to prevent the compiler from reordering memory operations and the S32RI instruction to prevent the hardware from reordering memory operations. The following example shows how each of these cases can be handled in the same code by using the compile-time HAL to detect what processor features are available.

```
#include <xtensa/config/core.h>
#if XCHAL_HAVE_S32C1I || XCHAL_HAVE_RELEASE_SYNC
#include <xtensa/tie/xt_sync.h>
#endif

...

/* Mutex is locked at this point. Assign to shared variable. */
*shared_var = ...;

/* Unlock the mutex... */
#if XCHAL_HAVE_RELEASE_SYNC
#pragma no_reorder_memory
XT_S32RI(0, lock, 0);
```

```
#else
#pragma flush_memory
*lock = 0;
#endif

xthal_dcache_line_writeback(lock);
```

## 5 LIBXMP: Xtensa Synchronization Library

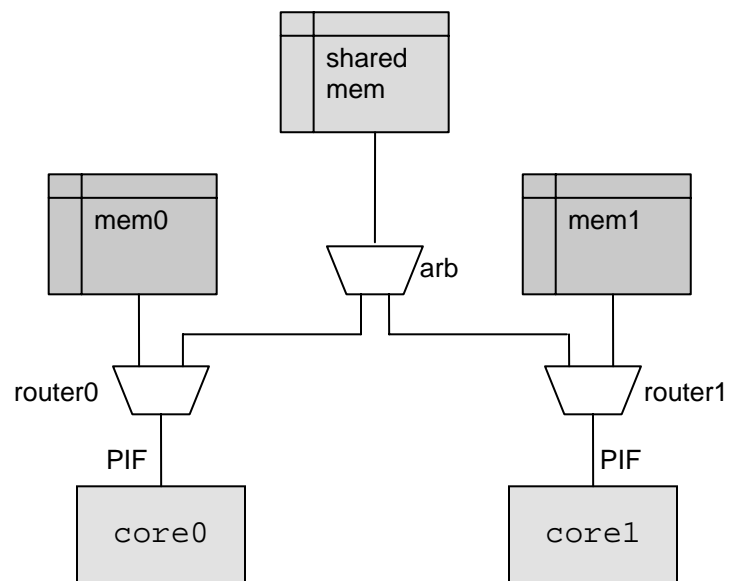
The files included with this application note contain the complete source code for a library that implements memory-based mutex and barrier synchronization primitives. The barrier synchronization primitive is not discussed in detail in this application note, but the library header file, `xmp.h`, contains a detailed description. The implementation correctly handles the issues of data cache coherence and program ordering of memory operations discussed in this application note. Documentation for the library is contained in `xmp.h`.

The library source, header, and makefile are contained in the `libxmp` directory.

## 6 Example Code

The files included with this application note contain example code that shows the usage of some of the mutex and barrier API provided by LIBXMP. The `mutex_test.c` file shows usage for the mutex API and `barrier_test.c` shows usage for the barrier API. A makefile is also included to build and run the examples.

The examples execute on the system shown in the following figure. In the example system, two identical cores each have a large private memory to hold instructions and data, and a small shared memory to hold shared data necessary for synchronization. The makefile simulates this system using the `xtsc-run` application. To use `xtsc-run`, you need a license for XTSC and you need to build `xtsc-run` as described in the *Xtensa XTSC User's Guide*.



You will also need to create a processor configuration that contains the `S32C1I` instruction and a particular memory layout assumed by the examples. Using the Xtensa Processor



Generator within Xplorer, create a new processor configuration with the name `xmp`. Edit the configuration and from the Instructions tab, enable the **Conditional store synchronization instruction** option. From the **Vectors** tab set the System RAM size to 256M and the address to 0x60100000. Set the System ROM size to 128K and the address to 0x60000000. Save, build, and install the configuration.

Once you have created and installed your `xmp` processor configuration, run the mutex example with the following command:

```
make test_mutex
```

Run the barrier example with:

```
make test_barrier
```