APPENDIX D
SYNCHRONIZATION PROGRAMMING EXAMPLES

The examples in this appendix show how synchronization instructions can be used to emulate various synchronization primitives and how to provide more complex forms of synchronization.

For each of these examples, it is assumed that a similar sequence of instructions is used by all processes requiring synchronization of the accessed data.

D.1 General Information

The following points provide general information about the **lwarx** and **stwcx.** instructions:

- In general, **lwarx** and **stwcx.** instructions should be paired, with the same effective address used for both. The exception is an isolated **stwcx.** instruction that is used to clear any existing reservation on the processor, for which there is no paired **lwarx** and for which any (scratch) effective address can be used.
- It is acceptable to execute an **lwarx** instruction for which no **stwcx.** instruction is executed. For example, such a dangling **lwarx** instruction occurs if the value loaded in the test and set sequence shown in **D.3.2 Test and Set** is not zero.
- To increase the likelihood that forward progress is made, it is important that looping on **lwarx**/**stwcx.** pairs be minimized. For example, in the sequence shown above for test and set, this is achieved by testing the old value before attempting the store — were the order reversed, more **stwcx.** instructions might be executed, and reservations might more often be lost between the **lwarx** and the **stwcx.** instructions.
- The manner in which **lwarx** and **stwcx.** are communicated to other processors and mechanisms and between levels of the memory subsystem within a given processor is implementation-dependent. In some implementations performance may be improved by minimizing looping on an **lwarx** instruction that fails to return a desired value. For example, in the test and set example shown above, to stay in the loop until the word loaded is zero, the programmer could change the **bne** S+ 12 to **bne** loop. However, in some implementations better performance may be obtained by using an ordinary load instruction to do the initial checking of the value, as follows:

```
loop:  lwz     rS,0(r3)     #load the word
       cmpwi   r5,0         #loop back if word
       bne     loop         #not equal to 0
       lwarx   rS,0,r3      #try again, reserving
       cmpwi   r5,0         #(likely to succeed)
       bne     loop         #try to store nonzero
       stwcx.  r4,0,r3      #loop if lost reservation
       bne     loop
```

- In a multiprocessor, livelock is possible if a loop containing an **lwarx**/**stwcx.** pair also contains an ordinary store instruction for which any byte of the affected memory area is in the reservation granule of the reservation. For example, the first code sequence shown in **D.5 List Insertion** can cause livelock if two list elements have next element pointers in the same reservation granule.

D.2 Synchronization Primitives

The following examples show how the **lwarx** and **stwcx.** instructions can be used to emulate various synchronization primitives. The sequences used to emulate the various primitives consist primarily of a loop using **lwarx** and **stwcx.**. Additional synchronization is unnecessary, because the **stwcx.** will fail, clearing the EQ bit, if the word loaded by **lwarx** has changed before the **stwcx.** is executed.

D.2.1 Fetch and No-Op

The fetch and no-op primitive atomically loads the current value in a word in memory. In this example it is assumed that the address of the word to be loaded is in GPR3 and the data loaded are returned in GPR4.

```
loop:     lwarx     r4,0,r3     #load and reserve
          stwcx.    r4,0,r3     #store old value if still reserved
          bne       loop        #loop if lost reservation
```

The **stwcx.**, if it succeeds, stores to the destination location the same value that was loaded by the preceding **lwarx**. While the store is redundant with respect to the value in the location, its success ensures that the value loaded by the **lwarx** was the current value (that is, the source of the value loaded by the **lwarx** was the last store to the location that preceded the **stwcx.** in the coherence order for the location).

D.2.2 Fetch and Store

The fetch and store primitive atomically loads and replaces a word in memory.

In this example it is assumed that the address of the word to be loaded and replaced is in GPR3, the new value is in GPR4, and the old value is returned in GPR5.

```
loop:     lwarx     r5,0,r3     #load and reserve
          stwcx.    r4,0,r3     #store new value if still reserved
          bne       loop        #loop if lost reservation
```

D.3 Fetch and Add

The fetch and add primitive atomically increments a word in memory.

In this example it is assumed that the address of the word to be incremented is in GPR3, the increment is in GPR4, and the old value is returned in GPR5.

```
loop:     lwarx     rS,0,r3     #load and reserve
          add       ra,r4,rS    #increment word
          stwcx.    ra,0,r3     #store new value if still reserved
          bne       loop        #loop if lost reservation
```

D.3.1 Fetch and AND

The fetch and AND primitive atomically performs a logical AND of a value and a word in memory.

In this example it is assumed that the address of the word to be ANDed is in GPR3, the value to AND into it is in GPR4, and the old value is returned in GPR5.

```
loop:     lwarx     rS,0,r3     #load and reserve
          and       ra,r4,rS    #AND word
          stwcx.    ra,0,r3     #store new value if still reserved
          bne       loop        #loop if lost reservation
```

This sequence can be changed to perform another Boolean operation atomically on a word in memory, simply by changing the AND instruction to the desired Boolean instruction (OR, XOR, etc.).

D.3.2 Test and Set

The test and set primitive atomically loads a word from memory, ensures that the word in memory contains a non-zero value, and sets the EQ bit of CR field 0 according to whether the value loaded is zero.

In this example it is assumed that the address of the word to be tested is in GPR3, the new value (non-zero) is in GPR4, and the old value is returned in GPR5.

```
loop:     lwarx     r5,0,r3     #load and reserve
          cmpwi     r5,0        #done if word
          bne       $+12        #not equal to 0
          stwcx.    r4,0,r3     #try to store nonzero
          bne       loop        #loop if lost reservation
```

Test and set is shown primarily for pedagogical reasons. It is useful on machines that lack the better synchronization facilities provided by **lwarx** and **stwcx.**. Test and set does not scale well. Using test and set before a critical section allows only one process to execute in the critical section at a time. Using **lwarx** and **stwcx.** to bracket the critical section allows many processes to execute in the critical section at once, but at most one will succeed in exiting from the section with its results stored.

Depending on the application, if test and set fails (that is, clears the EQ bit of CR field 0) it may be appropriate to re-execute the test and set.

D.4 Compare and Swap

The compare and swap primitive atomically compares a value in a register with a word in memory. If they are equal, it stores the value from a second register into the word in memory. If they are unequal, it loads the word from memory into the first register, and sets the EQ bit of the CR0 field to indicate the result of the comparison.

In this example it is assumed that the address of the word to be tested is in GPR3, the comparand is in GPR4, the new value is in GPR5, and the old value is returned in GPR6.

```
lwarx     r6,0,r3     #load and reserve
cmpw      r4,r6       #first 2 operands equal ?
bne       $+8         #skip if not
stwcx.    rS,0,r3     #store new value if still reserved
```

Compare and swap is shown primarily for pedagogical reasons. It is useful on machines that lack the better synchronization facilities provided by **lwarx** and **stwcx.**. A major weakness of typical compare and swap instructions is that they permit spurious success if the word being tested has changed and then changed back to its old value: the sequence shown above does not have this weakness.

Depending on the application, if compare and swap fails (that is, clears the EQ bit of CR0) it may be appropriate to recompute the value potentially to be stored and then re-execute the compare and swap.

D.5 List Insertion

The following example shows how the **lwarx** and **stwcx.** instructions can be used to implement simple LIFO (last-in-first-out) insertion into a singly-linked list. (Complicated list insertion, in which multiple values must be changed atomically, or in which the correct order of insertion depends on the contents of the elements, cannot be implemented in the manner shown below, and requires a more complicated strategy such as using locks.)

The next element pointer from the list element after which the new element is to be inserted, here called the parent element, is stored into the new element, so that the new element points to the next element in the list: this store is performed unconditionally. Then the address of the new element is conditionally stored into the parent element, thereby adding the new element to the list.

In this example it is assumed that the address of the parent element is in GPR3, the address of the new element is in GPR4, and the next element pointer is at offset O from the start of the element. It is also assumed that the next element pointer of each list element is in a reservation granule separate from that of the next element pointer of all other list elements.

```
loop:     lwarx     r2,0,r3     #get next pointer
          stw       r2,0(r4)    #store in new element
          sync                  #let store settle (can omit if not
MP)
          stwcx. r  4, a, r3    #add new element to list
          bne       loop        #loop if stwcx. failed
```

In the preceding example, if two list elements have next element pointers in the same reservation granule then, in a multiprocessor, livelock can occur. (Livelock is a state in which processors interact in a way such that no processor makes progress.)

If it is not possible to allocate list elements such that each element's next element pointer is in a different reservation granule, then livelock can be avoided by using the following, more complicated, code sequence.

```
          lwz       r2,0(r3)    #get next pointer
loopl:    mr        r5,r2       #keep a copy
          stw       r2,0(r4)    #store in new element
          sync                  #let store settle
loop2:    lwarx     rZ,0,r3     #get it again
          cmpw      r2,r5       #loop if changed (someone
          bne       loopl       #else progressed)
          stwcx.    r4,0,r3     #add new element to list
          bne       loop2       #loop if failed
```