Memory Cycle Accountings for Hardware-Assisted Real-Time Garbage Collection

TR 91-21c
Kelvin Nilsen

May 5, 1992

Iowa State University of Science and Technology
Department of Computer Science
226 Atanasoff
Ames, IA 50011

Memory Cycle Accountings for Hardware-Assisted Real-Time Garbage Collection *

Kelvin Nilsen

ISU TR 91-21(c)

ABSTRACT

Hardware-assisted garbage collection makes use of dedicated circuits located within a special expansion memory module to enhance the response time and throughput of garbage collection operations. This paper provides detailed descriptions of the memory cycles required to implement each of the primitive garbage collection operations provided by the hardware-assisted garbage collection module.

17 November 1992

Department of Computer Science

Iowa State University

226 Atanasoff Hall

Ames, Iowa 50011-1040

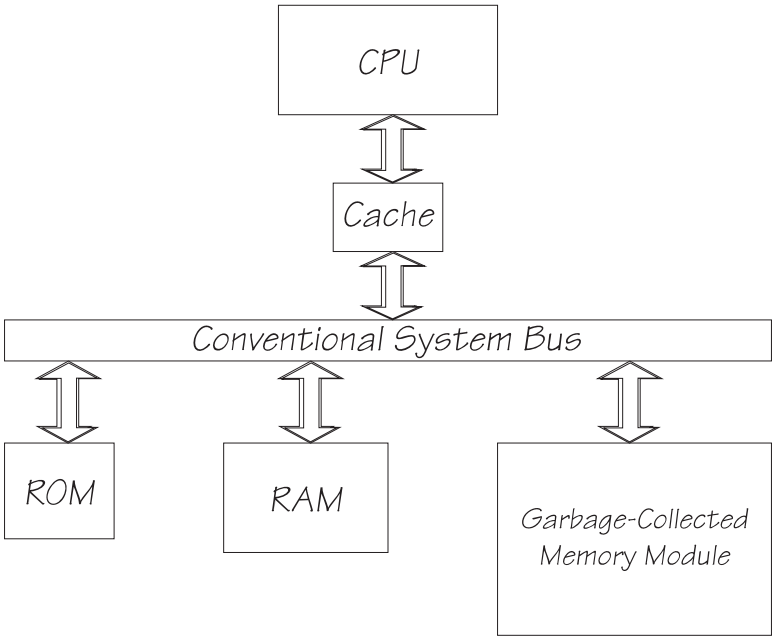
*This work was supported by the National Science Foundation under Grant MIP-9010412.

Memory Cycle Accountings for Hardware-Assisted Real-Time Garbage Collection *

Kelvin Nilsen

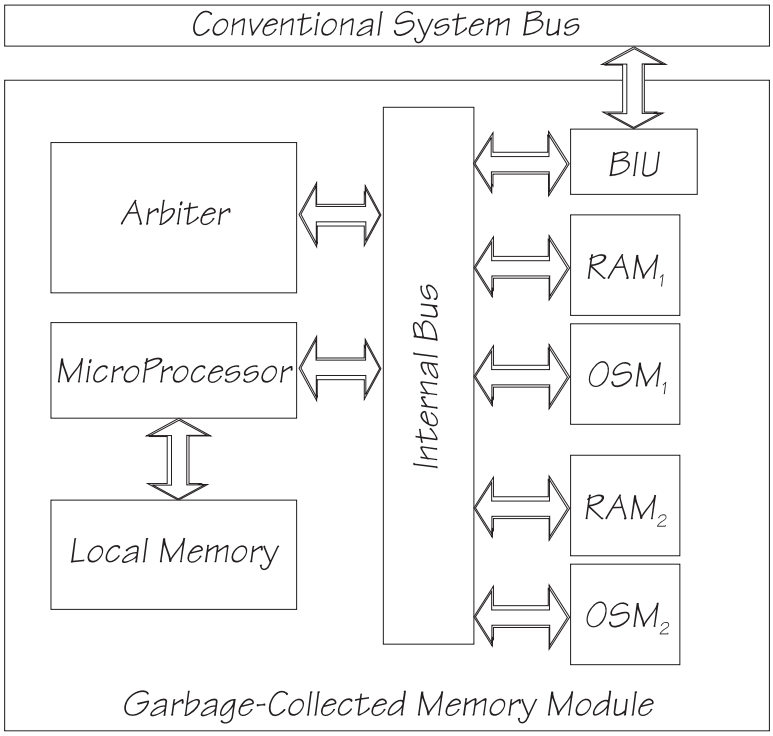
1. Background

A complete description of the hardware-assisted garbage-collection architecture and algorithm is provided in references [1-3]. The overall system architecture is illustrated below:



The garbage-collected memory module plays the role of traditional expansion memory within a standard bus-oriented system architecture. Application processes run on the CPU and garbage-collection tasks run within the garbage-collected memory module. Throughout the remainder of this paper, tasks running on the CPU are collectively referred to as mutators, since insofar as garbage collection is concerned, their only role is to modify (or mutate) the contents of heap-allocated memory. The internal organization of the garbage-collected memory module is as shown:

*This work was supported by the National Science Foundation under Grant MIP-9010412.
Last revised 10/20/92.



In the illustration above, BIU is an abbreviation for *Bus Interface Unit*. The BIU provides an interface between the system bus and an internal bus used for communication between components within the garbage-collected memory module. Each word of RAM is accompanied by a one-bit tag that distinguishes pointers from non-pointers. OSM stands for *Object Space Manager*. Each OSM module manages one RAM memory module by maintaining a data base of locations at which each object residing in the memory module begins. Given a pointer to any location within a memory module, the corresponding OSM is capable of reporting the address of the start of the object that contains that address in approximately the same time required to perform a traditional memory read or write. The OSM's primitive operations are reset, which reinitializes the OSM, createHeader, which installs an object into the OSM's data base, and findHeader, which reports the beginning address of the object containing a particular address. The *Arbiter* oversees access to the internal bus, and performs a number of important garbage collection activities using circuitry dedicated to providing rapid context switching between background garbage collection activities and mutator demands. The *microprocessor*'s main responsibility is to oversee garbage collection by issuing commands to the arbiter. The arbiter works on commands from the microprocessor as a background activity, giving highest priority to servicing BIU requests. Note that the organization illustrated above allows multiple memory and OSM components to work in parallel.

We assume that the arbiter implements a 3-slot write buffer for each of the two memory spaces. Furthermore, we assume that each OSM can buffer one createHeader request. In other words, we assume that as long as sufficient time has passed since a preceding OSM request has been issued, a createHeader request completes instantly. Furthermore, subsequent findHeader requests need not wait for the buffered createHeader request to complete.

The flow-chart illustrations that follow do not necessarily represent actual code fragments. Rather, they abstract the sequence of actions and memory accesses that must be performed in order to implement particular operations. In some cases, a single block of hardwired code corresponds to multiple distinct blocks in the diagrammed flow charts. Many low-level details, such as

the implementation of memory alignment restrictions, are omitted from the flowcharts. The C++-like code fragments that describe arbiter operations below make frequent reference to the following variables and data structures:

```
const int SemiSpaceSize;      // the number of words in each semi-space

struct MemWord {
  int value;                  // the data stored in a particular location
  int ptrtag:1;               // 1 bit to distinguish descriptors
} Memory[2 * SemiSpaceSize], *toMem, *fromMem;
// toMem and fromMem point into Memory

Address
  Relocated,                  // points to next object to be copied into to-space
  Reserved,                   // points to next word of to-space to be reserved for copying
  New,                        // points to most recently allocated object in to-space
  CopySrc,                    // points to next word to be copied out of from-space
  CopyDest,                   // points to next to-space location to be copied
  CopyEnd;                    // points one word beyond the to-space object
                              //  currently being copied

struct ExplicitCache {
  struct MemWord data;        // data cached from memory
  Address addr;               // location of cached data
} CREG;                       // holds the memory value currently being scanned
                              //  or copied by the garbage collector

class OSM {
  // clear the OSM data base
  void reset();

  // create a new object
  void createHeader(Address addr, int numWords);

  // find the beginning address of the object containing a particular address
  Address findHeader(Address derivedAddr);
} toOSM, fromOSM;
```

We use comma-separated parenthesized lists of values to represent record constructors. We assume that the translator infers the type of constructors from their context.

In the flow chart illustrations that follow, a rectangle with a light border represents an action that can be implemented without access to the memory subsystem. Rectangles with a heavy border represent actions that normally incur the overhead of a memory access. A cloud symbol is used to represent the cost of interrupting background garbage collection activities. Depending on the amount of specialized circuitry dedicated to minimizing this latency, the time represented by this action ranges from a single processor cycle to multiple memory cycles. In our memory cycle accountings, we assume that the worst-case time required to interrupt background garbage collection activities is one memory cycle. A hexagon is used to highlight actions used to control concurrency between background garbage collection activities and the high-priority services provided to the mutator.

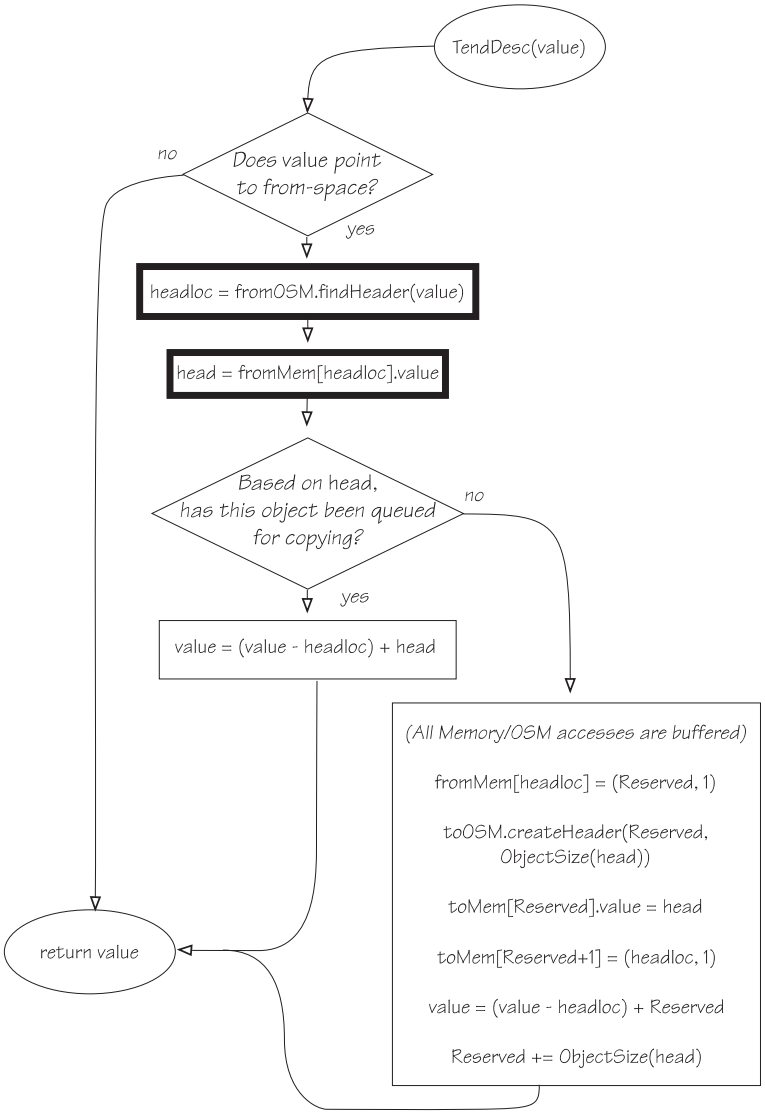
In diagramming the control flow associated with memory allocation, we omit the test to determine whether garbage collection must be initiated and omit the pacing between garbage collection and allocation. Neither of these concerns requires any access to the memory subsystem.

2. Operations Provided by the Arbiter to the Mutator

The highest priority responsibilities of the arbiter are to service requests from the mutator. Between servicing mutator requests, the arbiter dedicates itself to background garbage collection activities. Certain sequences of garbage collection activities are atomic. These are described in more detail in §3. The remainder of this section discusses in detail each of the primitive operations provided to the mutator.

TendDesc

Tend a single descriptor, returning its updated value.



In the worst-case, two memory cycles are required to tend a descriptor. The standard protocol guarantees that garbage collection is not active during times when TendDesc is invoked, so there is no need to interrupt background garbage collection activities. Note that we are guaranteed sufficient space in the write buffers for the buffered writes illustrated above because the block that performs the buffered writes is preceded by two memory cycles, neither of which accesses the *to-space* memory or OSM. Only one of the preceding memory cycles accesses *from-space* memory.

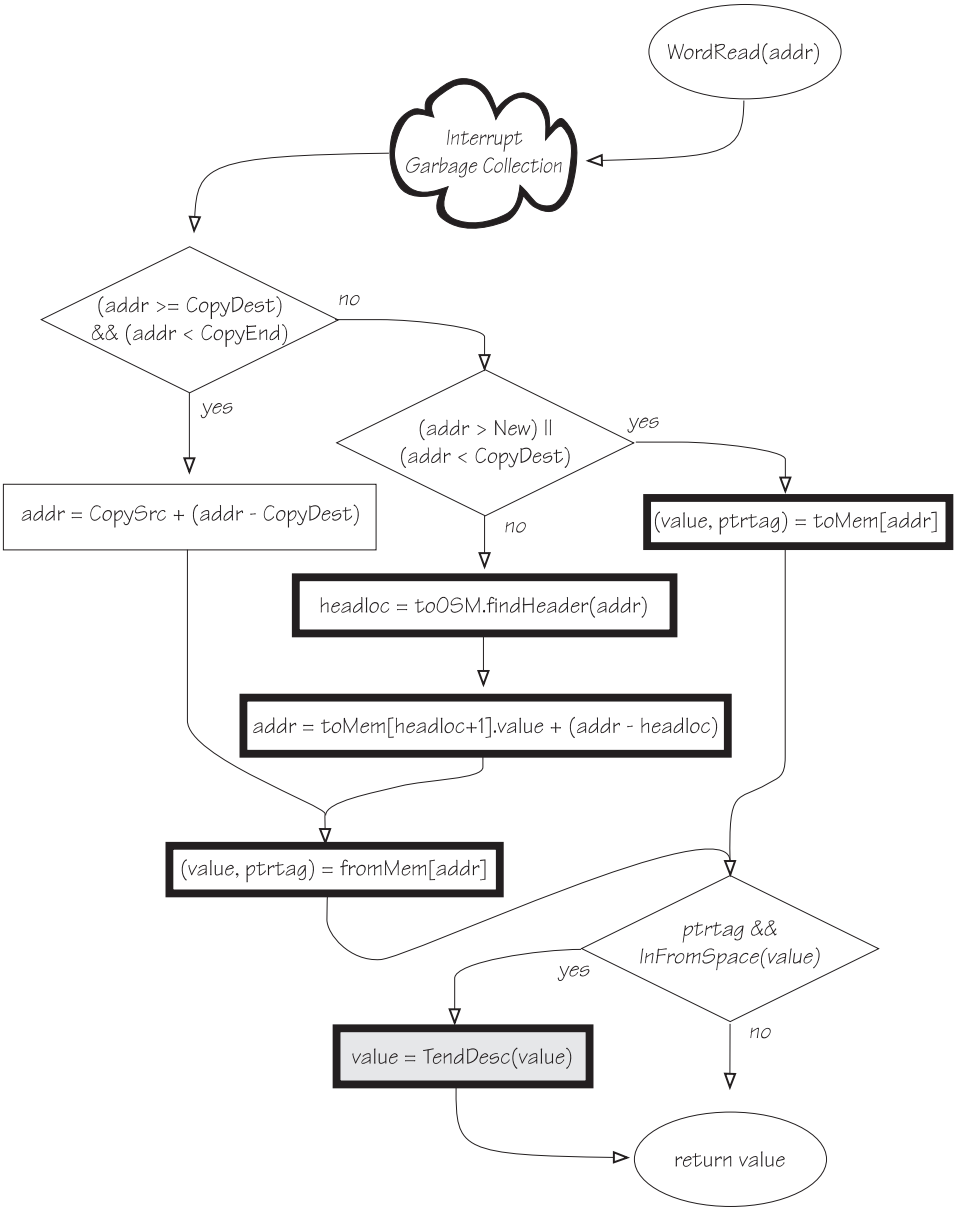
TendingDone

Report to the garbage collector that all descriptors have been tended so the garbage collector may begin scanning and copying of live data.

Since this operation requires no access to memory, no flow chart illustration is provided.

WordRead

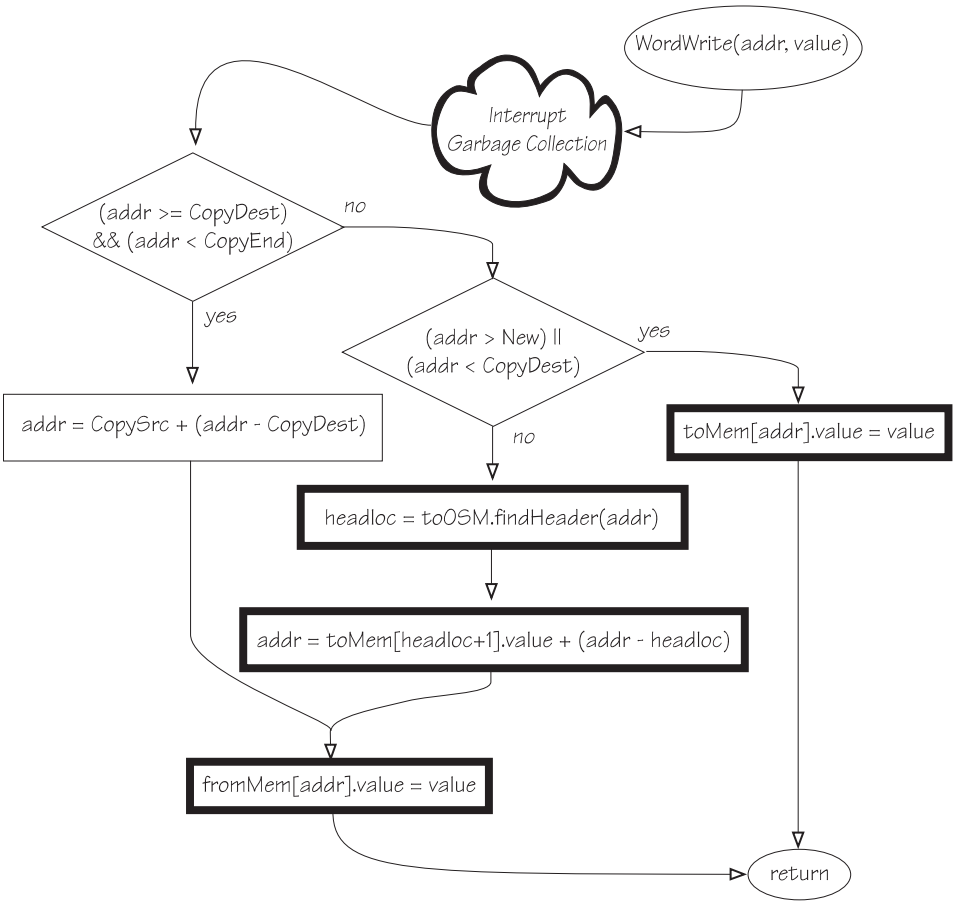
Read a single word from memory. This operation services traditional mutator fetches that refer to garbage-collected memory.



Including the costs of interrupting garbage collection, the worst-case number of memory cycles required to service a read request is four plus the cost of executing a TendDesc instruction. Thus, the total worst-case cost to read a word of memory is six memory cycles.

WordWrite

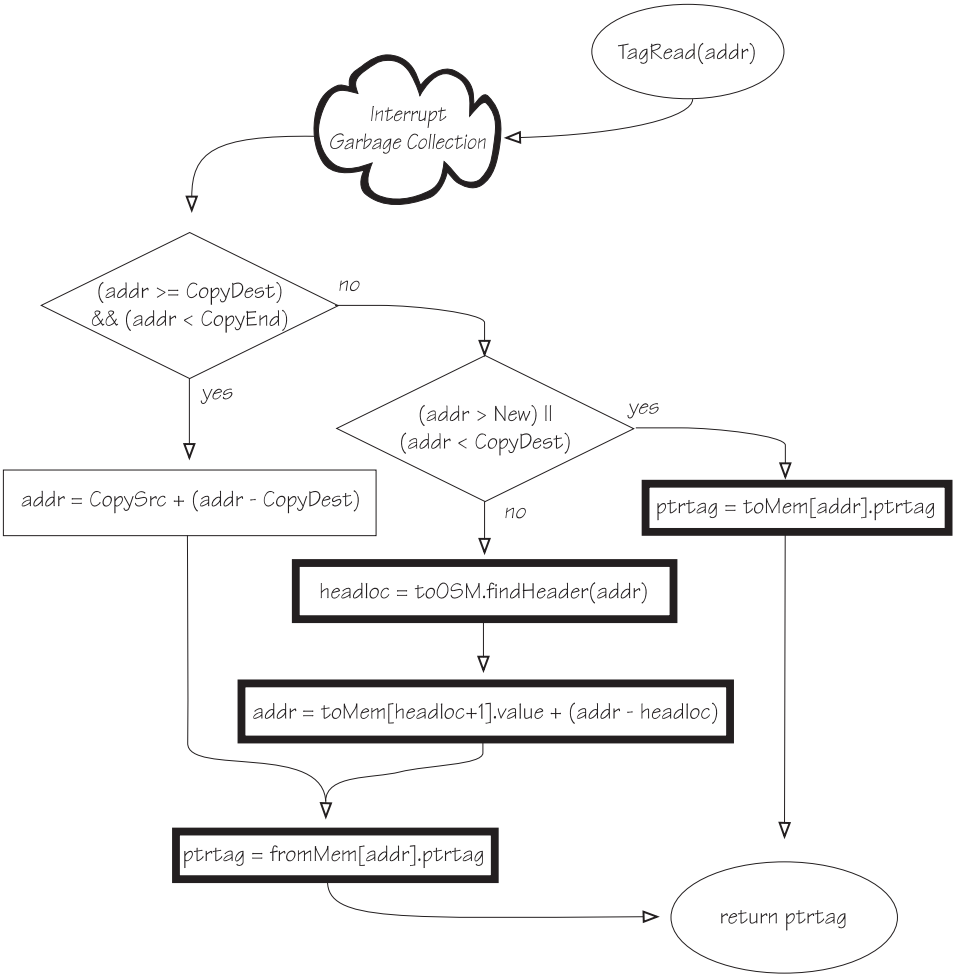
Write a single word to memory. This operation is invoked in response to a traditional mutator store operation that refers to garbage-collected memory.



The longest path through the WordWrite function makes four memory accesses. However, the last of these four memory accesses can be buffered. Thus, the worst-case total cost to write a word of memory is three memory cycles.

TagRead

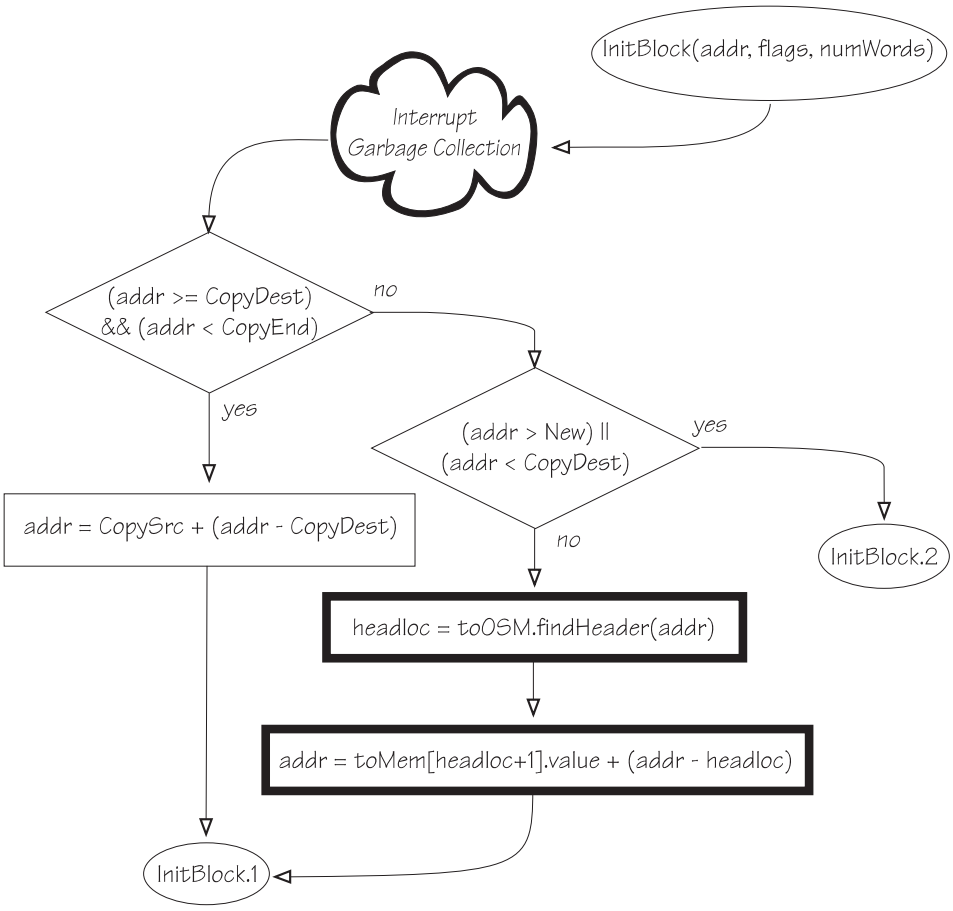
Read the descriptor tag associated with the word at a particular memory location.



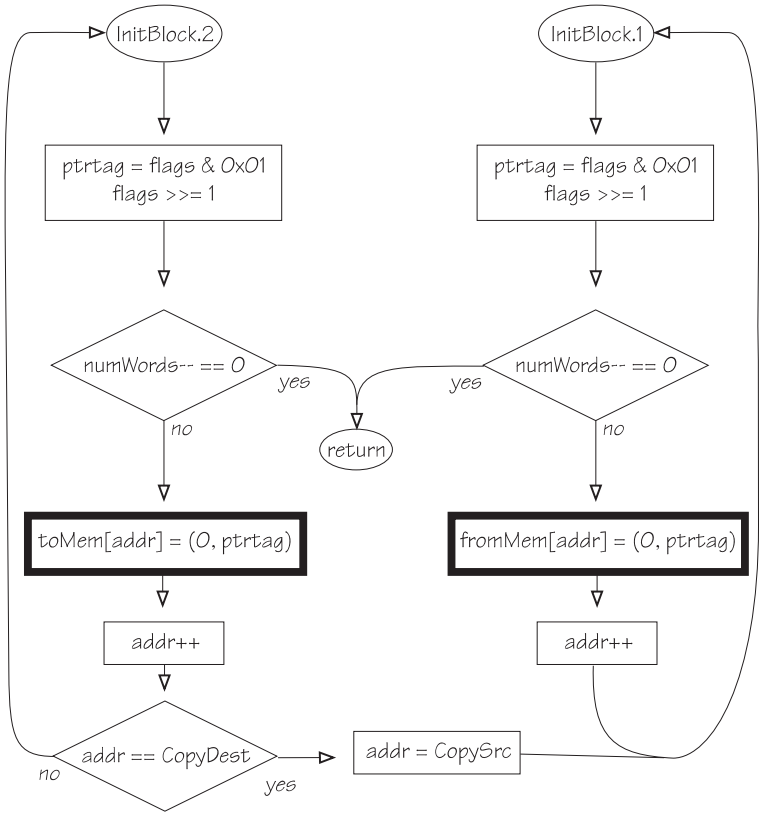
The worst-case path through the TagRead function requires four memory cycles.

InitBlock

Initialize numWords of memory starting at addr to zero. numWords is less than or equal to 32. flags is a 32-bit mask with one bit for each of the words to be initialized, the least significant bit corresponding to the first word to be initialized. A non-zero flags bit signifies that the corresponding memory word holds a descriptor.



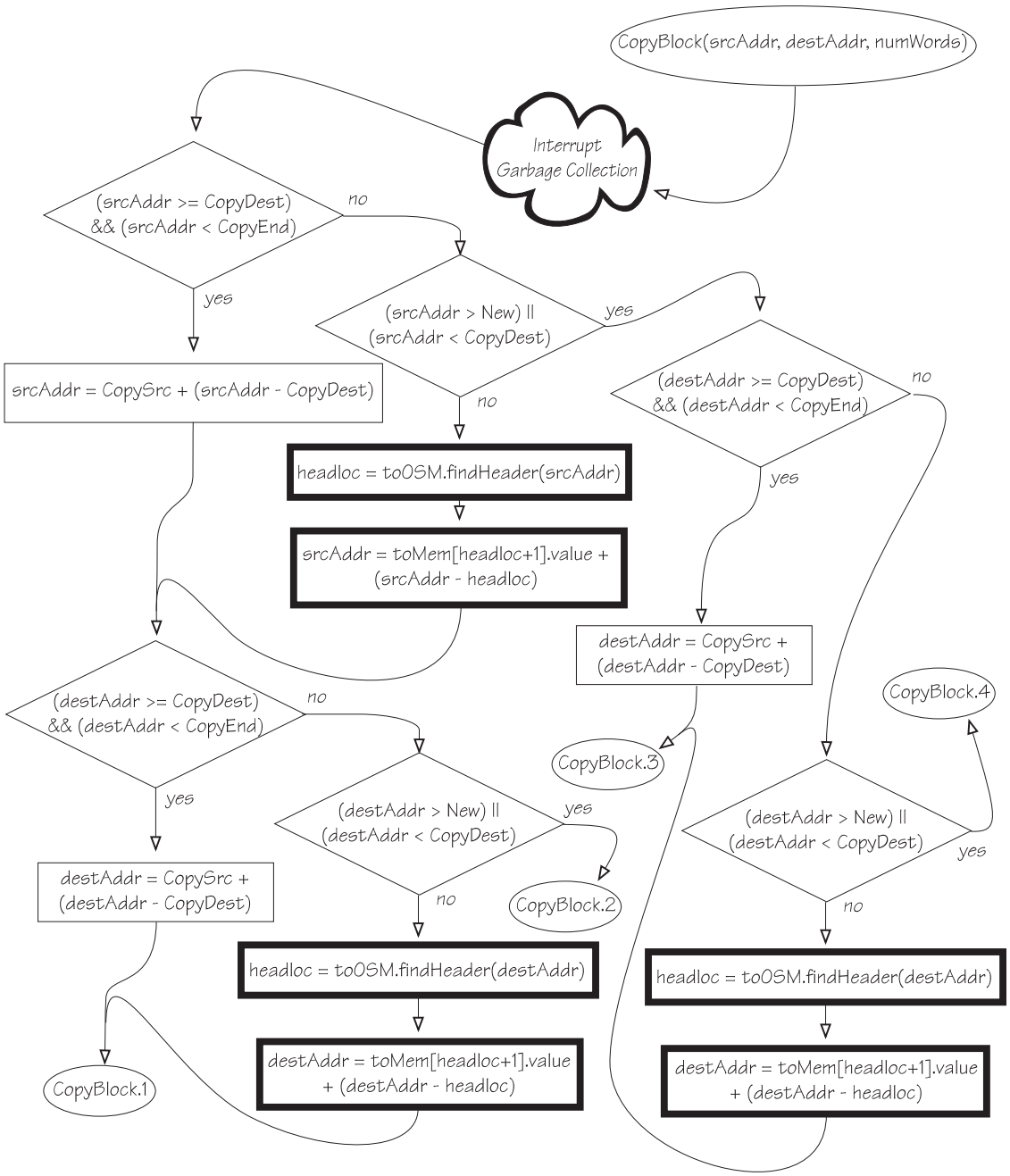
The startup code for InitBlock, shown above, requires in the worst case three memory cycles to complete. The startup code is followed by iterative execution of the following:



In the flow chart above, the InitBlock.1 label corresponds to the case in which the data to be initialized currently resides in *from-space*. The InitBlock.2 label provides handling for data newly allocated from or already copied into *to-space*. Note that control passes from the InitBlock.2 loop to the InitBlock.1 loop whenever the mutator requests to initialize a portion of the object that is currently being copied into *to-space*. In total, numWords memory accesses are required to complete the loop iterations. Whenever numWords is greater than or equal to two, and the InitBlock startup costs are charged as three memory cycles, at least two of the iterative memory accesses can be buffered. Therefore, the total number of memory cycles required to implement InitBlock is numWords plus one.

CopyBlock

Copy numWords words of data with accompanying descriptor tags from srcAddr to destAddr. We assume that both the source and destination memory regions each reside entirely within a single object. Memory words are copied in ascending order.



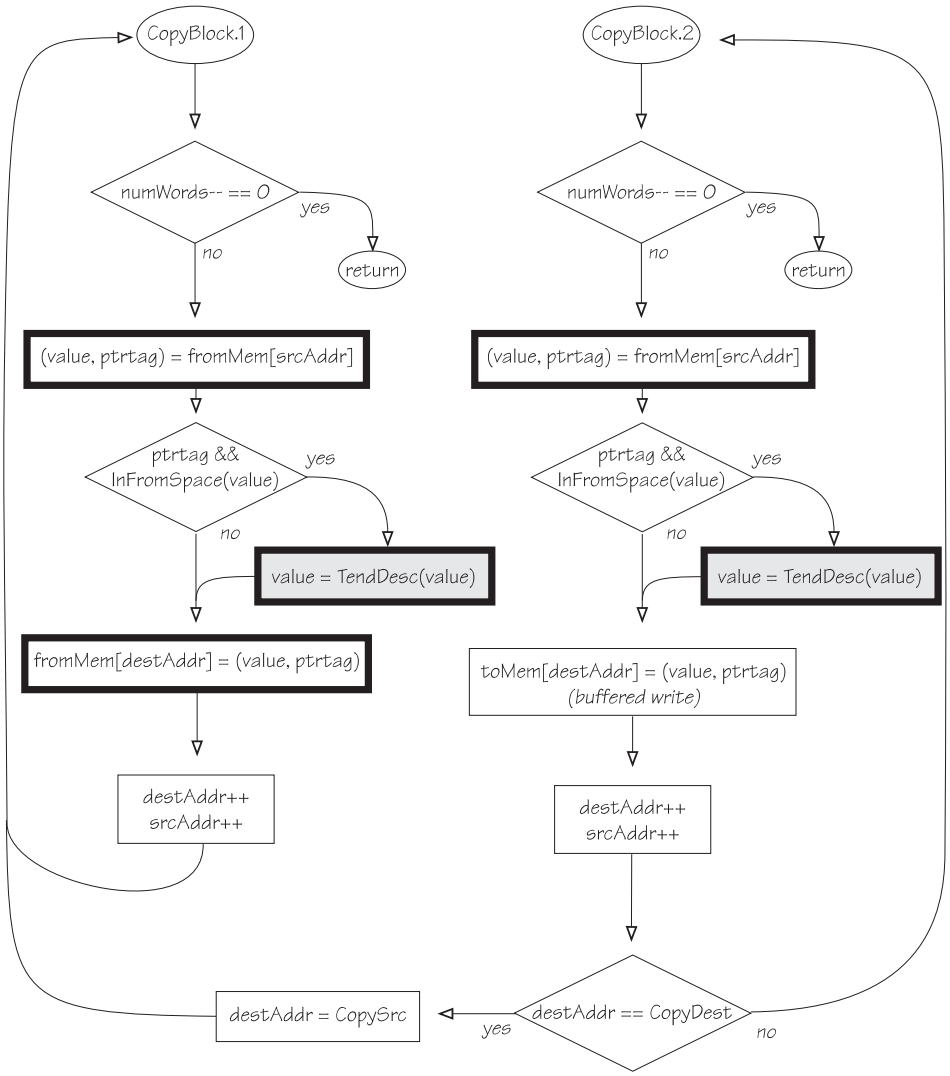
The startup code for CopyBlock requires, in the worst case, five memory cycles. The purpose of the startup code is to determine which memory regions hold the source and destination data and where the data resides within the appropriate region. The four labels denote the following:

CopyBlock.1 Source and destination data resides in *from-space*.

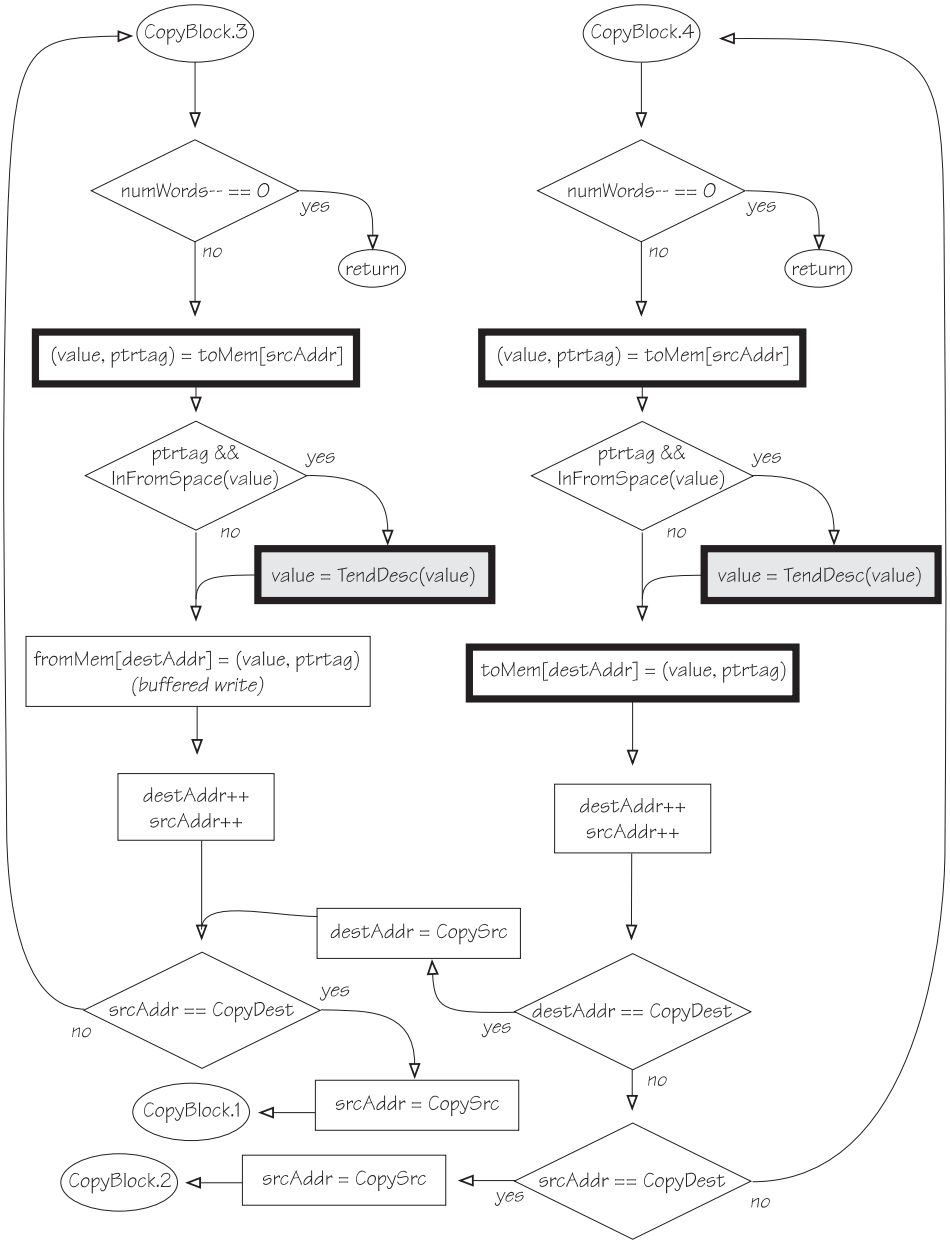
CopyBlock.2 Source data resides in *from-space*. Destination data resides in *to-space*.

CopyBlock.3 Source data resides in *to-space*. Destination data resides in *from-space*.

CopyBlock.4 Source and destination data resides in *to-space*.



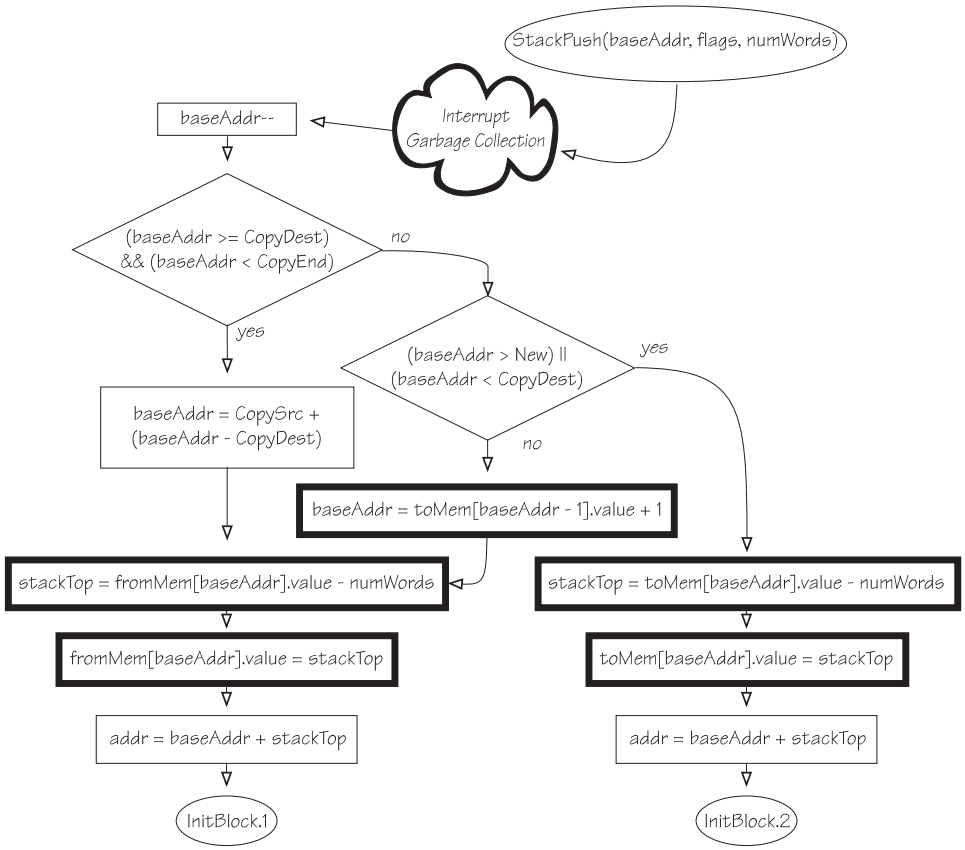
Each iteration of the CopyBlock.1 loop requires, in the worst case, four memory cycles. Each CopyBlock.2 iteration costs three memory cycles in the worst case because one of TendDesc's memory cycles can overlap CopyBlock.2's write to *to-space*. For similar reasons, the worst-case costs of executing each iteration of the CopyBlock.3 and CopyBlock.4 loops are three and four memory cycles respectively.



The worst-case total cost, therefore, of executing a CopyBlock instruction is five plus four times numWords memory cycles. An even tighter bound on memory cycles could be derived by accounting more carefully for the write buffering made possible by the CopyBlock startup code.

StackPush

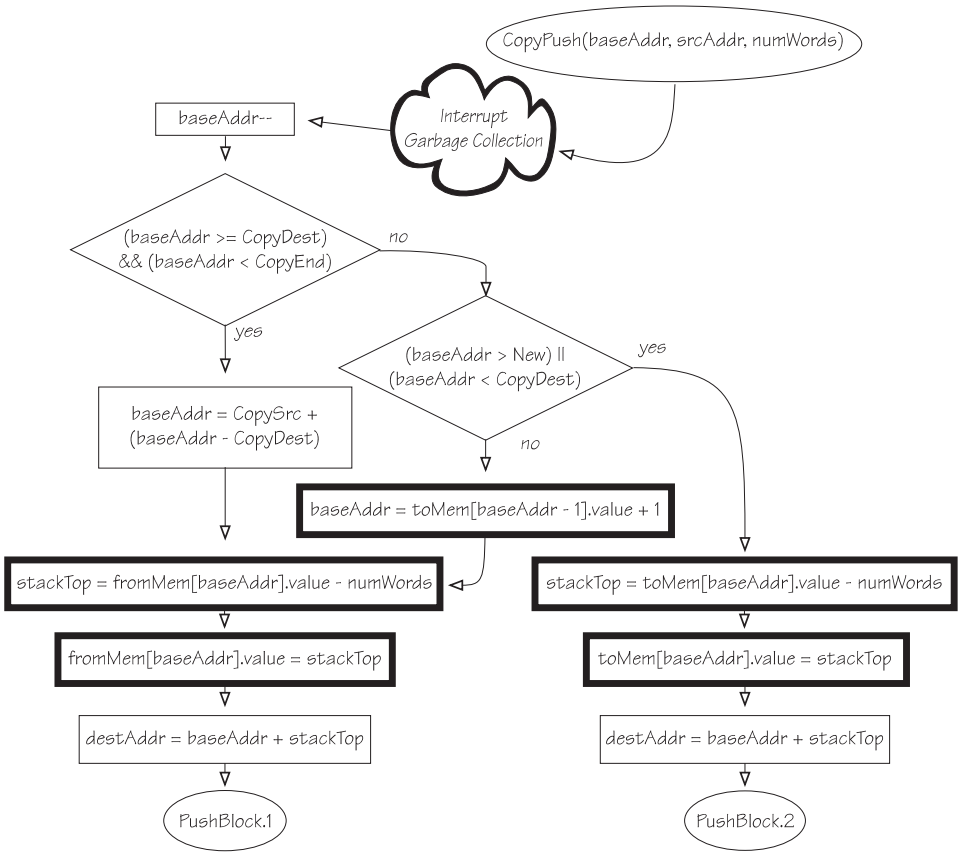
Increase the live portion of the stack based at baseAddr by numWords, initializing each of the stack-allocated words to zero, and setting descriptor tags according to flags. The flags parameter is interpreted as outlined in the InitBlock description above.



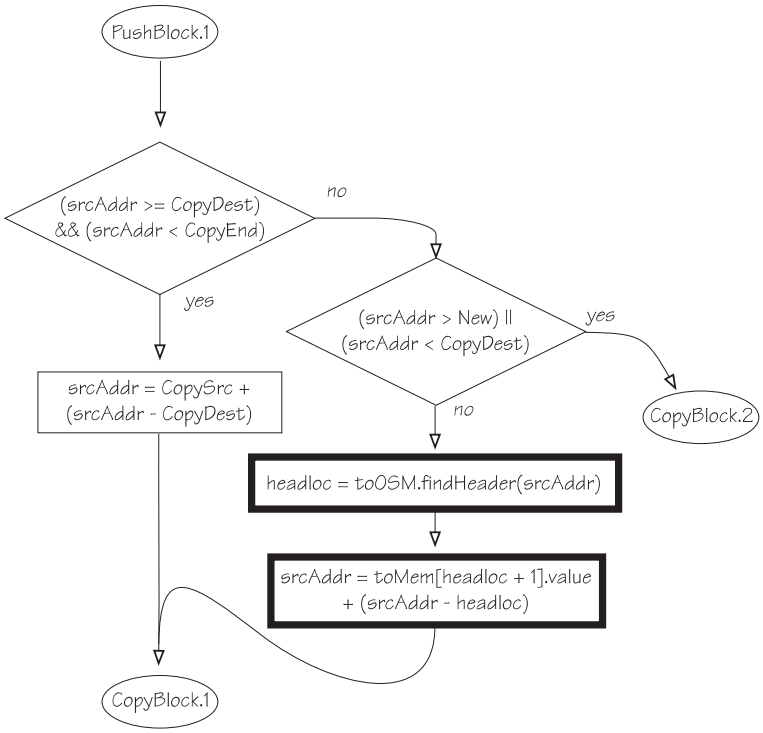
Every path through the StackPush startup code requires, in the worst case, three memory cycles. There is one path through this code that appears to require four memory cycles. However, the last memory write on that path can always be buffered, since one of the preceding three memory cycles on that path reads from *to-space* without requiring any access to *from-space*. Execution of the startup code is followed by *numWords* iterations of the InitBlock.1 or InitBlock.2 loops, each of which costs one memory cycle. Thus, the total cost of executing a StackPush operation is three plus numWords memory cycles.

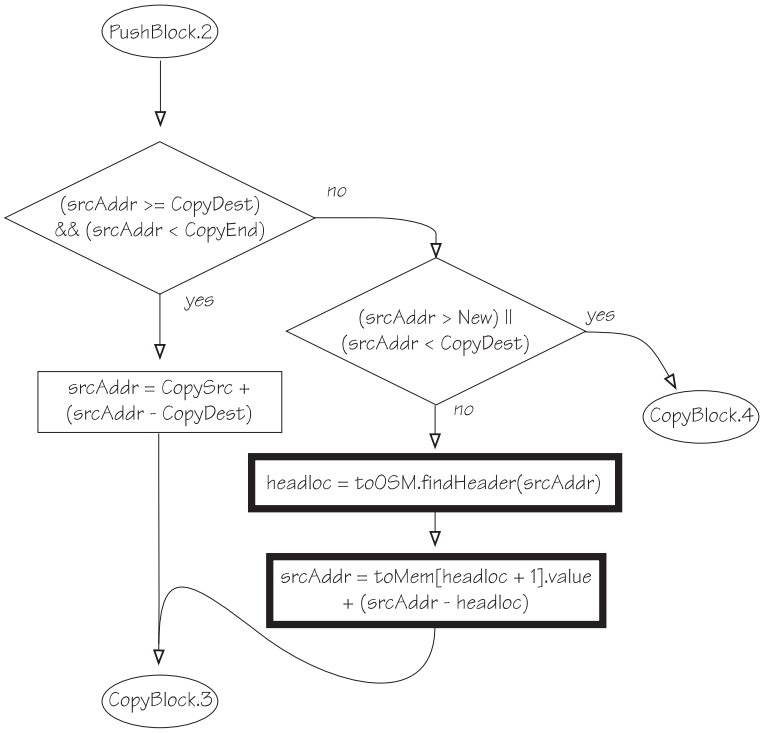
CopyPush

Copy numWords words of data with accompanying descriptor tags from srcAddr onto the top of the stack found at baseAddr, expanding the stack before the data is copied. Assume that both the source and destination memory regions each reside entirely within a single object.



Every path through the CopyPush startup code illustrated above requires, in the worst case, three memory cycles. There is one path through this code that appears to require four memory cycles. However, the last memory write on that path can always be buffered, since one of the preceding three memory cycles on that path reads from *to-space* without requiring any access to *from-space*. The startup code above is followed by execution of the additional startup code associated with either the PushBlock.1 or PushBlock.2 labels, as illustrated below.



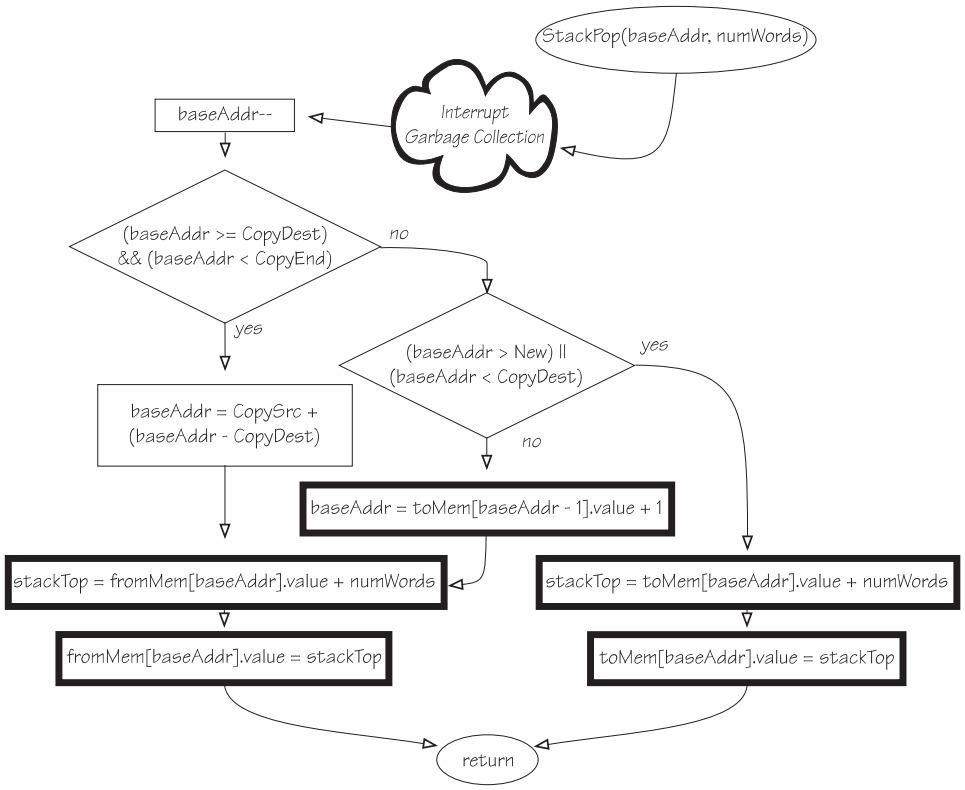


The worst-case requirements of the PushBlock.1 and PushBlock.2 routines are each two memory cycles, but the first memory cycle of each can overlap with the last memory cycle of the CopyPush code described above. This code is followed by *numWords* iterations of the

appropriate CopyBlock subroutine, which are discussed above. Each CopyBlock iteration costs, in the worst case, four memory cycles. So the worst case total cost for execution of CopyPush is four plus four times numWords memory cycles. An even tighter bound on memory cycles could be derived by accounting more carefully for the write buffering made possible by the CopyPush startup code.

StackPop

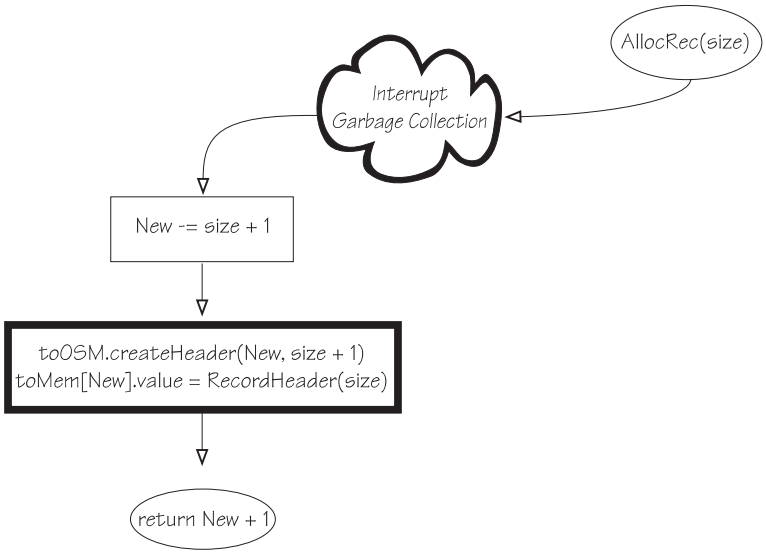
Shrink the size of the active stack by numWords.



Every path through the StackPop code requires, in the worst case, three memory cycles. There is one path through this code that appears to require four memory cycles. However, the last memory write on that path can always be buffered, since one of the preceding three memory cycles on that path reads from *to-space* without requiring any access to *from-space*.

AllocRec

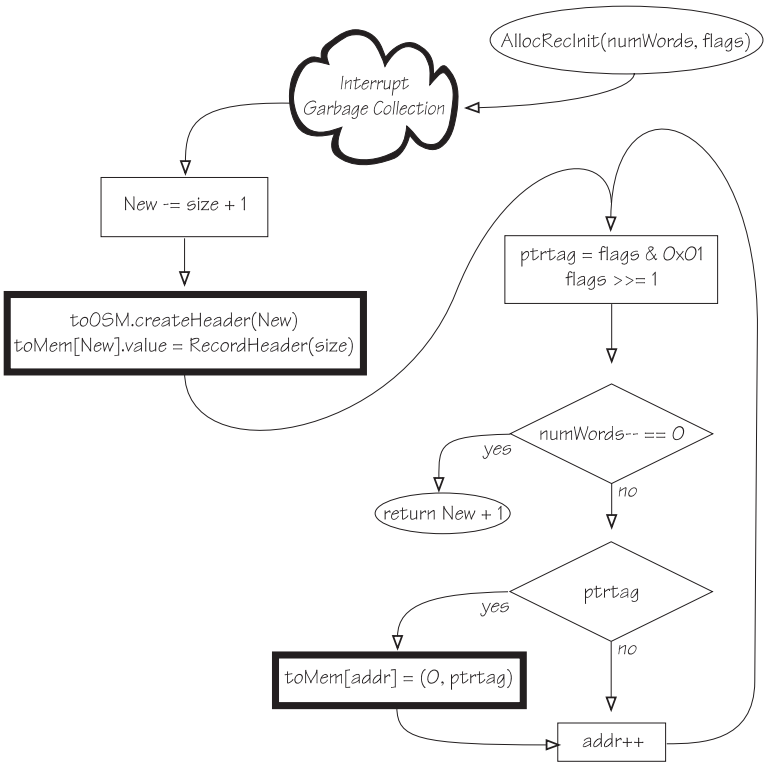
Allocate a record of size numWords.



Two memory cycles are required to allocate a record.

AllocRecInit

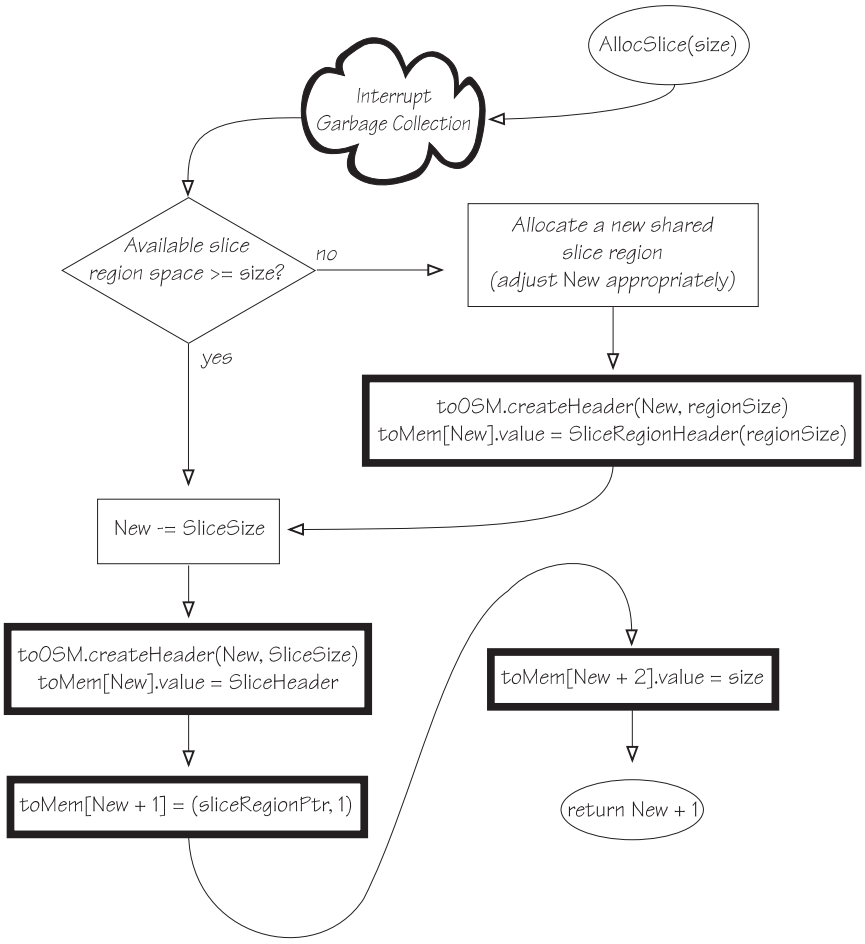
Allocate a record of size numWords ≤ 32, initializing the descriptor tag of each word according to flags.



Only two memory cycles are required to implement the startup code. An additional memory cycle is required for each non-zero bit in AllocRecInit's flags argument.

AllocDSlice/AllocTSlice

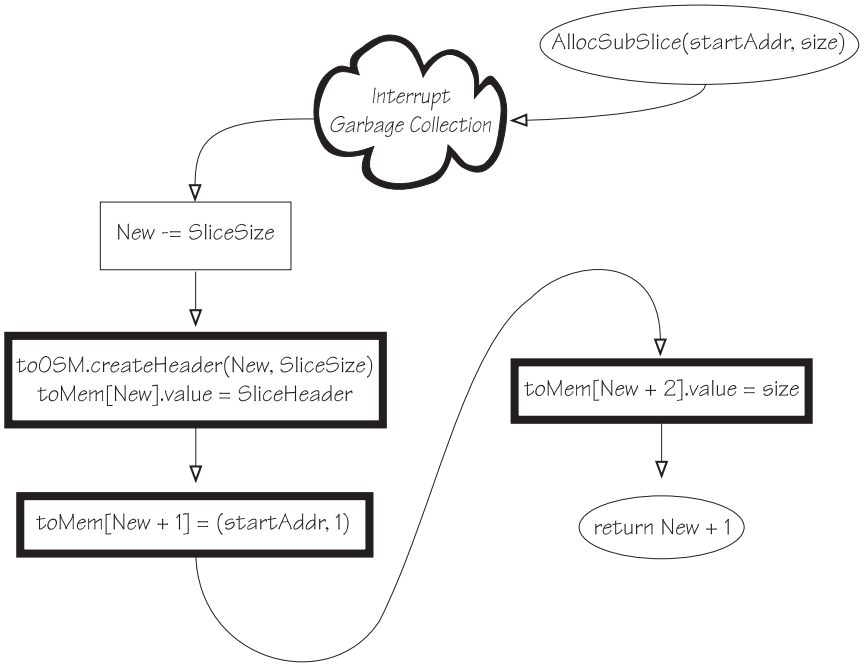
Allocate numWords slice region data and a slice object that references the allocated slice region data, returning a pointer to the slice object. To provide efficient support for large numbers of short slice region allocations, the size of newly allocated slice regions is generally larger than is required to meet the immediate slice region needs. Before creating new slice regions in response to subsequent slice allocation requests, the storage allocator first tries to squeeze the requested slice region data out of a previously allocated slice region. The only difference between AllocDSlice and AllocTSlice is the format of the slice header.



A worst-case total of five memory cycles is required to allocate a new slice.

AllocDSubSlice/AllocTSubSlice

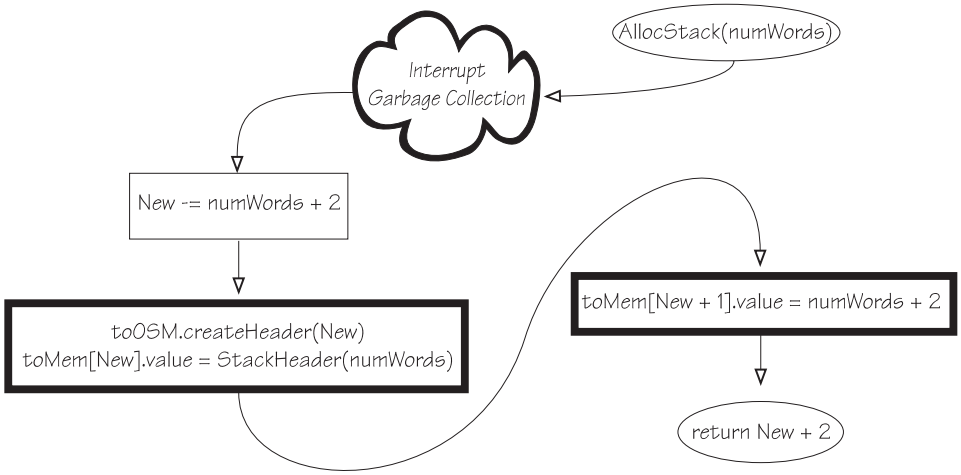
Assuming that startAddr points to a currently live segment of slice region data (containing at least size words of live data starting at startAddr), allocate a new slice object. The only difference between AllocDSubSlice and AllocTSubSlice is the format of the slice header.



Four memory cycles are required in the worst case to allocate a subslice.

AllocStack

Allocate a stack with room to hold numWords of data, returning a pointer to the first of the allocated words. The stack, which grows downward, is initially empty.



Three memory cycles are required in the worst case to allocate a stack.

3. Arbiter Support for the Garbage-Collection Microprocessor

Garbage collection executes as a low-priority task under the direction of the microprocessor. Whenever the mutator requires access to garbage-collected memory, garbage collection is interrupted to service the request. Care must be taken when interrupting the garbage collector to ensure that the integrity of garbage collection is not violated. For example, one of the garbage-collection tasks implemented by the arbiter is to scan a region of memory. Scanning consists of reading each word in the region, tending any descriptors, and overwriting their values with the results of tending. If the mutator stores to the word currently being scanned after the garbage collector has fetched the word but before the updated word has been rewritten to memory, then it is important that the garbage collector abort (or restart) scanning of that particular word. Similar concurrency control is necessary whenever the garbage collector enqueues an object onto the copy queue. The enqueue operation consists of reserving space in *to-space* into which the object will eventually be copied, overwriting the title of the original object with a forwarding pointer to the new location for the object, and writing a title and source pointer into the first two words of the space reserved for eventual copying. Suppose the mutator interrupts garbage collection after space has been reserved for an object to be copied, but before any of the links have been written to memory. If the mutator requests to fetch a pointer that happens to refer to the same object that was being queued for copying, the arbiter will automatically place the object on the copy queue and update the pointer before returning the pointer's value. Since the interrupted garbage collection operation has not yet overwritten the original object's title with a forwarding pointer, the arbiter does not know that memory has already been set aside for the *to-space* copy of the object. Thus, without special concurrency controls, the arbiter would reserve a new block of memory for the object's copy. Then, when the interrupted garbage collection operation is resumed, the forwarding pointer for the original object would be overwritten to point to the memory originally reserved for the object's copy. The result of this is that some of the pointers originally referring to the object are updated to point to the memory reserved for the first copy of the object, and some pointers are updated to refer to the object's second copy.

A number of approaches might be employed to resolve these sorts of race conditions. The memory cycle accountings presented in this paper are based on the following techniques:

CREG

We assume that the arbiter has a special cache register named CREG. This register is loaded with a memory value and a memory address. All memory stores and fetches that refer to the address held in the CREG's address field access CREG's data value rather than memory.

Rollback

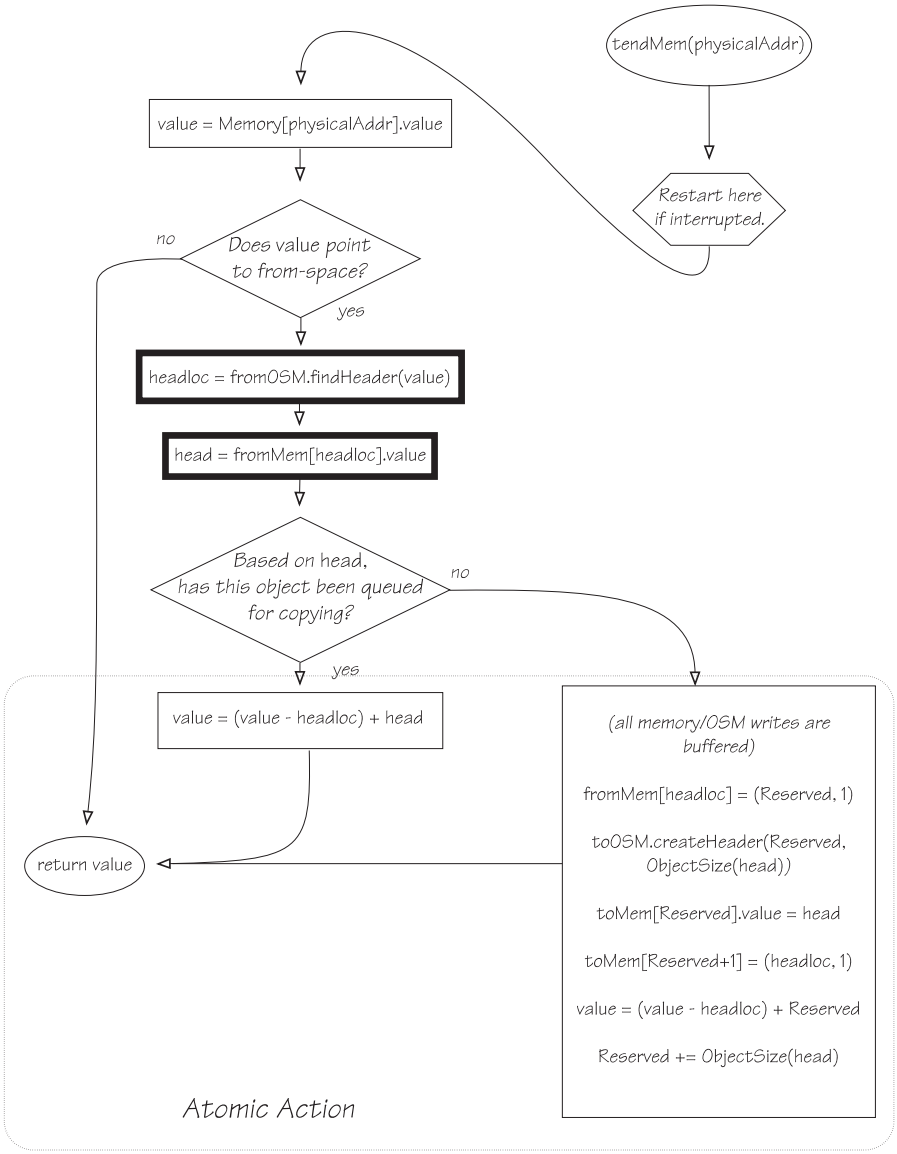
Since the highest priority of the arbiter is to service demand operations of the mutator, our goal is to minimize the time required to interrupt background garbage collection activities. To reduce the complexity of the required circuitry and to simplify our performance analysis, we have assumed that critical sections of garbage collection code simply rollback to a safe restart point whenever they are resumed after being interrupted.

The memory-cycle accountings reported throughout the remainder of this section assume that the routines execute without interruption. If certain routines are interrupted, they will rollback to a safe restart point and require additional memory cycles to complete execution. In cases where the frequency of interrupting background activities is so high as to negatively impact system throughput, the mutator's forward progress is automatically slowed appropriately by requiring longer delays on allocation requests, as controlled by the ScanBalance state variable described in reference [1].

Detailed accountings of each of the services provided by the arbiter to the garbage-collecting microprocessor follow.

tendMem (for internal use)

tendMem tends the descriptor held in the memory location named by its physicalAddr argument. Since tendMem assumes that the word of memory to be tended is held in the CREG register, the word can be fetched without accessing the memory system.

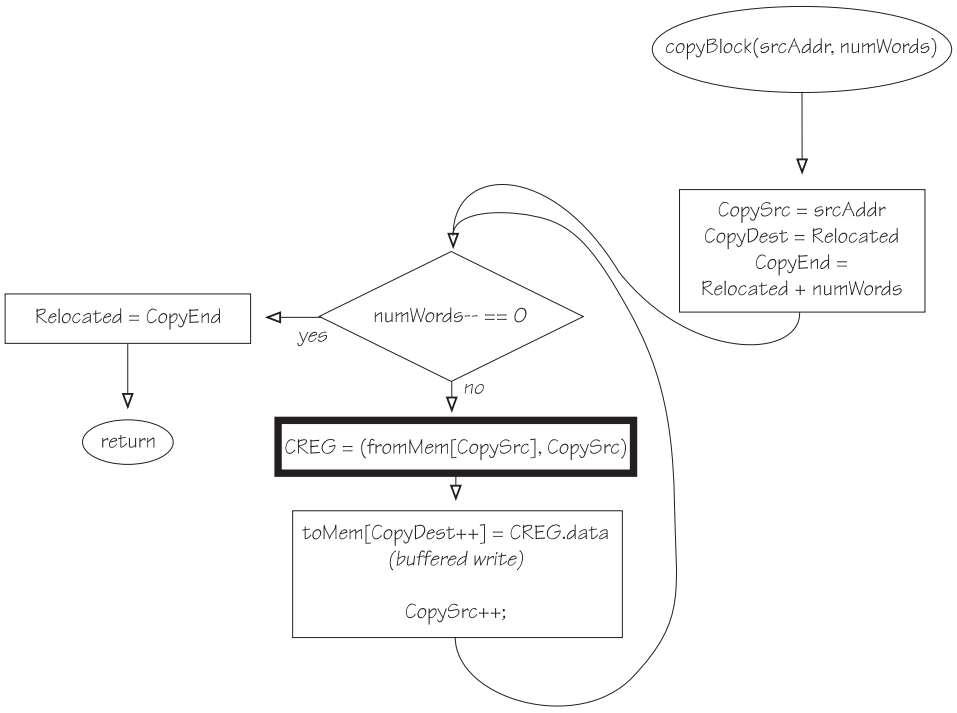


Note that the large box containing multiple buffered writes to memory and the OSM stores two values to *to-space*, one value to *from-space*, and issues a single update request to the *to-space* OSM module. The path to this large box includes one read from the *from-space* OSM and one read out of *from-space* memory. While these reads are being executed, other memory and OSM modules are able to process buffered write requests. Each module is guaranteed sufficient idle cycles to prepare its write buffers to hold all of the new requests to be buffered when the large box at the end of the tendMem operation eventually executes.

The worst-case time required to execute a tendMem instruction is two memory cycles.

copyBlock

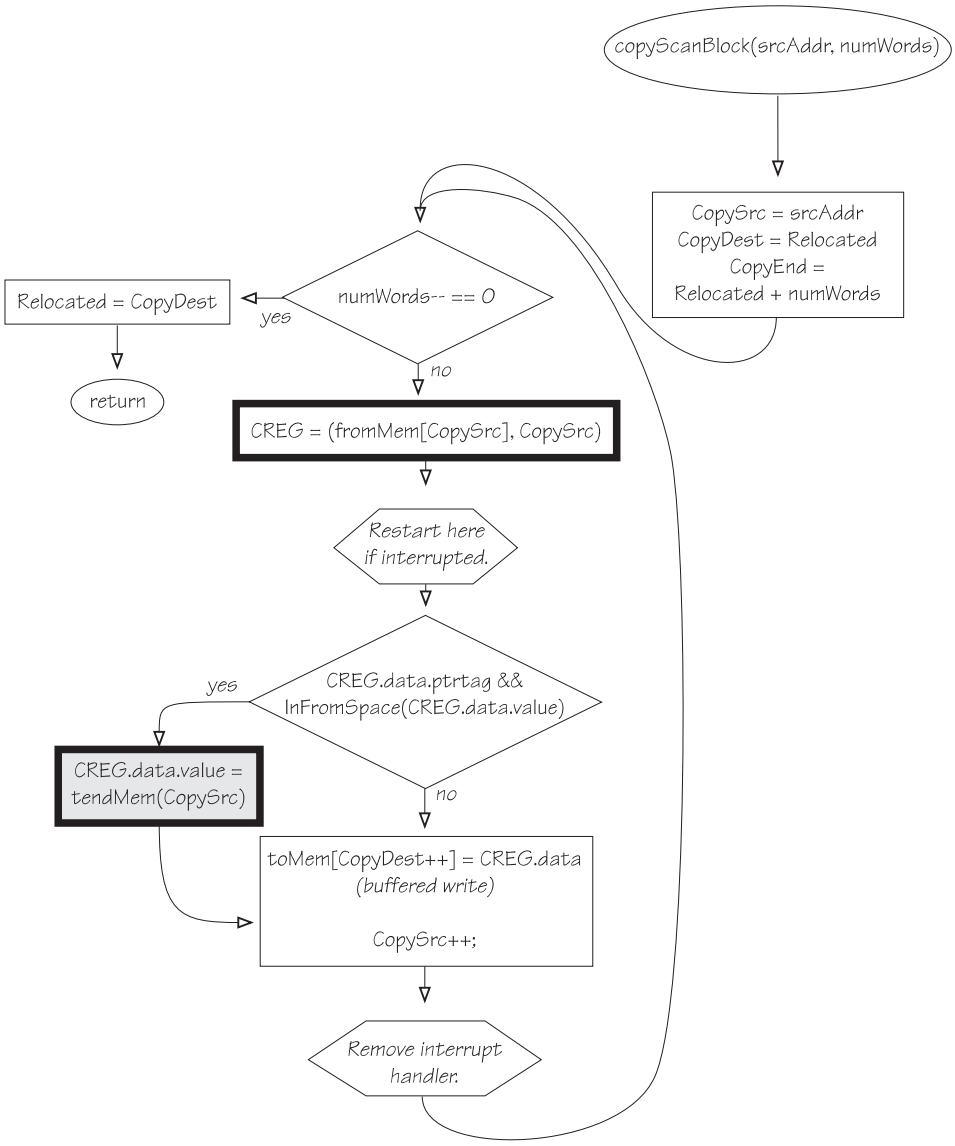
Assume that srcAddr points to at least numWords of contiguous data contained within a single *from-space* object. Incrementally copy this data to Relocated, maintaining the contents of the CopySrc, CopyDest, and CopyEnd registers during copying.



The worst-case time required to execute a copyBlock instruction is numWords memory cycles.

copyScanBlock

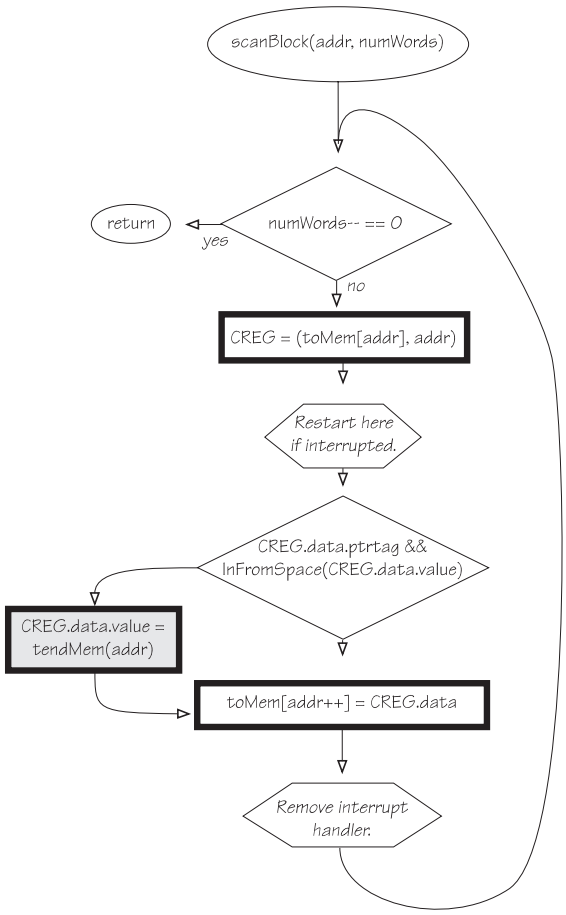
Assume that srcAddr points to at least numWords of contiguous data contained within a single *from-space* object. Incrementally scan this data while copying it to Relocated, maintaining the contents of the CopySrc, CopyDest, and CopyEnd registers during copying.



The worst-case time required to execute copyScanBlock is three times numWords memory cycles.

scanBlock

Assume that addr points to at least numWords of contiguous data contained within a single *to-space* object. Incrementally scan this data.



The worst-case time required to execute scanBlock is four times numWords memory cycles.

4. Additional Arbiter Support for the Microprocessor

This section describes arbiter operations that do not require coordination with the service routines provided to the mutator. Because these routines are much simpler than the others, their flow charts are not illustrated.

readWord

Read a single word of memory, which may reside in either *to-space* or *from-space*. This operation requires one memory cycle to execute.

writeWord

Write a single value to either *to-space* or *from-space*. This operation requires one memory cycle to execute.

incScanBalance

Increment the ScanBalance register. This operation does not access memory.

incCopied

Increment the Relocated register. This operation does not access memory.

getReserved

Return the value of the Reserved register. This operation does not access memory.

setDescriptorTag

Set the descriptor tag of a single memory location, which may reside in either *to-space* or *from-space*. This operation requires one memory cycle to execute.

getDescriptorTag

Fetch a single descriptor tag from either *to-space* or *from-space* memory. This operation requires one memory cycle to execute.

findHeader

Lookup the header location corresponding to a particular *to-space* or *from-space* memory address. This operation requires one memory cycle to execute.

zapFromSpace

Zero out all data and descriptor tags in *from-space* memory and reset the corresponding OSM circuits. Execution of this primitive signals completion of garbage collection. After clearing *from-space*, this primitive suspends operation of the arbiter's microprocessor interface until the arbiter's CPU interface receives a TendingDone invocation, at which time the microprocessor is prompted to begin work on the next garbage-collection pass.

N memory cycles are required to implement this instruction, where N is the total number of words in *from-space*.

References

1. K. Nilsen and W. J. Schmidt, Hardware-Assisted General-Purpose Garbage Collection for Hard Real-Time Systems, Iowa State Univ. Tech. Rep. 92-15, 1992.
2. K. Nilsen and W. J. Schmidt, Cost-Effective Object-Space Management for Hardware-Assisted Real-Time Garbage Collection, *ACM Letters on Prog. Lang. and Systems*, submitted.
3. K. D. Nilsen and W. J. Schmidt, Preferred Embodiment of a Hardware-Assisted Garbage-Collection System, Iowa State Univ. Tech. Rep. 92-17, 1992.

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Tech Report: TR 91-21c
Submission Date: May 5, 1992