

Modeling Sparsely Utilized Memories in VHDL

Scott Bilik

Sanders, A Lockheed-Martin Company

sbilik@sanders.com

Abstract

This paper focuses on dynamic allocation techniques used when large memory addresses spaces must be modeled. The address space is often too large and sparsely utilized to be statically allocated as a simple array. Using dynamic allocation and some simple optimizations, one can model large address spaces while minimizing the time overhead of dynamic allocation and space overhead of static allocation. The optimizations covered will include hashing, linked lists, spatial locality clustering, and temporal sorting. The techniques will be demonstrated inside of a VHDL package that has been reused for many types of memory models on the Sanders RASSP program.

1.0 Background

The need to model a memory is common in VHDL simulation. There are two main approaches to modeling a memory: static allocation and dynamic allocation. For small memory sizes static allocation has several benefits. First it is trivial to understand and implement. One simply uses an array type to represent the memory. Memory accesses are as simple as array accesses. Reading and writing memory is computationally simple.

However, the number of memory reads and writes during a simulation run is often much smaller than the memory space being modeled. If the memory space being modeled is sufficiently large, it may be impractical to allocate an array to cover the address space. For instance, a typical RTL simulation will run less than 1 million clock cycles. The address space, though, may be 32 bits wide. It would be more efficient if the simulation

only required a few megabytes to record the transactions rather than gigabytes.

The desire to conserve memory is met by using dynamic allocation. Memory on the host computer is allocated on demand as the model simulates new memory transactions. For most RTL simulations only a few megabytes of computer memory is needed to hold the transactions for long simulation runs.

The big problem with dynamic allocation techniques is how to organize this memory so that the location of desired address can be retrieved quickly. In the static case an address was simply mapped to an array index. For dynamic allocation one would like to find a way to organize the memory storage so that the mapping of simulation address to host computer address is as quick as indexing an array.

2.0 Approach

The approach taken in this paper is similar to that used in hardware caches. It is summarized in Figure 1. One builds a hash table. Elements in the hash table are pointers to linked lists. Nodes of the linked lists are records of address and data for a particular location. To look up a simulation memory reference, one first hashes the address to a location in the hash table. One then searches the linked list pointed to by that hash table entry for the desired simulation address by following the links until a match is found. If there is no match found, a new node is added to the linked list.

The linked list is required because multiple addresses will hash to the same location in the hash table. Depending on the simulation memory access pattern, the user

will typically need to make a tradeoff between the size of the hash table and the length of the linked lists. If m is defined as the number of unique memory references in a simulation run, h as the number of entries in the hash table, and l is the average length of the linked lists, then we can say that $m = hl$. A larger hash table will result in shorter linked lists resulting in shorter searches of the linked list. If a simulation run is only going to access a few thousand unique locations, it is sufficient to set the hash table to 1024 entries. In this case, the average linked list will have a handful of entries and traversing the list will be quick.

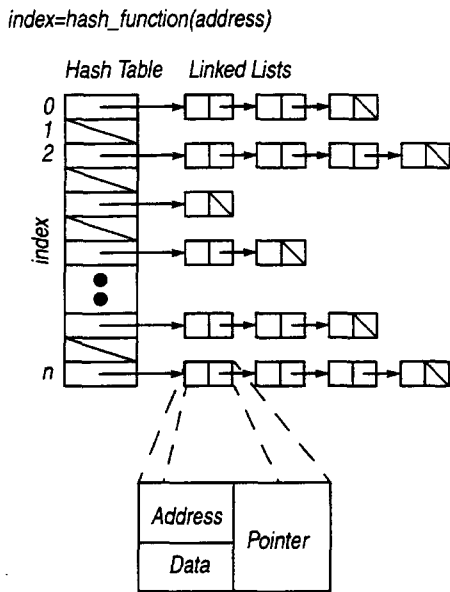


FIGURE 1. Data Structure

The hash table is an overhead to this approach since its sole function is to help organize and partition the data. One would like to keep the size of the hash table small if possible. If millions of unique address references are required for a particular simulation, one would like to avoid having to allocate a hash table with millions of entries. Unfortunately smaller hash tables lead to larger linked lists. This increases the average search time. There are two further optimizations one can use to alleviate this problem.

3.0 Optimizations

First one can take advantage of spatial locality. Address references are typically clustered in regions of memory. One can utilize this characteristic by having each node

in the linked list hold a line or block of consecutive memory locations, as shown in Figure 2. This is similar to the behavior of processor primary caches. The size of the line depends on how clustered the address space utilization is. Typical values could range between 4 and 256 words in powers of 2.

The search algorithm would need to be modified slightly. Instead of searching for a particular address, the algorithm searches for the line that address would be within. Upon finding or creating that line, the algorithm would then index within that line using standard array referencing. The index is formed from the least significant bits of the address.

This approach reduces the size of the hash table and the average length of the linked lists. However each node in the linked list is larger to accommodate the line of data it now stores. In a memory access pattern with a lot of spatial locality, the overall effect is typically a reduction of workstation memory used. This is because there are fewer overall nodes and thus less of the address and link overhead. A larger percentage of the computer memory is holding data. Where the access pattern doesn't have a lot of locality, memory could be wasted as a whole line of data is allocated when only a single word of data might be required. The user needs to estimate a line size that will not cause excessive wasted memory.

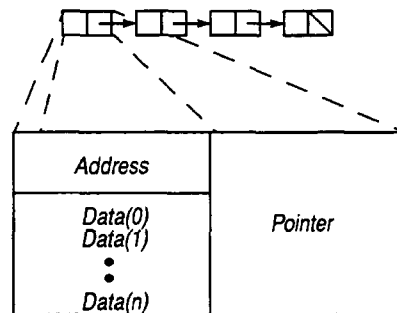


FIGURE 2. Spatial Clustering

The second optimization takes advantage of temporal locality. This is the assumption that the current address access is likely to be accessed again in the near future. Conversely, locations that have not been recently accessed have a lower probability of being accessed again in the near future. This is the fundamental principle of hardware memory caches. To take advantage of this property one should keep the nodes in the linked lists sorted from most recently used to least recently used.

Luckily, sorting the nodes is a low overhead operation. On new memory accesses the created node should be placed at the head of the linked list. On accesses to previously referenced locations, the node should be moved to the head of the linked list, as shown in Figure 3. This operation requires no copying but merely changing a few pointers. The pointer changes must be done with care, though, or nodes can be accidentally dropped from the linked list.

Sorting the nodes does nothing to shorten the length of the linked lists. It merely attempts to reduce the time required to search the linked list for an address match by increasing the probability that a match will be found early in the list. It should be noted that whether the nodes are sorted or not, a reference to a new location will require the traversal of the entire linked list. Accesses to new locations will be the worst case condition, especially later in the simulation when the linked lists are longer.

As in the case of the spatial optimization, there are cases where sorting provides no benefit. In these cases, the time taken to maintain the order of the nodes is wasted. One would have a faster simulation if the nodes had been left in their original position. This was seen on the Sanders RASSP program when the memory models were attached to a RAM BIST model. The same address space was traversed four separate times in same access pattern. In cases like these it is best to rely on the use of larger hash tables and larger block sizes to reduce the search time.

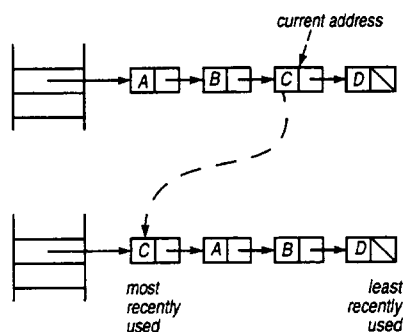


FIGURE 3. Temporal Sorting

4.0 Implementation

The concepts described above have been implemented in a memory support package. From the users perspective, there are three main functions and procedures used to access memory. There are also a few variables that

can be modified to adapt the package to particular simulation needs.

4.1 User Perspective

First, there is the `allocate_memory` function. Typically one need only call this once, but it can be called several times if the user desires to create and manage several separate address spaces. This function creates a hash table and initializes the entries. The user is returned a `memory_handle` that is required for the other two procedures.

The other two procedures are `read_memory` and `write_memory`. Both require the user to supply the memory handle, an address, and a data value. Obviously in the case of `write_memory`, the data value is an input to be stored. For `read_memory` the data value is an output to be retrieved. Both address and data are passed as standard logic vectors.

As it currently stands, the memory package is not intended to be centrally compiled but locally compiled for each project. This is because at the head of the package there are parameters that the users can tune to optimize the performance of the algorithm. The first of these is `hash_order`. The size of the hash table is set to $2^{\text{hash_order}}$. The second is `block_order`. The size of the memory data blocks at each node in the linked list is $2^{\text{block_order}}$.

There are also a handful of booleans that the user can modify to modify the operation of the algorithms and, indirectly, its performance. The first is `complain_about_ambiguity`. When set true, procedures that access memory will output assertion warnings if a 'U', 'X', 'W', 'Z' or '-' is present on either the address or data inputs. Regardless of its status, these bits will be forced to zeroes during memory accesses. The second is `complain_about_new_read`. When set to true, a read from a location not previously accessed will cause an assertion warning. Regardless of how it is set, the read will result in all zeroes on the data.

4.2 VHDL Implementation Details

Although the user interface to the read and write procedures deals with standard logic vectors, to save more space the address block tag and data are stored as integers. Although this is a saving over storing standard logic vectors, it limits the data width per memory location to 32 bits. Trivial modifications could store 64 bit words if necessary. At the other extreme if the simulated

memory is intended to be byte addressable, an integer uses four times the necessary space. The user could modify the procedures to be tailored to 8 bit or 16 bit memory accesses if space becomes a problem.

This model depends upon two primitive types called `cell` and `link`. `link` is merely a pointer to a cell. The cell contains an address block tag (stored as an integer), a data block (stored as an array of integers), and a link to the next cell. Linked lists are formed from the connection of these cells. The hash table is an array of links. Each link element of the hash table points to the heads of the individual linked lists or NULL if there is no list associated with that element.

Address decoding proceeds as shown in Figure 4. The lowest `block_order` bits are converted to an integer that will index into the data block array within a cell. The next `hash_order` bits are converted to an integer to form an index into the hash table. This is a rather simple hash function, but yields reasonable results. The remaining most significant bits are converted into an integer to form an address tag.

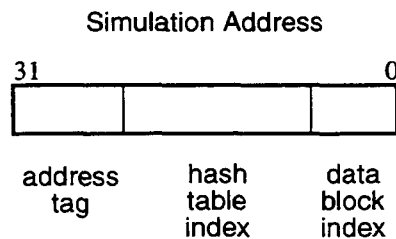


FIGURE 4. Address Decode

The search begins by indexing into the hash table to find the head of the appropriate linked list. The cells in the linked list are searched sequentially for an address tag that matches the tag formed from the most significant bits of the desired address. If none is found in this linked list, a new cell is created with that tag. The index formed from the address least significant bits is used to index into the data array within that cell. At this point, the data is retrieved or stored as appropriate. Lastly some minor manipulations of the links are performed to move this cell to the head of its linked list.

5.0 Conclusion

This paper has presented techniques to organize the dynamic allocation of memory so that the memory consumed and the memory search time is minimized. It used hashing, clustering, and sorting to reduce search

times. Dynamic allocation, clustering and the generous use of integers in lieu of standard logic vectors reduced the memory consumed. When large but sparsely used memory spaces must be modeled, these techniques can minimize the memory resources and paging activity in many virtual memory operating systems common today. This can lead to improved simulation times or even the ability to model an otherwise impossible address space.

6.0 Acknowledgements

The work presented in this paper was sponsored by the RASSP Program. The Sanders RASSP team is under contract to Naval Research Laboratory, 4555 Overlook Avenue, SW, Washington DC 20375-5326. The Sponsoring Agency is: Defense Advanced Research Projects Agency, Electronics System Technology Office, 3701 North Fairfax Drive, Arlington, VA 22203-1714. The Sanders RASSP team consists of Sanders, Inc., Motorola, Hughes, and ISX.