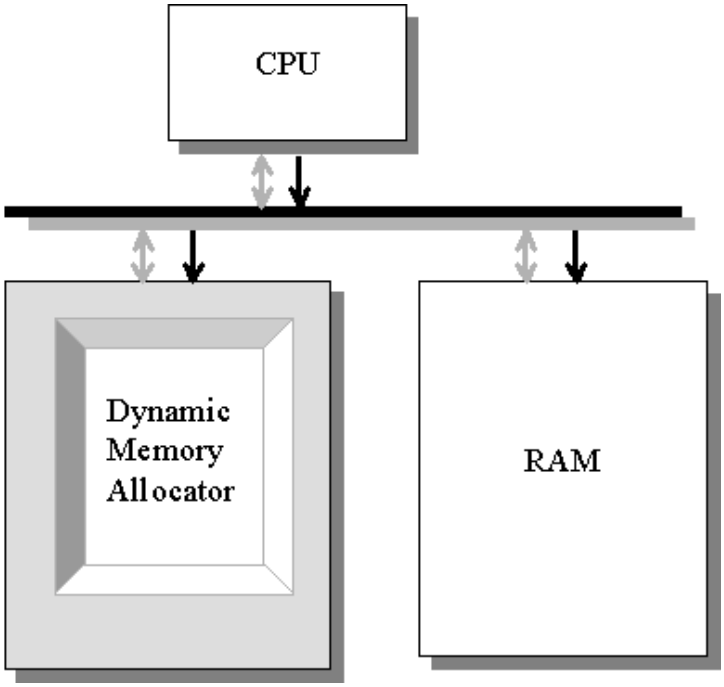
Dynamic Memory Management in Hardware

A master thesis presented to the Mälardalens University in fulfilment of the thesis requirement for the degree of Master of Science in Electrical Engineering

Västerås, Sweden, 2002



Authors :

Karl Ingström
E-mail:kim99001@student.mdh.se
Phone: (+46)070-5586023

Anders Daleby
E-mail:ady99002@student.mdh.se
Phone: (+46)021-129884

Abstract

This project has been performed at the Department of Electronics ,Mälardalens University, Västerås, Sweden. The project was initiated and conducted in co-operation with RealFast AB, www.realfast.se and The Department for Computer Science and Engineering Mälardalens University.

The project is divided into two parts. The first part is a state of the art report of memory management. In the second part a hardware dynamic memory allocator (DMA) is developed.

Part I
Part I gives a background to memory management. It presents the memory management techniques commonly in use today, regarding process management and dynamic memory allocation techniques. A closer look is taken on how memory management is handled in some real time operating systems including eCos, OSE, VRTX and VxWork. To assist memory handling, a hardware memory management unit (MMU) is often implemented in present processors architectures. Two such units, the powerPC and the MIPS MMU, has been studied. Two existing hardware allocator at research level are also presented. According to those it is possible to perform dynamic memory allocation in hardware. Thereby increasing allocation speed and determinism and making dynamic memory allocation more appropriate for embedded real-time systems or system-on-chip (SoC). This paper will serve as the foundation for future implementation of a hardware dynamic memory allocator.

Part II
In part II we present a novel approach for performing memory allocation and deallocation in hardware, a dynamic memory allocator, DMA. The DMA uses a pre-search approach (before allocation request) for finding free objects. The pre-search will reduce the response time. A predictable minimum timing restriction between two service requests to the DMA is introduced, which guarantees a pre search time for the DMA. The dynamic memory is represented by a bitmap.

The DMA has been implemented in hardware using VHDL. The implementation is generic and can be mapped to different environments.

The DMA is shown to reduce allocation time, compared to a software allocator, in a test application. The allocation response time is fast, less the 15 clock cycles. Resulting in significant speed up for the application, up to 90 % compared to the software allocator. The DMA has predictable behaviour, allocation always take the same amount of time, making it useful in real time systems.

Table of contents

State of the art

Memory management

Abstract

This report gives a background to memory management. It presents the memory management techniques commonly in use today, regarding process management and dynamic memory allocation techniques. A closer look is taken on how memory management is handled in some real time operating systems including eCos, OSE, VRTX and VxWork. To assist memory handling, a hardware memory management unit (MMU) is often implemented in present processors architectures. Two such units, the powerPC and the MIPS MMU, has been studied. Two existing hardware allocator at research level are also presented. According to those it is possible to perform dynamic memory allocation in hardware. Thereby increasing allocation speed and determinism and making dynamic memory allocation more appropriate for embedded real-time systems or system-on-chip (SoC). This paper will serve as the foundation for future implementation of a hardware dynamic memory allocator.

Authors :

Karl Ingström
E-mail:kim99001@student.mdh.se
Phone: (+46)070-5586023

Anders Daleby
E-mail:ady99002@student.mdh.se
Phone: (+46)021-129884

Karl Ingström
kim99001@student.mdh.se

Anders Daleby
ady99002@student.mdh.se

Table of contents

Karl Ingström
kim99001@student.mdh.se

Anders Daleby
ady99002@student.mdh.se

1 Introduction

This *State Of The Art* report gives an introduction to some topics of *memory management*. The need for memory management arises with the usage of multiple processes in a system. Multiple processing is used to boast system utilisation i.e. a process could spend much time in a blocked state waiting for some I/O to be completed, it would then be efficient to allow another process to run during the block time.

When several processes are to execute at the same time they must share the memory between them. The task of dividing the memory is carried out dynamically by the operating system and is called **memory management**. The job of the memory manager is to keep track of which parts of the memory that are in use and which parts are not, to allocate memory to processes when they need it and to deallocate it when they are done. It also manages swapping between main memory and secondary memory if a multiple level memory hierarchy is used, and the main memory is not big enough to hold all the processes that are executing. The use of swapping is not the common case in the embedded world which this survey is partly aimed at and will not be discussed in great detail.

In chapter two the basic requirements a memory manager is intended to satisfy is introduced to the reader. Chapter three provides an overview of the theory for memory management. The parts that are covered in the chapter is, different techniques used for allocation of memory, how to divide the memory into partitions, virtual memory is introduced, the loading and linking relationship to memory management is also presented. In chapter four an investigation concerning the memory management in four existing real-time operating sytems, eCos, OSE, VRTXsa and VxWorks are made. Two different processors, powerPC and MIPS, are examined in chapter five, the main focus is on the hardware memory management unit (MMU) in the architecture. There are research going on in the topic about memory allocation performed in hardware, two existing hardware allocator are examined in chapter six. Software allocation is a time consuming and usually unpredictable task. Development of larger chips makes it possible to implement more and more software functions in hardware there by increasing execution speed.

Karl Ingström
kim99001@student.mdh.se

Anders Daleby
ady99002@student.mdh.se

2 Memory manager requirements

There are some basic requirements a memory handler should support :

- Transparency
- Protection
- Relocation
- Sharing
- Efficiency

Transparency
The memory manager should be invisible to the user, i.e. the programmer. He/She shouldn't need to be aware of the existents of the memory handler. A process running on the system should not be aware of the sharing of memory between processes.

Protection
A process should be protected against unwanted interference of any other process. Memory belonging to a process should not be accessible from other processes without permission.
Figure 1 illustrates the address requirements for a process, the memory part of a process image doesn't need to be contiguous as in figure. Due to the relocation property (se below) the location of the process in main memory will not need to be determined pre-run time. This makes it impossible to check absolute addresses for protection errors during compilation time, implying that protection checking must take place during run time. Also mechanisms for declaring region of memory as read only, execute only or read write should exist. The protection mechanism adopted must probably be assisted by hardware otherwise it will become a very time consuming job for the operating system to check all memory references during run time.

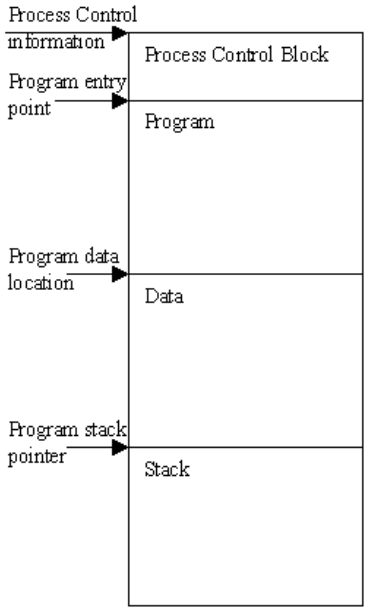


Figure 1 A process image showing the address requirements for a process

Sharing
There will be need for sharing memory between processes. If a number of processes is to execute the same code it would be better to use a single copy of that code instead of letting each process having its own copy. For example in Figure 2, two processes are running the same program (program A). In a system with a memory handler that doesn't support any sharing mechanism there will exist two copies of A, whereas in a system with sharing only one copy of A is required there by reducing memory usage.

Also processes that are co-operating on solving a job will have the need of exchanging data between each other. Therefor the memory management system must provide some form of controlled access to shared memory space without relinquish the protection demands.

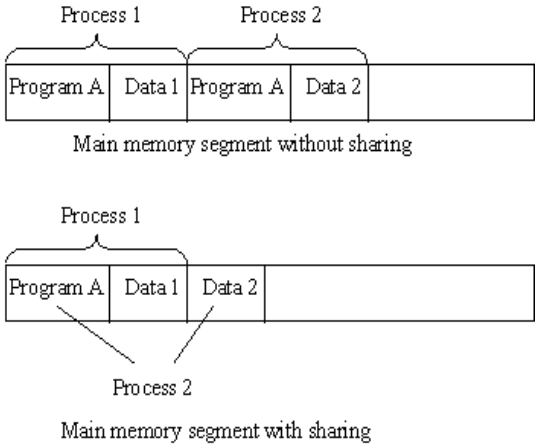


Figure 2 Illustration of sharing requirement for multiprocessing system

Relocation
Due to the sharing capabilities of a multiprogramming operating system the programmer can't tell where and who much memory that could be used by the program. The programmer doesn't know if there are any other programs taking memory space. It must therefor be the memory managers job to place program into main memory. When loading programs the memory manager must determine where the different parts of the program shall be located in main memory. The information of the location of the program must become available to the operating system. In
Figure 1 it is shown what information the operating system must handle for each process. The memory manager should get free hands regarding where to locate memory for a certain process. It would be quite limiting if a process needed to be loaded into the same memory region every time, especially in the case swapping is used. A process could in such case be blocked even though there are other suitable memory regions where it could be placed. The memory references inside a program must be solved during load time or run time (see loading /linking) in a relocatable environment, thus the absolute addresses of a program can't be known at compile time.

Efficiency
The memory management should not incorporate sufficient overhead for the operating system.

Embedded system considerations
However in embedded applications some of these requirements is more or less unnecessary since the common case seems to be that there is just one program running (may consists of several threads). Making the needs set by multiprogramming obsolete because static loading and linking can be applied. It could however be useful to use more sophisticated memory management techniques during development to catch possible memory access bugs.

3 Memory management theory

This chapter will introduce some well known memory management theory. Different allocation techniques and memory partitioning schemes is presented. Also the concept of virtual memory is introduced. The topics covered can be found in most operating system literature. The information here have been compiled from *Operating systems: Internals and design principle* by William Stallings [8], O*perating system concepts* by A. Silberschatz and P.B Galvin [9] and *Modern operating systems* by Andrew S. Tanenbaum [20]. The part about allocation techniques has manly been gathered from *Dynamic storage allocation: A survey and critical review* by Paul Wilsson, Mark Johnstone, Michel Neely and David Boles [6].

3.1 Memory allocation techniques

Several general allocation exists and Paul Wilsson, Mark Johnstone, Michel Neely and David Boles gives a good overview in *Dynamic storage allocation: A survey and critical review* [6]. The techniques are constantly recurring in most aspects of memory allocation, whether it is about variable allocation from a heap, space allocation when loading processes into main memory or any other reason for allocating memory space. [6] divides the common allocation techniques into these mechanisms :

- Sequential fits
- Segregated free lists
- Buddy system
- Bitmapped fits

In this section its assumed that the memory to be allocated is taken from some memory set which here will be called a heap.

3.1.1 Sequential fits

A sequential fit allocation algorithm use a linear list of free memory blocks representing free space to allocate from. The list consists of blocks in the heap that have not been allocated. At the beginning the list will only consists of one block with a size equal to the whole heap size. The list itself is usually encoded inside the free blocks causing no significant memory overhead for the data structure. However this cause a restriction on how small blocks that can be allocated. Each block will need to be large enough to contain the information used by the linked list. An overhead is added to every block in form of a header field. This header field is usually used to store the size of the block, to make freeing of blocks easier. Most standard allocator interfaces for freeing objects doesn't require the size of the object to be passed to the deallocation algorithm, i.e. the ANSI-c standard interface for freeing objects, **free()** [22]. The list used is often a double linked and/or circular linked list. To make coalescing fast and simple a sequential fit algorithm usually utilise Knuth's [21] boundary tag technique. The search of the free list for available memory could be done in several different ways. The classical approaches are first-fit, best-fit and next-fit. The biggest problem with sequential fits is that the implementation might not scale well to large memories. When the number of free blocks grows the time to search the list might become unacceptable.

More sophisticated implementations of sequential fits using other data structures then linked list exists [6], i.e. trees. Which improves the algorithms in term of allocation speed and scalability to large heaps.

First fit
A first fit algorithm simply search the list from the beginning and allocates the first block of memory that is big enough to serve the request. If a larger block then necessary is found, the block is divided and the remainder is inserted to the free list. This introduce a little problem to first fit. As larger block at the beginning of the list is used first, the splinted part of these blocks causes fragmentation into small block (splinters) in the beginning of the list. As the list is traversed from the beginning, all these blocks need to be traversed in every search for a larger block and thereby increasing search time. Another issue is in which way to insert deallocated blocks into the free list. A freed block could be inserted first (LIFO), last (FIFO) or in address

order into the free list. An advantage of address ordered first fit is that coalescing can be made fast without any use of boundary tags.

Bets fit
The allocator search through the complete free list to find the best possible fit, the search could however stop if a perfect fit is found before the complete list has been traversed. The goal of this strategy is to minimise the amount of wasted space by ensuring that fragments are as small as possible to serve the request. According to Stalling [8] the result of best fit might be that the reminder when splitting blocks will be quite small. These small reminders will be so small that they will hardy be able to serve any request for memory. Therefor memory compactation must be done more frequently with best fit opposed to first-fit. However this has not been shown to be any real problem for real workloads [22]. As in the case of first fit this technique doesn't scale well to large memory heaps with a lot of free blocks. The main downside is however the exhaustive search that is needed.

Next fit
As a variation of first-fit there is next-fit. With next-fit the free memory list is searched form where the last allocated block where found, instead of always from the beginning as in first-fit. The purpose of this allocator technique is to avoid the accumulation of splinters which will occur if the search always start at the same location. Since the allocation of objects start at different locations, allocations for objects in a programming phase may be scattered in memory which might affect fragmentation. Next fit has been shown to cause more fragmentation than best fit or address ordered first fit [22][6].

3.1.2 Segregated free Lists

A simple way to handle allocation is to use an array of free list, where each list contains free blocks of a certain size range. Each array position corresponds to a size class encoding a size range. When freeing a block it is simply put into the list containing the particular size class. When allocating a block the free list for the appropriate size class is used to satisfy the request. This is illustrated in Figure 3.

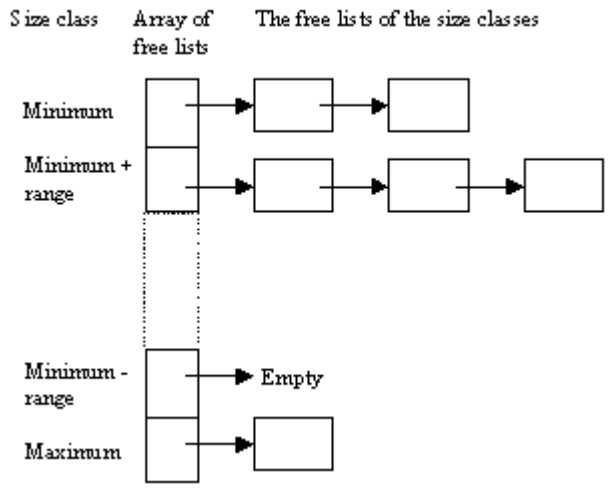


Figure 3 Overview of segregated free lists

Simple segregated storage
In this variation of segregated free lists no splitting or coalescing of free blocks is done. The request for a block of a size is rounded up to the closest size class. If a request for a block in a size class which has an empty free list is made, the allocator tries to get more memory from the underlying system by trying to extend the heap size. Typical a number of pages is requested (if the underlying system is using paging) and divided into blocks put on the free list for the size class in question. An advantage of simple segregation is that the overhead for each block can be made small because no header is required for a single block, the size information can be

recorded once for all blocks in a page (assumed paging is used). Allocation speed is an advantage for simple segregated storage since no searching is necessary, assuming there exists free blocks in the free list, allocation time can be made constant. Also freeing can be made in constant time. The disadvantage for this scheme is that it could introduce severe fragmentation, both internal and external. No attempts is made to split or coalesce block to satisfy requests for other size classes. The worst case is a program that allocates many objects of one size class, deallocate the objects and then does the same for many other size classes. In this case separate storage is required for the maximum number of blocks in each size class, because none of the memory allocated in one size class can be used by another size class.

Segregated fits
As with simple segregated storage this technique uses an array of free list where each free list is corresponding to a size class. The blocks in a free list of a size class need not to be of the same size, so the free list must be searched for an appropriate block to serve an allocation request. The search is usually performed using a sequential fit technique. When a free list is empty segregated fit algorithms try to found large blocks in a class above the searched size class to serve the request. If a larger block is found it is split and the remainder is put into an appropriate free list. When a block of a given size is freed it might be coalesced with other block and be put on a different free list. Segregated fits can be further divided into the following categories:

Exact lists
With an exact lists system each possible block size could have it's own free list. As a result there might exist a large number of free lists. However the array of free lists can be represented sparsely. Stadish and Tadman's Fast Fit algorithm [24, 25], make use of an exact list array for small classes and uses a binary tree of exact lists for larger sizes, but only the sizes that are present.

Strict size classes with rounding
This is a simple segregated storage scheme which allow splitting and coalescing. The class series must be carefully designed so that when splitting occurs the splinted blocks is also in the series. Otherwise the result might be blocks that are of sizes with no corresponding free list.

Size classes with range lists
The most common technique is to allow blocks of slightly different size to be hosted in the same size class. The search of the free list for a size class is the performed by any of the sequential allocation techniques described above.

3.1.3 Buddy system

Buddy systems are a subclass of strict size classes with rounding segregated fit, in which splitting and coalescing are made fast by pairing each block with a unique adjacent *buddy* block. The buddy system uses a array of free lists, one for each available size class. When allocation is to be performed the requested size is rounded up to the nearest allowed size and allocation is made from the corresponding free list. If the free list is empty, a larger block is selected and split into two buddies. Buddy sizes are usually powers of two, or from a Fibonacci sequence, such that any block in the in the sequence can be divided into two block of a smaller size that is also in the sequence. A block may only be split into a pair of buddies.

A block may only be coalesced with its buddy, and this is only possible if the buddy has not been split into smaller blocks or has been allocated. The block resulting from coalescing will be a block at the next higher level of the hierarchy.

The advantages of buddy systems is that the buddy of a block being freed can be quickly found by a simple address computation. Furthermore only minimal overhead per block (1 bit) is necessary to indicate whether the buddy is free or not. Also fast and predictable coalescing can make buddy systems an option for real-time systems [22]. The disadvantage of buddy systems is that the restricted set of block sizes leads to high internal fragmentation, as does the limited ability to coalesce.

Different sorts of buddy system are distinguished by the available block sizes and the method of splitting. They include binary buddies (the most common), Fibonacci buddies, weighted buddies, and double buddies [6].

For example, an allocator in a binary buddy system might have sizes of 16, 32, 64,... 64 kB. It might start off with a single block of 64 kB, Figure 4. If the application requests a block of 8 kB, the allocator would check its 8 kB free list and find no free blocks of that size. It would then split the 64 kB block into two block of 32 kB, split one of them into two blocks of 16 kB, and split one of them into two blocks of 8 kB. The allocator would then return one of the 8 kB blocks to the application and keep the remaining three blocks of 8 kB, 16 kB, and 32 kB on the appropriate free lists, Figure 5. If the application then requested a block of 10 kB, the allocator would round this request up to 16 kB, and return the 16 kB block from its free list, wasting 6 kB in the process, Figure 6. A Fibonacci buddy system might use block sizes 16, 32, 48, 80, 128, 208,... bytes, such that each size is the sum of the two preceding sizes. When splitting a block from one free list, the two parts get added to the two preceding free lists.

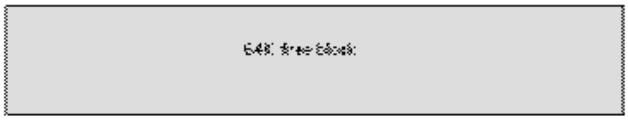


Figure 4 A binary buddy heap before allocation

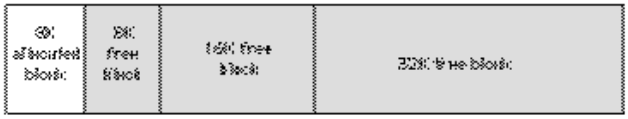


Figure 5 A binary buddy heap after allocating a 8 kB block

Figure 6 A binary buddy heap after allocating a 10 kB block, note the 6 kB wasted because of rounding up

3.1.4 Bitmapped fits

With bitmapped fits a bitmap is used to record which words of the heap that are free for allocation and which are not. The bitmap is a vector of one bit flags where every word on the heap have a corresponding flag. Allocation is done by searching the bitmap for a run of clear bits suiting the requested allocation size. Another bitmap could be used to encode allocated block boundaries, i.e. the size of allocated blocks. If object sizes are small, bitmapped allocation could have space advantages over systems that uses whole word headers, due to less overhead. The main disadvantage is that search times are linear, however there could be ways of working around this [6].

3.1.5 Coalescing

Eventually when using dynamic memory allocation the memory could become external fragmented. That is there will be lots of small pieces of memory that are to small to be useful. Then it could happen that a memory allocation request could fail because there is not any large enough free block to serve the request although there is enough memory free in total. By merging adjacent free memory blocks into larger free blocks (coalescing) external fragmentation could be reduced. Coalescing can be done when blocks are set free or postponed to a later moment, called deferred coalescing. [6] has a discussion on deferred coalescing.

3.2 Memory management partitioning

In most system it can be assumed that the operating system occupies some fixed partition of main memory and the rest of the memory is to be divided among several (possible dynamically allocated) processes. This shared memory must be handled between processes some how. In the discussion below its assumed that a processes must be completely loaded into main memory for execution.

3.2.1 Fixed partitioning

The simplest memory partitioning technique is fixed partitioning. With fixed partitioning the main memory is divided into regions with fixed boundaries. Two variations exists depending on partition size :

- equal size
- unequal size

The partition size is determined during system boot.

Equal sized partitions
With equal sized partitions the memory available for user processes is divided in equal sized partitions with pre-determined size. A processes of less or equal size of the partition may be loaded into any available partition. This technique suffers from some obvious drawbacks. It is not possible to load a process of a size bigger then the partition size without using an overlay technique. Another drawback is that equal sized partitioning suffers from internal fragmentation. That is when a process of a size less then the partition size is loaded to a partition, it will not occupy all memory assigned to the partition, there will be internal fragmentation. For example assume a 240Kb process is to be loaded into a equalised partition in Figure 7. Then there will be a internal fragmentation of 272 Kb (512-240).

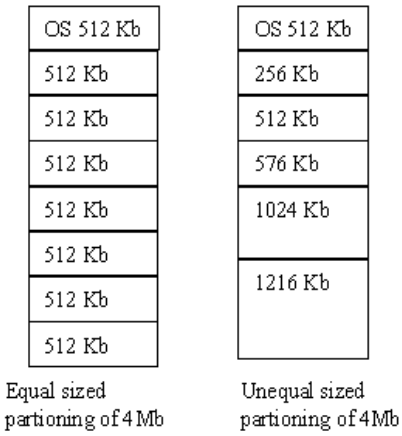


Figure 7 Fixed size partitioning

Unequal sized partitions
With unequal sized partitioning the drawbacks of equalised partitioning is slightly reduced but not totally eliminated. The internal fragmentation problem is reduced by using partitions that better fits the processes size. Some large partitions could be used to accommodate relative large processes. Processes can be placed in any partition of equal or larger size that is available. If the example with equal sized partition the partition of 240 Kb process should be placed in the unequal sized partitioning of Figure 7. The process could be placed in the 256 Kb partition only leading to a internal fragmentation of 16Kb (256 –240 Kb).

Placement algorithms
With equal sized partitioning the task of placing process in memory is easy. As long as there is any free partitions a process can be loaded into anyone of them, since all partition is equal in size. Processes bigger then the partition size can't be loaded and special techniques such as overlays must be used by the programmer. If there is no available partitions, swapping might be used if that possibility exists. For unequal sized partitioning there are two ways of assigning a process to a partition. The simplest is to assign each process to the partition in which it fits best. A queue must be handled because several processes will possible be assigned to the same partition. This approach minimises fragmentation but processes could be unnecessary blocked, several processes might want the same partition some will then be blocked although they could have been assigned to any available larger partition. Another approach is to load a process into the smallest partition large enough to hold the process. If there is no available partitions swapping might be used if that possibility exists.

Overlays
If a program is to big to be loaded into any partition the programmer must engage in a technique called overlay. With overlays the programmer must divide the program into several modules that doesn't need to be loaded into main memory at the same time. [20] has a section describing the use of overlays.

Summary
The simple implementation and the little cost of operating system overhead are the advantages of fixed sized partitioning. On the negative side there is the issue of internal fragmentation resulting in insufficient memory usage. Also with fixed partitioning the maximum number of processes that can execute concurrently is limited by the number of partition's. It could also be an issue that the OS could not grow larger then its partition without special programming interference.

3.2.2 Dynamic partitioning

To overcome the problems of internal fragmentation when using fixed partitioning, dynamic partitioning was developed. With dynamic partitioning the partitions used are of variable length and numbers. When a process is to be loaded into main memory a partition of a size exactly enough to fit the process is dynamically allocated. This means that there will be no main memory wasted in internal fragmentation. To allocate memory for a process can probably any of the allocation techniques described in chapter 3.1 be used. Note that some allocation techniques will introduce some internal fragmentation i.e. if using a buddy system to perform the dynamic allocation.

Compactation
Some of the allocation techniques introduce external fragmentation and a way to reduce the effect is to use compactation. With compactation the processes are shifted to be contiguous in memory so all of the free memory reside in one block. To be able to use compactation dynamic relocatable loading must be in use so that process addresses can be regenerated during compactation. This can make compactation a very time consuming operation.

3.2.3 Paging

With paging the main memory is partitioned into small equal fixed sized blocks called page frames. A process to be loaded is also divided into blocks of the same sizes, called pages. When a process is to be loaded it's pages could be assigned to any free page frames. This eliminates the external fragmentation problem that could occur with dynamic partitioning. It also minimises internal fragmentation because only the last loaded page frame will suffer from internal fragmentation. This fragmentation will be half the page size in general [9].

For every memory reference the system must localise the correct page frame location for the page to whom the address belongs. This is performed in hardware, otherwise the time consumption when doing all the address calculations and lookups would be to great. The hardware architecture needed to support paging is shown in Figure 8.

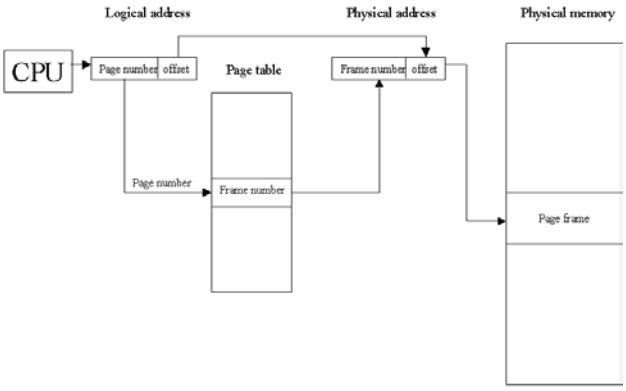


Figure 8 Address translation in a paging system

Every address generated by the CPU, a logical address, is divided into two parts, a page number and a page offset. The page number is used as an index into a page table. The page table is constructed during loading of pages into main memory. The page table provides a mapping between logical addresses and the physical addresses, by storing the base address of each page that have been loaded into memory at index location corresponding to the page number in the table. The base address from the page table is concatenated with the offset in the logical address to give the physical address corresponding to the wanted logical address. Each process will have it's own page table. This will give each process a contiguous view of the memory space although a process pages could be scattered around in main memory. Figure 9 illustrates this. Two process are executing and both have three pages loaded into main memory. From the processes view the memory seams to be contiguous but as seen in the figure the pages could be scattered around in main memory.

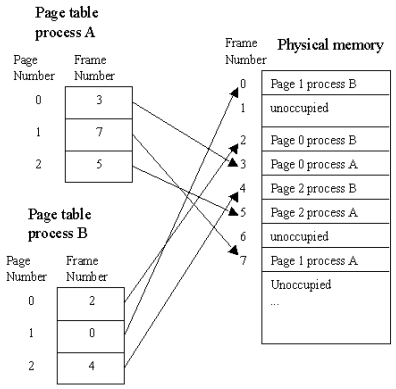


Figure 9 Example two processes and their page tables

The page table can be relative large and is therefore itself often stored in main memory. Thereby requiring two memory accesses for accessing one memory location. When a relative small number of page entries is to be stored in the page table, it can be implemented in dedicated hardware registers. A possible and common way to avoid the need of two memory accesses is to use a cache for page table entries. This cache is called a table lookaside buffer (TLB). The TLB will contain the page table entries (PTE) for the most recent used pages. Figure 10 shows the resulting hardware of using an TLB with paging. Given a logical address to translate to a physical address the TLB is first examined. If the wanted PTE is found (a TLB "hit"), the frame number is obtained and the physical address if formed. If no correct PTE is found in the TLB (a TLB "miss") the page table will be used as before.

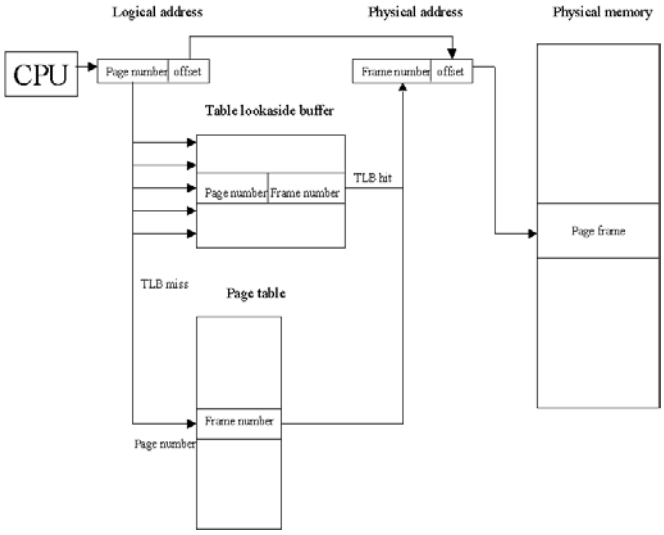


Figure 10 Paging using a table lookaside buffer

Super pages
When very large data structure are used they could flood the TLB and thereby prevent the TLB to be work in a efficiently manner. To solve this problem some system have introduced super pages. A super page is roughly a group of pages mapped by a single entry in the TLB or in a specialised block TLB. Talluri, Kong, Hill and Pattersson, [2] showed that large performance gain can be achieved when reducing the number of TLB entries needed to cover the working set by allowing super pages.

Summary
Paging is often used in virtual memory environment and a continued discussion on paging will be found in the virtual memory chapter.

The difference between paging and fixed size partitioning is that the pages are relative small compared to a fixed partition. With paging a process may occupy several pages that doesn't need to be contiguous in main memory, increasing memory utilisation compared to fixed or dynamic partitioning. Paging also solves the fragmentation problems with the partitioning schemes.

3.2.4 Segmentation

Still another way of partitioning main memory is by segmentation. With segmentation a process is divided into segments. The length of the segments may differ but there is a maximum length. The use of unequal sized segments make segmentation similar to dynamic partitioning, with the difference that a process may be loaded into several segments instead of one partition. Segmentation as with dynamic partitioning theoretically suffers from no internal fragmentation. External fragmentation can occur but less frequent then with dynamic partitioning since each process is divided into several segments that need not to be contiguous located in main memory.

With segmentation a logical address contains a segment number and an offset. There exists no easy relationship between logical addresses and physical addresses. As with paging a segment table can be used to locate the segments belonging to a process. An entry in the segment table will have the base address of the segment, also the length of the segment is needed. The length is needed to avoid invalid addressing outside a segment, as the number of bits used for the offset only sets the maximum size restriction for a segment and thereby accesses outside segments is possible. An example of an system using segmentation is given in Figure 11. To allocate space for a segment in main memory, possible all the allocation techniques presented in chapter 3.1, could be used. Depending on which algorithm is used the segmentation schemes suffers more or less of internal and external fragmentation. The solution for big segment tables is the same as for paging.

One advantages of segmentation is that protection comes natural since the segment is a semantically defined region of a program. Making it easy to include protection bits in a segment entry to provide fine-grained protection (read-only, read/write, execute only etc..) for a segment. Sharing of memory is another advantage in a segmentation system. Each process has it's own segment table and by letting several processes have the same segment table entry in their segment table, the segment becomes shared (doesn't need to have the same segment number). It also makes it possible to share only some part of the program by declaring the part to be shared, as a segment.

The main disadvantage of the segmentation technique is the external fragmentation. The external fragmentation problem can be somewhat relived with the use of compactation. The more complex algorithm needed to locate space in where to put segments in main memory opposed to paging is another disadvantage.

Figure 11 shows an example of segmentation with two processes sharing the same code segment and having individual data and stack segments.

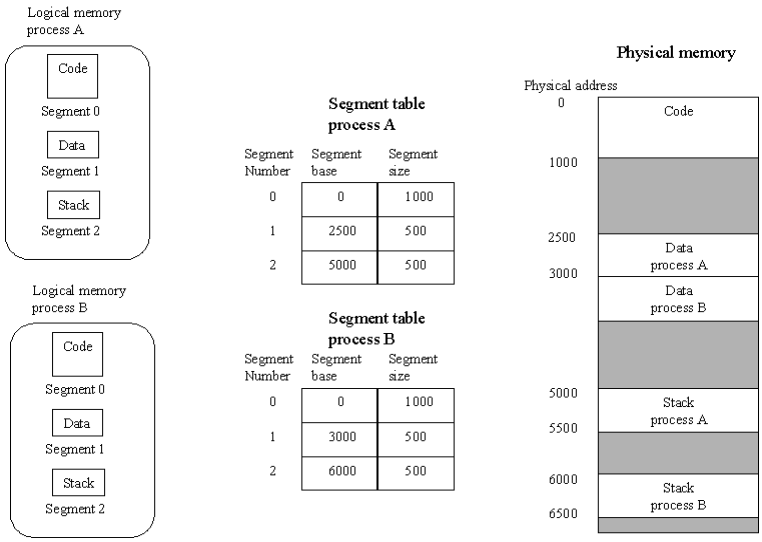


Figure 11 Example of segmentation

3.2.5 Combined Paging – Segmentation

The strengths of segmentation, modularity, sharing and protection, can be combined with the strengths of paging, transparency to the user and no external fragmentation. In a combined paging-segmentation system the user logical address space is first divided into a number of segments, the segments a further divided into a number of fixed-sized pages, with size equal to a main memory frame. If a segment is smaller then one page, one page is used for the segment. The user perceives the same view of the memory as with segmentation, a logical address consists of a segment number and an offset. The system however divides the offset into two parts, a page number and a page offset. An address translation is performed in the following manner. The segment number is used to find an page table for the segment, The page number part of the logical address is then used to index the page table to find the frame number for the corresponding page frame. The frame number is then concatenated with the page offset part of the logical address to form the physical address. Figure 12 illustrates the combined page-segmentation address translation.

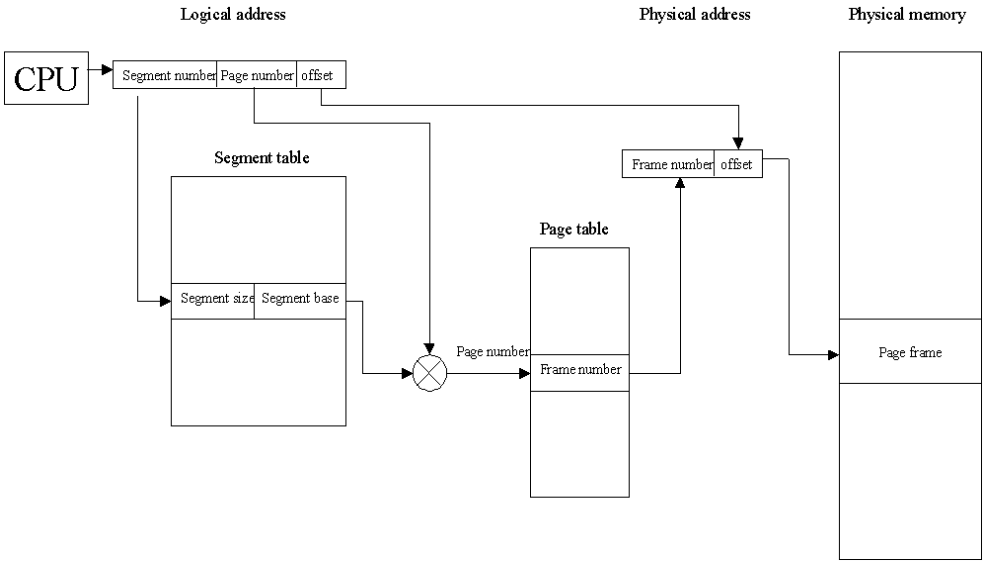


Figure 12 Address translation with combined paging-segmentation

3.3 Virtual Memory

Virtual memory is a technique that allows execution of processes that may not be completely loaded into main memory, thereby making it possible to execute programs larger then main memory. With virtual memory a large secondary memory space, usually a storage disk, is used to extend main memory. The user perceives both the main memory and the secondary memory as one large space to execute programs in, although programs only can be executed in main memory. The combined main memory and the secondary memory makes up a large virtual address space, which is separated from the physical address space constrained by the size of main memory. The main advantages of virtual memory is :

- The size of a program is not limited by the amount of physical memory.
- Because a process need not to be completely loaded into physical memory more process can be loaded and run at the same time. Thereby increasing CPU utilisation.

To be possible to apply a virtual memory scheme there must exist mechanisms for translating virtual addresses to physical addresses and features to move pieces of memory between main memory and secondary memory when needed. The first requirement can be met by using paging or segmentation. The second requirement is often solved by software.

Some aspects of virtual memory that are not presented in this report but are covered in most operating system literature are: frame looking, page buffering for improving page replacement, replacement policy and cache size, resident set management that is how many pages a process should be allowed to allocate and when replacing page should it replace its owns pages or other processes, multiprogramming level and thrashing and various other aspects.

3.3.1 Virtual memory with Paging

To support virtual memory with paging not many changes need to be done to the already presented paging mechanism. Some control bits need to be added to a page table entry. There need to be a present bit which tells if the wanted page is in main memory or not. A modified bit is also required. The modified bit indicates if the content of a page has been modified since the page was brought into main memory. The information is used to determine if the page should be written back to secondary memory, if the page is to be replaced. Also there could be control bits for managing protecting and sharing at the paging level. Frame looking could also be provided to prevent pages from being replaced in main memory.

When the virtual address space grows large the page table structure grows larger accordingly. The page tables might become so big that they need to be paged themselves, this is called multilevel paging. The rapid growth of the page table is because it's scaled with the virtual address space. If the size instead could be scaled with the amount of physical address space, the needed size of the page table should be reduced. A technique to accommodate this is by using inverted page table.

Inverted page table
In a inverted page table there exists one entry for each page frame in physical memory. In this way the page table scales with the physical address space instead of the virtual address space. An inverted page table entry (IPTE) consists of a virtual page number and a process identifier. The index into the page table is the frame number instead of the other way around as with ordinary paging. The process identifier tells which page is owned by which process. To find a frame number corresponding to a virtual address the inverted page table need to be searched. The entries in the table are compared with the virtual address until a match is found. The index to the matching IPTE is the wanted frame number. The frame number are then combined with an offset as with usual paging to form the physical address. To reduce the search time a hashing function is often used to construct and search the inverted page table. A hashing value is calculated on the process Id and the virtual page number. The hash value index a hash table where a frame number to a page is stored if a page exists for this hash value. The frame number, index the inverted page table and a comparison is made between the IPTE and the virtual address (excluding the offset). If a match the physical address will be formed. However there could be several combinations of virtual page numbers and process id's that produces the same hash value. This could be solved by chaining the IPTE's that should be accessed by the same hash value. Then when a compare of the IPTE and the virtual address produces a miss the chain is traversed to see if there is any other IPTE's that produces a match, otherwise its a page fault. Figure 13 illustrates the idea of an inverted page table using a hash function to perform the search through the inverted page table. As an example the powerPC uses an variation of the inverted paging scheme.

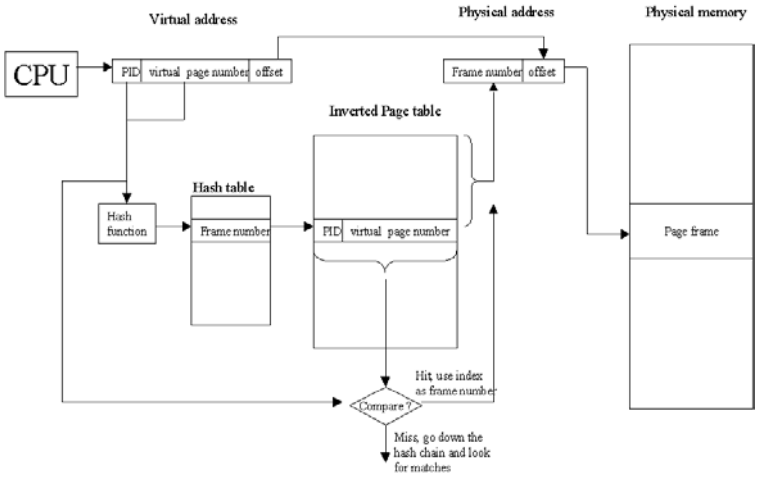


Figure 13 Hardware architecture for inverted paging

3.3.2 Virtual memory with Segmentation

With the use of segmentation to achieve virtual memory, the same modifications done to the paging mechanism in the previous section, must be done to the segmentation described in a previous chapter. A present bit and a modified bit needs to be added to the segment table entry.

3.3.3 Virtual memory with combined paging-segmentation

The present and modification bits are handled at the paging level in the same manner as in virtual memory with paging. Protection and sharing are often implemented at the segmentation level.

3.3.4 Fetch policy

There must be a decision made on when to bring in pages belonging to a process. Two strategies exists, demand paging and pre-paging. With demand paging a page is brought into main memory when a page fault occurs, that is when a referenced page is not in main memory. With this policy when a process starts up it hasn't any pages loaded into main memory and the page fault rate is initially high but drops as more pages are loaded into main memory. Pre-paging load a bunch of pages additional to the one causing the page fault. The motivation behind this scheme is that it is more efficient to load several pages at one time, because they are often stored contiguous in secondary memory and thereby reducing the seek and rotation latency for a specific page. This policy could be ineffective if the extra brought in pages are not referenced in the future.

3.3.5 Replacement policy

If main memory is full when a new page is to be loaded some other page has to be replaced. The optimal way of doing this would be to replace the page that will not be used for the longest period of time, this is called an optimal replacement algorithm. It is an impossible method to implement since it requires future knowledge of page usage. However it could be used to evaluate the performance of replacement algorithms, using simulation where the future of page references is known. The replacement algorithms in use try there best to approximate an optimal algorithm. Replacement algorithms are discussed in most operating system literature, for example in [8], [9] and [20]. Example of algorithms are First-in First-out(FIFO) and Last recently used (LRU).

3.4 Linking and loading

To create an active process it must be loaded into main memory and a process image must be created. A process typically consists of a number of compiled or assembled modules in object-code form. A linking stage are used to resolve reference between modules, it also resolves to library routines. The library routines may be included into the program or referenced as shared code that must be supplied by the operating system at runtime.

3.4.1 Linking

The linker stage take as the input object modules and produce a load module, which will contain the program and data module to be passed to a loader. In an object module there could exist references to other object modules. The linker solves these references and creates a single load module joining all referenced object modules. The linker must replace the reference to a object module with a reference inside the load module itself. This could be called static linking and has some drawbacks. Each load module must have a copy of the system library function its use instead of all processes sharing a common libraries. This lead to waste of memory.

Dynamic linking
To support sharing of libraries and other common code dynamic linking can be used. With dynamic linking the linkage of some external modules are postponed until after the load module is created. The load module will therefor contain unsolved references to other programs. These references could be managed during load time or during run time. When dynamic linking is used and an unresolved memory reference appears the operating system (or loader if load-time dynamic linking) locates the module wanted and links it to the calling module. There are several advantages when using dynamic linking :

- Its possible for processes to share common code in libraries and thereby only using one copy of the library and reducing memory usage.
- Updates to the target module, i.e. a OS utility or a general purpose routine, could easier be made. Because change to the module could be made without the need for relinking applications that runs on the target module.
- Automatic code sharing becomes a possibility. The Operating system recognises that several applications is using the same target code because the OS loaded and linked that code. Then the common code can be lifted outside the application code to form a single copy and link that copy to the applications instead.
- Independent software developers could more easier develop system utilities and pack those as dynamic link modules to be used by several applications.

Loading
To get a program up and running it must be loaded into main memory the issue here is where in main memory it should reside arise. When loading a load module into main memory the address requirement, Figure 1, must be solved i.e. address binding must take place. Three main strategies can be taken :

- Absolute loading
- Relocatable loading
- Dynamic run-time loading

Absolute loading
With the absolute loading technique the load module must always be loaded in the same memory region. Thus the load module contains absolute memory reference and the load module must be placed according to them. The absolute value assigned to memory references can be done by the programmer at program time or by the compiler at compile time. Absolute loading have several disadvantages. First the programmer must know how the load module should be placed in memory layout. Second if there is to be made any changes in the body of the load module the alteration has to be done for all other addresses just not the ones effected by the change. The second disadvantage can be avoided by assigning absolute values to memory references at compile time and not at programming time.

Relocatable loading
To overcome the disadvantages of absolute loading, relocatable loading is introduced. With relocatable loading the compiler instead of producing load modules with absolute addresses it

produces addresses that are relative to some known point, i.e. the start of the load module. When the loader shall bring a load module into main memory it puts the load module at some starting location *x,* then it adds *x* to all memory references in the load module. The disadvantage of relocatable loading is when there is need to swap parts of main memory out to secondary memory during run-time. With relocatable loading the swapped out pieces must be swapped back into the same address from where it where swapped out this because the addresses are bound during the initial load time.

Dynamic run-time loading
With dynamic run-time loading the load module remain the relative addresses while loading. The relative addresses isn't bound to any absolute address before an actual memory reference takes place during run-time. This solves the problem with the relocatable loading scheme. To calculate address during runtime the systems should use a dedicated hardware (MMU) to perform the translation, otherwise the system performance would degrade to much. Dynamic run-time loading gives a lot of flexibility, a program can be loaded anywhere in memory and if swapping occurs the swapped out pieces could be swapped back into different memory locations.

4 Existing software solutions

A total of four real-time operating system has been examined. These are eCos, Ose, VRTX and vxWorks. Investigation of how dynamic memory allocation is performed is the main focus. The memory protection property of the memory management has been examined. The hardware MMU support of the RTOS has also been covered.

4.1 eCos

If not otherwise stated the information regarding eCos has been gathered from [7].

4.1.1 Introduction

eCos, **e**mbedded **C**onfigurable **o**perating **s**ystem, is an open source real-time operating system developed by Redhat Inc and avliable free of charge at http://sources.redhat.com/ecos. eCos has support for the following platforms : ARM, IA32, Matsushita, AM3x, MIPS, NEC V8xx, PowerPC, SPARC and SuperH architectures. eCos also supports the µ ITRON version 3.02 specification.

4.1.2 General memory management layout

eCos development kit (open source) include a memory layout tool and also default memory maps for the supported platforms. With this tool the default memory map configuration can be changed to the actual wanted memory map layout for the specific target board.

By using the eCos Configuration memory layout tool a heap section can be defined. This heap section will be used by the system for managing dynamic memory allocations. If no specific heap section is specified in the memory layout of the system, a static array will be set up by the system configuration tools to form a heap. The size of this static array can be set by the user. It's possible to map around multiple disjointed physical memory locations to serve as a single heap to be used in the dynamic memory management.

There exists no support for the use of virtual memory. Neither is there any support for dynamic run time loading.

4.1.3 Dynamic memory management

There are two types of memory pools that can be used for dynamic memory allocation. A variable size memory pool for allocation blocks of any size and a fixed pool scheme for allocation blocks of a specific size. The size of the blocks for a fixed memory pool is specified when the memory pool is created and the pool can only provided block of this size. Both variable and fixed memory pools can be created and removed from the heap during runtime.

Variable sized pools
The variable size memory pool uses a sequential address ordered first fit algorithm to allocate memory objects. A double linked list of free memory block is maintained. When the pool is created the list consists of a head and one element. The element represent the entire free memory available. When a request for memory allocation is given the linked list is traversed from head on until a large enough block is found to suit the allocation request. If a block of exactly the requested size is found the block is linked out of the free list and the address to the block is returned. If a block lager then the requested size is found, the superfluous piece of that block is chopped of and inserted into the list of free blocks, provided that the chopped of piece is large enough to hold the internal data structures used. Then the address to the allocated block is returned. The complete defined memory pool can't be used because internal overhead from managing allocated block is stored in the pool. If a large enough block can't be found an error is returned.

When freeing allocated objects coalescing is an option. If this option is not turned on fragmentation of the memory can become an issue. If coalescing is turned off, the memory to be freed is simply put into the list of free blocks. If coalescing is used the memory to be freed is concatenated with any surrounding free memory block in the free list.

Fixed sized pools
The fixed size pool memory management scheme uses an bitmapped fit technique. The implementation uses a simple array of bits to represent each block in the pool. When a allocation request is given a linear search through the bitmap will find the first available block for allocation and set its corresponding bit in the bitmap to be allocated. When a block is freed the bit in the bitmap representing the block is cleared. The data structure for maintaining the bitmap is stored in the same space as the pool, thereby reducing the size that can be allocated from the pool to less then the pool size.

If a fixed memory pool is used to store types that have alignment constraints (such as 4-byte alignment), then its up to the user to align the memory appropriately for the type in question, or use a memory pool that is an exact multiple of the required alignment.

As mentioned above the memory available from the memory pools will not be the same size as the memory supplied to it. Some of the memory is used for internal data structures of the allocator. To determine how much memory really is available the cyg_mempool_fix_get_info() or cyg_mempool_var_get_info(), depending on pool type, can be used.

4.1.4 Security

In a study on the memory protection in eCos, [5], its stated that eCos at present offers little memory protection because :

- All memory is accessible from any code area (both read and write)
- All code is executed in a privileged mode (supervisor), except for hand-written wrappers around the exception vectors whose purpose is to switch the processor *into* supervisor mode.
- All RAM is mapped in one large chunk to address 0x00000000

4.1.5 Sharing

To allow threads to cooperate and compete for resources eCos provides the classic inter-task communication mechanisms mutexes/condition variables, and semaphores. Other synchronisation/communication mechanisms provided are event flags and message queues/message boxes.

4.1.6 Support for Hardware MMU

No information has been found about any eCos implementation that take full advantage of MMU for any of the supported architectures. However development of MMU support for some architectures are probably going to be developed in a near future.

4.1.7 System calls

The system calls used to handle dynamic memory comes in three forms, blocking, non-blocking and timed-wait.

System Call	Function
Cyg_mempool_var_create	Creates a variable block size memory pool
Cyg_mempool_var_alloc	Allocates a block of a variable size. This call will put the calling thread into blocking state until enough memory becomes available to serve the call.
Cyg_mempool_var_timed_alloc	Allocates a block of a variable size. This call will try for a given time to allocate memory before giving up and returning.
Cyg_mempool_var_try_alloc	Allocates a block of a variable size. Returns an error signal if memory wasn't available.
Cyg_mempool_fix_create	Creates a fixed block size memory pool
Cyg_mempool_fix_alloc	Allocates a block. This call will put the calling thread into blocking state until enough memory becomes available to serve the call.
Cyg_mempool_fix_timed_alloc	Allocates a block. This call will try for a given time to allocate memory before giving up and returning.
Cyg_mempool_fix_try_alloc	Allocates a block. Returns an error signal if memory wasn't available.

If the eCos standard C library implementation is configured to be used in the system the following c familiar calls is supported :
malloc(), calloc() and realloc().
These function is implemented using the try_alloc call.

4.2 OSE

4.2.1 Introduction

OSE Supports a large amount of CPUs, here are some of the CPUs that are supported:

- PowerPC
- Strong ARM
- Motorola 68K
- MIPS

The information about OSE has been gathered from [13], [14] and [15]. There are four levels of OSE implementation. The levels are A, B, C and D where D is the most complex implementation. On level D, a hardware MMU fully supported. This chapter will focus on the D level.

4.2.2 General memory management layout

In OSE a contiguous area of memory is called a pool. Signal buffers, stacks and kernel areas can be allocated from a pool. In the system there must be at least one pool, this is the global system pool. The system pool is crucial for the kernel and is always located in the kernel memory. It is possible to let all processes allocate memory from the system pool. But if the system pool is corrupted the disadvantage is that the entire system will crash. To avoid that the hole system crash OSE provides the alternatives shown in Figure 14.

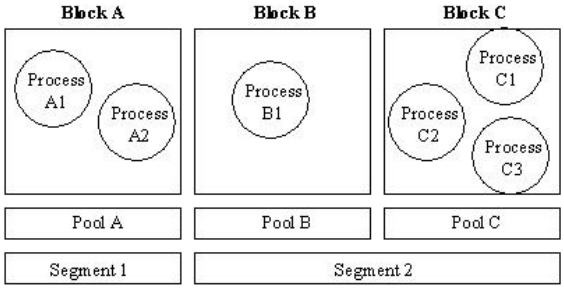


Figure 14 Blocks, local pools and segments in OSE

As seen in the figure above the OSE processes can be grouped into process blocks, a memory pool is attached to the block. The blocks can be grouped together into process segments.

Blocks and local pools
The block can allocate memory from the global system pool, or they can have their own memory pool. If a local pool is corrupted only the processes in blocks connected to that pool will be affected. Processes in other blocks will continue to execute normally if they don't depend on a process in the corrupted block. It can also be convenient to treat all processes in a block as one process, many system calls that operate on single processes also works on a block. This makes it for example possible to start or kill all processes in a block. Another benefit is that signals can be sent between blocks, signals to a block are redirected by a block redirection table.

Segment
To improve security blocks and their pools can placed in separate segments. Processes within the same segment can access memory in other pools, but they can only allocate memory in their own pool. If the signal is sent between processes located in different segments, the user can choose to send the signal by reference or to copy the signal contents.

4.2.3 Sharing

Interprocess communication
In OSE there are different ways to synchronise or communicate between processes. The option are as follows.

- Signals
- Semaphores
- Fast semaphores
- Environment variables

Signals
The most common way to communicate between processes is by using signals. A signal is a message that is sent from a process to another process or block. For a process to be able to send a message to another process it must allocate a memory buffer for the signal, it also need to know the receiving process identity. The signal consist of different parts, a signal number, data contents, signal owner, size of the buffer, sender and receiver. When a process gains access to a buffer by allocating it or by receiving it from another process, it becomes the owner of the buffer. When a signal has been sent to another process the sending process give away the ownership of the signal to the receiving process.

Operations on the buffer can only be performed by the owner. If the sending and receiving processes are located in different segments, there is two possibilities, transferring a pointer to the buffer or copying the signal buffer. OSE support redirection tables, a process that receives a signal can redirect it to another process, the same behaviour counts for a block that receives a signal. All processes have a queue for the signals sent to it.

4.2.4 Security

OSE have one single memory address space and no virtual memory is used. This means that the memory protection is very poor. Full security can only be achieved if the pools are put into different segments and the segments are isolated by a supporting hardware MMU.

Using signals in interprocess communication give some protection because of the ownership function. Higher degree of protection can be achieved if the method of copying the signal buffer when its send between pools in different segments is used.

4.2.5 Dynamic memory management

Allocation of memory from a memory pool is allowed if the caller has access to the pool, i.e. if the caller belongs to the block that manages the memory pool. Signals can be dynamically allocated. Eight different sized signal buffers can be used. A process making a signal allocation owns the signal. The signal buffer is stored in the control block of the process, it also get a reference ID that can be used later. When freeing a signal buffer, the memory is returned to the pool associated with the block. Its only the process that owns the buffer that is allowed to free it. Signal allocation can be used for dynamic variable allocations, but this gives an overhead according to the layout of a signal, only the buffer is used as a variable. OSE also support dynamic allocation by the use of a heap.

Processes
A process can be dynamically created if the corresponding code is defined as a dynamic. The process is set as part of a specified block. The new process gets a process ID number (PID). The PID is later used for system calls to the process, for example when the process should be removed.

Memory pool
Pool space can be dynamically allocated by the memory manager, the pool is attached to a specific block. The allocated pool is entirely managed by the kernel until all blocks using the pool are dead, pool space then may be reclaimed by the system without further notice to the kernel.

Local block
When a new block is created the segment of memory is inherited from the caller, i.e. all processes in the new block will execute in the same memory space as the creator unless another memory segment is attached to the block. A descriptor and a block ID is created for the new block. The descriptor for the block is removed from the system when either the block is explicitly killed, or the last process is killed.

Segment
Dynamic allocation of a segment is also possible, but the system must have a MMU. For further information see the chapter about *Support for hardware MMU.* Its possible to copy a block of memory from one memory segment to another, of course there must be enough memory available in the receiving segment.

Heap managed memory
OSE also provides a optional heap to be used for dynamic memory allocation. The basic allocation algorithm uses a buddy system with a fibonacci series to distinguished the block sizes. The smallest size is 16 byte and the largest roughly 8 Mb. When a allocation request is made it is rounded up to the nearest size in the fibonacci series. A fixed overhead of 9 bytes is added to the size. Allocations from the heap can be made private to the process performing the allocation, or shared with all other process in the same segment.

Private allocated memory will be automatically freed when the process who owns it is terminated. Shared allocated memory can be free by all process in the same segment. As with all buddy systems the main problem is the internal fragmentation. The use of a fibonacci series reduces internal fragmentation compared to the usual power of two series used with buddy systems. OSE guarantee that the internal fragmentation doesn't exceeds 61%. According to OSE, allocation and free requests has a bounded (deterministic worst case) execution time.

4.2.6 Support for Hardware MMU

The OSE memory management system (MMS) enables OSE applications to take full advantages of a hardware MMU. The MMS integrates the OSE kernel memory management features with the MMU functionality. At this moment the MMS only support PowerPC processors. For other processors with hardware MMU user defined system software must be created to handle the MMU.

At start up the MMS make a configuration of the static memory. It also sets up a memory protection map, which includes the drive routine for the cache hardware and the MMU that's on the specific target.

Regions
A contiguous part of memory can be put together to a memory region. The regions can be both global and local. The memory regions are statically configured at start-up, but it's also possible to dynamically allocate regions. For both the static and the dynamic regions the base address and size must be aligned to a page boundary of the MMU. Depending on the wanted behaviour MMS together with the MMU can define the following characteristics for a region.

- Read-only, read-write.
- Only read-caching, no caching or caching with copyback
- Resident or non-resident.

If a region is static it's automatically made resident and therefor it can never be deleted. For dynamic regions there are two options, making the region resident means it's never deleted. If the region is made non-resident it's automatically deleted when the segment is killed. If one region in a segment is resident then the hole memory segment is considered resident.

One or more of the global or local regions are then collected to a global or local memory segment. The global segment can always be accessed. In both cases the data access is additionally limited according to the access properties of each region, which can be different in supervisor mode and in user mode.

Global memory segment
With MMS all global regions, static and dynamic, is assigned to the global memory segment, this segment is always static and can therefor never be deleted. Because all global regions belong to the global memory segment they can be accessed and shared by all the users. There must always exist a global memory segment in the system.

Local memory segments
A local memory segment is a collection of local regions. If the memory segment is static it can only contain one static local region. MMS ensures that in user-mode, processes only has access to it's own memory regions, if the wanted region is in another local memory segment access is denied. If the process is in supervisory mode it can access all regions.

The MMS also allows dynamic allocation and deallocation of memory segments. When a segment is dynamically created is receives a segment ID, the segment ID can then be used by other system calls, for example to attach and kill processes and local blocks. If the system has a MMS there is also an alternative to kill a memory segment, then all of the processes and blocks in the segment is deleted. If the segment is killed the memory manager can then reclaim the memory segment.

The kernel notifies the MMS what type of processes that reside in the specified segment. If the segment contains at least one timer-interrupt process or interrupt process, the MMU must make the hole memory segment resident, i.e. disable swapping. For the other types of processes swapping is enabled.

Dynamic linking and loading
For a run time linking and loading of programs the MMS administrates dynamic allocation and de-allocation of memory. If a new program should be downloaded the program loader requests dynamically allocated memory segments from the MMS.

4.2.7 System calls

Basic calls

Function	Description
Alloc	Within the pool that is available to the caller's block a signal buffer is allocated and a pointer is returned. The calling process becomes the owner of the buffer. The call is also used as an ordinary allocation function.
Free_buf	Returns a signal buffer that is freed to the pool associated with the caller's block. It's only the process that owns the buffer that is allowed to free it.

System Calls for Dynamic Processes

Function	Description
Attach	Stores a signal buffer owned by the caller in the control block of the specified process or block. It returns a reference ID that later can be used by other calls. The call can also be used to supervise a memory segment, a segment ID is then passed as an argument to the attach call. If the segment is killed the memory manager can then reclaim the memory segment.
Create_block	Creates a block descriptor owned by the specified user number and returns the block ID. The descriptor for the block is removed from the system when either the block is explicitly killed, or the last process is killed.
Create_process	Creates a process as part of the specified block and returns the process ID.
Detach	Removes a signal that has been attached to a process or block by the caller.
Get_bid	Finds the block that the specified process is part of and returns the block ID.
Get_pri	Returns the priority of the process, that is specified by the process ID.
Kill_proc	If the argument is a block ID, all processes known to the caller are killed in that block. If the argument is a process ID, only the specified process is killed. If the system has a memory management system (MMS), the call can be used to kill a memory segment. Then all of the processes and blocks in the segment is deleted.

System Calls for Memory Management
The following are normally not user system calls. The calls operates between the kernel and the memory manager block (MMU). If the calls should have any effect the OSE memory management system, MMS, or a user defined software must be used.

Function	Description
Attach_block	When a new block has been created the MMU software is notified by the kernel. Any allocation of memory issued by the function refer to the system pool.
Attach_segment	The function returns segment ID, the segment ID can then be used by other system calls, for example attach and kill_proc.
Create_pool	The MMU software allocates and creates a new memory pool and attach it to a specified block. The kernel manages the memory pool as long as there are at least one block that hasn't been killed.
Get_segid	Find the segment that a specified block or process is part of and returns the segment ID.
Mem_move	The function copies a block of memory from one memory segment to another. It's the MMU software that performs this operation. Any allocation of memory issued by the function refers to the system pool.

Select_segment	A call from the kernel requiring the MMU to select the address space where the indicated memory segment resides.
set_segment	The effective segment number for the calling process is set by the MMU software.
set_segment_mode	The kernel notifies the MMU what type of processes that reside in the specified segment.

4.3 VRTXsa

4.3.1 Introduction

Mentor Graphics VRTX Real-Time Operating System (RTOS) support at least the following architectures : Motorola 68xxx and Motorola CPU32, Motorola/IBM PowerPC, Motorola ColdFire, ARM Ltd ARM, MIPS. The information about VRTX has mainly been gathered from *VRTXsa real-time kernel, programmers guide and reference* [1].

4.3.2 General memory management layout

The memory of a VRTX system will be separated into modules:

- VRTXsa Code
- Application load module
- VRTXsa managed memory which is divided into a workspace and a application managed memory

The application load module holds the application code and any system code defined by the user. The application load module also contains system address table, exception vectors, configuration table, processor control block and any static variables. The application load module is statically determined before run-time and could therefor be placed in none dynamical memory, i.e. ROM.

The workspace module contains system variables, task control blocks, control structures for queues, flags, semaphores, control structure for the application workspace, interrupt stack, idle stack and stack areas for each task in the system. The size of the workspace is defined in a configuration table. The workspace must be large enough to hold the VRTX system variables (one TCB for each task) and stack areas respectively. In addition the workspace must also accommodate a control block for each memory partition in the application memory. The workspace is heap allocated with the initialisation of the system from the VRTXsa managed memory space.

The application managed memory module consists of space for user handled partitions and/or heaps from which tasks can dynamically allocate memory..

4.3.3 Dynamic memory management

VRTXsa provides two approaches to perform dynamic memory allocation. Partition managed memory and heap managed memory.

Partition management
Partitioned memory is allocated and returned in fixed sized blocks. These blocks can be dynamically allocated during run–time. Different partitionons can have different block sizes. The partitions can also be allocated, resized and deallocated during execution as long as the defined workspace memory for handling the partitions is sufficient. Partitions can be created inside other partitions to achieve different block sizes in the same memory area. Further partitions can be created to cover the same memory region, which allow allocation of different sized block in the same memory region given that all blocks of one size is first released before allocation of another size occur.

With partitioned memory, allocation and deallocation of blocks can be made in constant time.

Heap management
The heap management is based on the algorithm described in *Design of a General Purpose Memory Allocator for the 4.3 BSD UNIX Kernel*, [27]. In general it consists of a combined sequential address ordered first-fit allocator for large blocks and a simple segregated storage allocator for small blocks. The goal is to take advantage of the allocation speed of simple segregated storage for small blocks and switch to a slower but more space efficient fast-fit allocator for large blocks.

A heap is divided into a number of pages. The size of a page should be a power of 2 and is typically between 512 and 8192 bytes. The page size of a user defined heap is set when the heap is created. The workspace heap page size is set to 512 bytes. The boundary between large and small blocks is set to two times the page size. The data structures for representing and managing a heap is allocated outside the heap in the workspace when a heap is created. So a heap allocation request has no internal overhead.

For small block sizes a simple segregated storage allocator with size class divided into powers of two is used. There is a minimal block size of 8 byte. Allocation request for smaller block then the minimum size is rounded to the minimum size. When an allocation request for a small block is made, the request is rounded up to a power of two, then the free list for the corresponding size class is consulted. If a free block exists it is removed from the free list and returned to the caller. If the free list is empty a whole page is allocated from the large block list to the free list of the requested size class. The first block in the newly allocated page is returned to the caller and removed form the free list. When a small block is freed it's simply returned to the corresponding free list. No attempt is made to coalesce returned block, i.e. when a page has been assigned to a specific size class it is dedicated to this size class forever. As said in chapter 3.1 this could introduce severe external fragmentation.

For large block sizes a sequential address ordered first fit allocator is used to allocate blocks which is multiples of the page size, i.e. an allocation request is rounded to the next larger multiple of the page size. Large block are coalesced with it's neighbours if possible when freed.

Allocating with the small block allocator is faster then allocating with the large block allocator.

Heaps offer greater flexibility then partitions but as a consequence performs worse regarding allocation speed and determinism.

4.3.4 Security

In a VRTXsa system all application tasks and system shares the same address space. A faulty application can therefore accidentally access system resources or other applications resources, if the MMU support, described below, is used these kinds of protection faults can be detected.

Passing invalid addresses to heap and partition creation give undefined results. When freeing a heap allocated block a check is made to ensure that the block provided to the freeing function reside in the heap and is current allocated.

4.3.5 Sharing

VRTX provide the following mechanisms for exchanging data between tasks, synchronize tasks and to mutual exclude task from each other:

- Mailboxes
- Queues
- Event Flags
- Semaphores

4.3.6 Support for Hardware MMU

From VRTX 5.0 hardware MMU support is added [30], for target processors/cores that has an MMU. The MMU support is an option and can be disabled. The MMU support mainly exsists to support selective cache control, i.e. mark which pages that should be cacheable or not.

The MMU can be programmed to trap common programming errors [29] such as:

- Null-pointer and other illegal memory access.
- ROM and unpopulated memory accesses.
- Access to system resources for user level tasks.
- Threads that under flow or overflow their stacks.
- Attempts to execute data.
- Attempts to overwrite code.

Also there exist a VRTX variant, VRTX x86/spm which is developed for the Intel 80x86 architecture. It takes full advantage of the Intel 80x86 memory management hardware and the MMU features it provides.

4.3.7 System calls

Partition management
The table below presents the calls used in VRTXsa to handle partition managed memory.

sc_pcreate	Creates a partition of contiguous memory managed by VRTX. The sc_pcreate call specifies the partition ID number, partition size, block size and partition address. The partition size must be greater then or equal to the block size. In addition the partition cannot contain more then 32k blocks, although the partition can be extended using sc_pextend call. To avoid wasted space the partition size should be an integer multiple of the block size. Block size is used in sc_gblock to obtain memory blocks. The call can be mad from task code or initialization code.
sc_pextend	Extends a memory partition defined by sc_pcrete. The extension adds an additional range of memory to the partition. This newly added memory range dose not need to be contiguous with the partition it extends. The block size for the extension will be the same as for the original partition. An extension can not be larger then32k block but the total amount of blocks together with the original extension can exceed 32k blocks.
sc_gblock	Obtains a memory block from a partition of memory block that have been created using sc_pcreate. The block size has been specified when the partition was created.
sc_rblock	Realeases a memory block to the partition from where it came. All blocks of a partition should be release before a task that have created a partition is deleted because no automatic deallocation of blocks are made when tasks is deleted.

Heap management
The table below presents the calls used in VRTXsa to perform dynamic memory allocation on a heap.

sc_halloc	Allocates a memory block form a specified heap.
sc_hcreate	Creates a heap from a area of contiguous memory.
sc_hdelete	Removes a specified heap, which makes the heaps controlblock and system memory reusable.
sc_hfree	Returns a memory block to the heap from where it was allocated.
Sc_hinquiry	Used to get information about number of allocated blocks, freed blocks and the $\log_2$ size of the heap.

4.4 VxWorks 5.4

4.4.1 Introduction

VxWorks is developed by Windriver systems, and was initially a development and network environment for VRTX. Windriver developed their own microkernel wind later to replace the original VRTX kernel. Tornado II also from Windriver is a development environment for VxWorks. The information about VxWorks has been gathered from [10], [11] and [12].

VxWorks support a large number of processors, the architectures currently supported are: Motorola MC68*0*x0, Sun SPARC, Intel i960, i386, i486, Pentium, and PentiumPro, MIPS, PowerPC and ARM

4.4.2 General memory management layout

In a VxWorks system the memory will be separated into different modules. The memory layout differs depending on the architecture, but the following modules are always represented.

- System image
- System memory pool

System image
The system image module contains a VxWorks system image for a specific target board. The system image consists of all desired system object modules linked together into a single non-relocatable object module. If an application should be bootable it can be a part of the VxWorks image, and that makes the application a part of the system image. The application can also be downloadable, this is described in the dynamic memory chapter.

System memory pool
The system memory pool is where the user handled memory is located. Allocation for dynamic module loading, task control blocks, stacks, memory for partitions, and so on, all come from the system memory pool. Default in the VxWorks system is that the pool is set to start immediately following the end of the booted system, and to contain all the rest of the available memory. If the system has other non-contiguous memory areas, it is possible to make them available in the general memory pool by later calling the system call *memAddToPool*. To determine the available memory a architecture dependent routine is used. The system memory pool is created when the kernel is initialised by the command *kernelInit*. The system memory pool is managed by system calls in the library memPartLib.

Partitions
When the kernel is initialised a partition is created, the system memory partition, which contains the system memory pool. Connected to the system memory partition is a memory free list to keep track of the free memory. It is possible to create partitions from the system memory pool, each new partition contains at least one specified memory pool. When a partition is created it's assigned with a partition ID, and a memory free list corresponding to the memory in the partition. The partition ID can later be passed to other routines to manage the partition.
A descriptor for the partition is allocated out of the system memory pool.

Virtual Memory Mapping
To take full advantage of boards with MMU support, the optional component VxVMI is needed. When virtual memory mapping is used no swapping of pages will take place.

Deterministic behaviour
For boards with a MMU the system can be non-deterministic depending on the architecture, according to page 307 [10].

4.4.3 Dynamic memory management

VxWorks supplies a facility for memory management, that supports dynamic allocating, deallocating, and reallocating blocks of memory from a memory pool. VxWorks uses standard *malloc*() ANSI-C interface for dynamic allocation of memory from the system memory pool. Internally, VxWorks uses *malloc*() for all dynamic allocation of memory.

If a new partition is dynamically created, enough memory for the partition descriptor and the memory pool must be allocated from the system memory pool. It is possible to add more memory to a partition later. The memory added don't need to be contiguous with memory previously assigned to the partition. To allocate or deallocate memory from the user created partition, the partition ID must be an argument to the commands. There are possibilities to allocate a block of wanted size that have a wanted alignment, in this case the block begins on a memory address evenly divisible by the wanted alignment, the alignment must be a power of 2. Its also possible to allocates a block of wanted size that begin on a page boundary if the MMU support is used.

The allocation of memory from the system memory partition or a user created partition, is done with *malloc(),* which implements a first-fit algorithm. Searches for the largest available block in the system memory partition or user created memory partition free list can be done before trying to allocate memory. When a memory block is deallocated it is returned to the corresponding memory free list. If a newly deallocated memory block is adjacent to another free block they are coalesced.

As mentioned before an application project can be linked and downloaded dynamically to VxWorks. The memory for the application is allocated from the system memory pool. This allows objects to be loaded onto a already running system.

4.4.4 Sharing

Inter task communication

VxWorks supplies a set of intertask communication mechanisms.

- Shared memory, for simple sharing of data.
- Semaphores, for basic mutual exclusion and synchronisation.
- Message queues, for intertask message passing within a CPU.
- Sockets and remote procedure calls, for network-transparent intertask communication.
- Signals, for exception handling.

To provide intertask communication for tasks running on different CPUs, VxWorks provide the optional product, VxMP.

Shared Memory Objects

The VxMP option provides facilities for sharing semaphores, message queues, and memory regions between tasks on different processors. It can support up to 20 CPUs. The objects data structures must reside in a memory that all processors has access to. While local objects only are available for tasks on a single processor. The following shared memory objects are used for communication and synchronisation between tasks on different CPUs:

- Shared semaphores can be used to synchronise tasks on different CPUs as well as provide mutual exclusion to Shared data structures.
- Shared message queues allow tasks on multiple processors to exchange messages.
- Shared memory management is available to allocate common data buffers for tasks on different processors.

The advantages with the shared memory objects is mainly that it gives a transparent interface so the same routines that are used for the local objects can be used, there is no need to send packages between the processors and the memory can be placed on a separate memory board.

4.4.5 Security

In a VxWorks system all application tasks and system shares the same address space. A faulty application can therefore accidentally access system resources or other applications resources. To overcome this problem it is possible to use the optional component VxVMI if a MMU is available on the target architecture. The component allow every process to have their own virtual memory. How the VxVMI component works is described in the *support for hardware MMU* chapter.

4.4.6 Support for Hardware MMU

VxWorks provide two levels of support for a processor with a hardware MMU. One basic level and a full level support. This chapter will focus on the full level support for the MMU. The information in this chapter is gathered from [10].

In order to get the full level support VxWorks need the optional component VxVMI. The component allows text segment, the exception vector table and any text segment downloaded to be write protected. Other features are mapping between physical and virtual memory, modifying and examine the current state of the virtual memory, including the ability to make portions of memory non cacheable or read-only as well as a set of routines for virtual-memory management.

Pages state
The physical and virtual memory is divided into pages. Each page of virtual memory has three state information variables:

Valid/invalid	Indicates if the virtual to physical translation is true. Memory accesses to a page marked as invalid will result in an exception. Pages may be invalidated to prevent them from being corrupted by invalid references.
Writable/nonwritable	Indicates if the page is made to be read only or read-write.
Cacheable/non-cacheable	Memory accesses to pages marked as not cacheable will always result in a memory cycle, bypassing the cache. This is useful for multiprocessing, multiple bus masters, and hardware control registers.

Page status can be changed during run-time. Notable is that not all target hardware supports write protection. VxVMI uses the MMU to prevent portions of memory from being overwritten. This is done by write protecting pages of memory. When VxWorks is loaded all text segment and the exception vector table is write protected. An attempt to write to a memory location that is write-protected causes a bus error.

Virtual Memory Context
To implement virtual memory a structure called virtual memory context is used. The context contains a translation table that maps a virtual address to a physical address. The context also contains information about the page state of the memory connected to the virtual memory. It's possible to create several virtual memory context, but there are only one that can be the current. VxVMI provide a mechanism for creating these virtual memory contexts dynamically. For individual task's the contexts are not automatically created, but there is a possibility to create them dynamically. If a context is deleted the underlying translation table is deallocated.

Global Virtual Memory
Global virtual memory is created by mapping all physical memory to the same address in the virtual memory, one-to-one mapping. The global virtual memory is then accessible from all virtual memory context. This must be done so all the needed system objects are accessible for all tasks in the system. The state of all pages in the newly mapped virtual memory is unspecified and must be set.

Private Virtual Memory
In order to protect data by making in inaccessible for other tasks a private virtual memory can be created. To create private virtual memory a task must create a new virtual memory context, witch is stored in the task's TCB. A mapping between virtual and physical memory is then made, for this to work the virtual and physical memory must be defined before the mapping. The memories must start on a page boundary and with a length that is a multiple of the page size. The physical memory is allocated out from the system memory pool and it has to be aligned to a page. The virtual memory is a contiguous memory block that is large enough and not devoted to global memory. The page state is initialised to be invalid so only the context it was mapped to has accessibility to the memory. If a private memory partition is created for the task, the commands in the libraries memPartLib and memLib can be used.

4.4.7 System calls

The system calls that manage the memory are placed in two different libraries, memPartLib and memLib.

memPartLib
Provides core facilities for managing the allocation of blocks of memory from memory partitions. The library consists of two sets of routines. The first set, *memPart...*(), comprises a general facility for the creation and management of memory partitions, and for the allocation and deallocation of blocks from those partitions. The second set provides a traditional ANSI-compatible *malloc*()/*free*() interface to the system memory partition.

Function	Description
MemPartCreate	Creates a new memory partition containing a specified memory pool. The descriptor for the new partition is allocated out of the system memory partition.
MemPartAddToPool	Adds memory to a specified memory partition already created. The memory added need not be contiguous with memory previously assigned to the partition.
MemPartAlignedAlloc	Allocates a buffer of wanted size from a specified partition. The allocated buffer begins on a memory address evenly divisible by a wanted alignment. The alignment must be a power of 2.
MemPartAlloc	Allocates a block of memory from a specified partition. The size of the block will be equal to or greater than the number of bytes to allocate.
MemPartFree	Returns a block of memory previously allocated. The memory is returned to the partition's free memory list.
MemAddToPool	Adds more memory to the system memory partition.
Malloc	Allocates a block of memory from the system memory partition.
Free	Returns a block of memory previously allocated with *malloc*() or c*alloc*(). The memory is returned to the system memory free memory list

memLib
Provides full-featured facilities for managing the allocation of blocks of memory from memory partitions. The library is an extension of memPartLib and provides enhanced memory management features, including error handling, aligned allocation, and ANSI allocation routines. The system memory partition is created when the kernel is initialised.

Function	Description
MemPartOptionsSet	Sets the debug options for a specified memory partition.
Memalign	Allocates a buffer of wanted size from the system memory partition. The allocated buffer begins on a memory address evenly divisible by the wanted alignment. The alignment must be a power of 2.
Valloc	Allocates a buffer of wanted size from the system memory partition. The allocated buffer begins on a page boundary.
MemPartRealloc	Changes the size of a specified partition of memory and returns a

	pointer to the new partition. The memory alignment of the new block is not guaranteed to be the same as the original block.
MemPartFindMax	Searches for the largest available block in the memory partition free list and returns it's size.
MemOptionsSet	Sets the debug options for the system memory partition.
Calloc	Allocates a block of memory for an array that contains a number of elements with a specific size.
Realloc	Changes the size of a specified block of memory and returns a pointer to the new block of memory. The contents that fit inside the new size (or old size if smaller) remain unchanged. The memory alignment of the new block is not guaranteed to be the same as the original block.
cfree	Returns a block of memory previously allocated with *calloc*(). The block of memory is returned to the system memory free list. The block is not allowed to be invalid.
MemFindMax	Searches for the largest available block in the system memory partition free list and returns its size.

4.4.8 System calls for VxVMI

The system calls that are used to manage virtual memory and text segment protection are placed in to libraries.

VmLib - architecture-independent virtual memory support library
The library provide an interface to the MMU that is independent of the architecture.

Function	**Description**
VmLibInit	Initialises the virtual memory support module.
VmGlobalMapInit	This routine creates and installs a virtual memory context with global mappings defined for each contiguous physical memory. The context ID returned becomes the current virtual memory context. A physical memory descriptor is passed as an argument. The descriptor contains state information used to initialise the state information in the MMUs translation table.
VmContextCreate	This routine creates a new virtual memory context. The newly created context does not become the current context until explicitly installed.
vmContextDelete	Deallocate the underlying translation table associated with a virtual memory context. It does not free the physical memory already mapped to the virtual memory space.
vmStateSet	Changes the state of a block of virtual memory.
vmStateGet	This routine extracts state bits. Returns ERROR if address to the page is not on a page boundary, the validity check fails, or if the virtual address doesn't match the architecture-dependent state.
vmMap	Maps physical pages into a contiguous block of virtual memory. The virtual and physical address must start on a page boundary, and the length must be a multiple of the page size. The state of all pages in the newly mapped virtual memory is valid, writeable, and cacheable.
vmGlobalMap	Maps physical pages to virtual space that is shared by all virtual memory contexts. To insure that the shared global mappings are included in all virtual memory contexts this call should be made before any virtual memory contexts are created. The state of all pages in the newly mapped virtual memory is unspecified and must be set.
vmGlobalInfoGet	Provides a description of those parts of the virtual memory space that is dedicated to global memory. Returns a pointer to an array where each element of the array corresponds to a block of virtual memory.
VmPageBlockSizeGet	Returns the size of a page block for the current architecture.
vmTranslate	Translate a virtual address to a physical address. This routine

	retrieves mapping information for a virtual address from the page translation tables.
*vmPageSizeGe*t	Returns the architecture-dependent page size.
*vmCurrentGe*t	This routine returns the current virtual memory context.
*vmCurrentSe*t	This routine installs a specified virtual memory context.
vmEnable	Turns virtual memory on and off. Memory management should not be turned off once it is turned on except in the case of system shutdown.
*vmTextProtec*t	This routine writeprotects the VxWorks text segment and sets a flag so that all text segments loaded will be write-protected.

vmShow – virtual memory show routines
This library contains virtual memory information display routines.

Function	**Description**
*VmShowIni*t	Include virtual memory show facility.
*vmContextSho*w	Displays the translation table for a specified context. It shows blocks of virtual memory with consecutive physical addresses and the state information.

5 Existing hardware support units (MMUs)

5.1 PowerPC - The 600 family

The PowerPC MMU support demand-paged virtual memory. It does so by using paged-segmentation. It also features a block address translation mechanism for supporting super pages. The PowerPC MMU supports the following three types of address translation :

- Paged-segmentation address translation. Memory is divided into 16, 256 Mb segments which is further divided into 4 Kb pages.
- Block address translation, translates the block number for blocks in the range of 128 Kb to 256 Mb.
- Real addressing mode address translation. The address translation is disabled and the physical address is the same as the effective address.

The architecture described in Figure 15 uses a 32-bit effective address. The information about the powerPC have been gathered form [3] and [4].

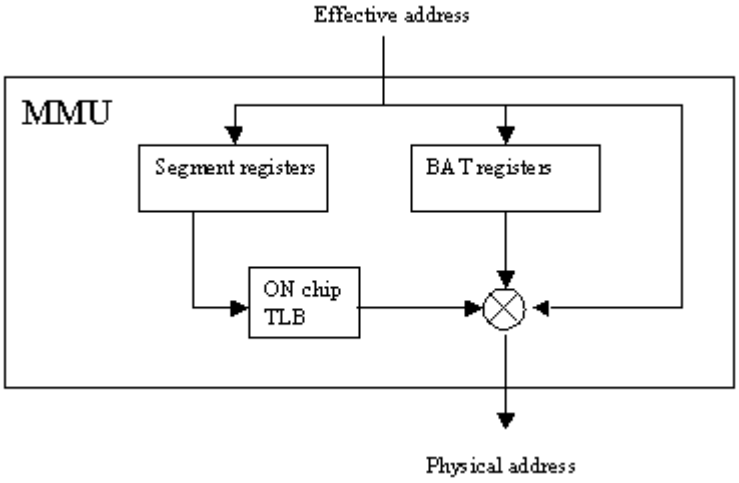


Figure 15 PowerPC MMU Architecture

Block address translation
The block address translation (BAT) operates in parallel with the paged segmentation translation. The BAT takes precedence over the paged-segmentation translation whenever a hit occur. The block size ranges from 128 Kb to 256 Mb. The block address translation has some resemblance with an unequal sized fixed partitioning scheme. The BAT array is implemented with 16 special purpose registers. Two registers make up one BAT array entry. Four entries are provided for instruction translation and four are used for data address translation. Each BAT array entry contains the start of the block in effective address space, the size of the block, the start of the block in physical memory and some control and fine-grained protection bits. As fine-grained protection a block can be made read only, read/write or no access. If an effective address is within the range defined by an BAT entry it is a BAT hit and a physical address is formed by concatenating the start of the block in physical memory with some low-order effective address bits, an offset.

The BAT doesn't provide much of address space protection. If a block is available to one user process it is shared between all processes. A block can be marked to be access only be privileged (kernel) access, although it might be assessable by the paged segmentation translation mechanism for user process.

Paged-segmentation address translation
With paged segmentation the effective memory address space is first divided into 16, 256 Mb, segments. The top four bits of the effective address is used to index a segment table. From the segment table a 24 bit segment identifier is obtained, which replace the four bits in the effective address to form a 52 bit virtual address. This virtual address is mapped to physical memory through a TLB and a page table structure. The architecture doesn't define a TLB structure but allow such to be implemented. If a TLB is implemented it should work in the following manner. The virtual address is compared with the TLB entries and in case of a TLB hit a 20-bit physical page number is acquired from the TLB and is concatenated with the 12 byte offset in the effective address to form the physical address. In case of a TLB miss a search through the page table is needed.

The page table is stored in a variant of an inverted page table. The page table contains a number of page table entry groups (PTEG's). Each PTEG contains eight page table entries (PTE). A PTE contains a virtual segment identifier field, a reference and a changed (dirty) bit, some fine-grain protection bits and a physical page number. A PTE can reside in one of two PTEG's, one which is the primary and the other is a secondary. A page table search is done by performing a hashing function on the virtual address forming a physical address for locating the primary PTEG. A linear search is then performed on the primary PTEG. If none PTE is found a secondary hashing function is performed to find the secondary PTEG. The secondary PTEG is then searched. The second hashing function is simply the one-complement of the first hash function value. If no PTE can be found in the second PTEG a page fault occurs. The hashing functions must be used when construction the page table accordingly.

The memory protection mechanism allows selectively granting read access, read/write access , and prohibiting access to memory areas based on a number of control criteria's. Access can be set to be allowed by only supervisory processes or both supervisory and user processes, i.e. a page can be read only for a user process and read/write for a supervisor process. Address space protection is provided through the segment registers. If two process have the same segment identifier in one of their segment register they share the virtual segment. Protection is enforced by the operating system to disallowing shared segments, and can allow sharing by overlapping segment identifiers.

The architecture supports large address spaces, a process could extend its address space by having the operating system move new values to its segment registers, thereby giving the process access to the whole 52-bit virtual space. No page replacement algorithm is defined by the architecture for replacing pages in the main memory so the operating system has a free choice. However the history recording bits (referenced and dirty bit) is provided for use by such an algorithm.

5.2 MIPS32 4K

The information about the processor have been gathered from the processor core family software user's manual, [31] and the processor core datasheet [32].

There are a couple of different types of the MIPS32 4K processor core. In this chapter the 4Kc will be studied. It contain a MMU that interacts between the execution core and the cache controller. The MMU receive a virtual address from the execution core and translates it to a physical address. The MMU in the MIPS32 4Kc processor core works with a TLB (Translation Lookaside Buffer). The TLB is managed by software. MIPS32 4K provides a 32 bit virtual address space.

5.2.1 Page Table

The pages can be given individually different sizes from 4K to 16M in multiples of 4. The page table contains all the mappings between the virtual and physical addresses. The page table is located in the memory and are accessed by software when an update of the TLB is required.

5.2.2 TLB

The TLB block consists of three different parts, the *Joint TLB* (JTLB)*, the Instruction TLB* (ITLB) and *Data TLB* (DTLB). The JTLB handles pages from 4 Kb to 16 Mb, while the other two only handles 4 Kb pages. Both the ITLB and the DTLB are managed by the hardware and are totally transparent to the software, i.e. the programmer only have knowledge of and access to make changes to the JTLB. If the wanted page is found a TLB hit is achieved otherwise there is a TLB miss. When there is a TLB hit in the JTLB, the page should be copied to the ITLB or the DTLB. Consideration for the page size in the JTLB must then be made, this is due to the limit of page size in the ITLB and DTLB. In the case of a TLB miss software responsible for refilling of the JTLB. The architecture of the TLB is shown in Figure 16

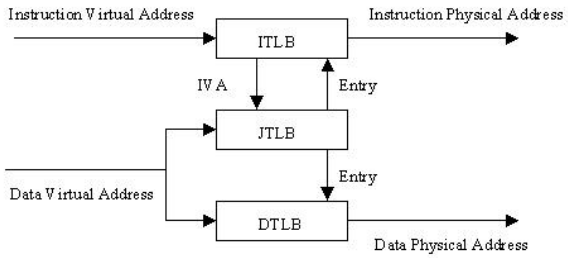


Figure 16 TLB architecture in MIPS

Instruction TLB (ITLB)
The ITLB maps 4 Kbytes pages and is used for the translation of the instruction stream. The ITLB are small and only contains 3 TLB entries. If an address cannot be translated by the ITLB the JTLB is accessed. If a TLB hit occur in the JTLB the wanted 4 Kbytes page is copied into the ITLB, and the ITLB is reaccessed to translate the addresses.

Data TLB (DTLB)
Like the ITLB the DTLB only maps 4 Kbytes pages and contains 3 entries. It is used for the translation of the Load/Store addresses. When a translation is requested both the DTLB and the JTLB is accessed, this reduces the miss penalty for the translation. If the JTLB translation produce a hit and the DTLB a miss, the wanted 4 Kbytes page is copied to the DTLB, and the DTLB is reaccessed.

Joint TLB (JTLB)
The JTLB is used to translate both instruction and data virtual addresses. The JTLB consist of 16 entries where each entry contains a tag entry and a data entry. Each tag entry correspond to two physical data entries, one even page entry and one odd page entry, i.e. the JTLB maps 32 physical page frames.

The JTLB tag entry contains the following parts:

- ASID: Address Space Identifier, Indicates which process the entry is associated with.
- Global bit: When set it indicates that all processes has access to the entry, this overrules the ASID.
- VPN2: Virtual Page Number divided by two. Bit 31:25 are always included in the comparison, while bit 24:13 are included depending on the page size.
- CMASK: Compressed page mask value. Indicates the size of the page, determines how many bits in the VPN2 that should be involved in the comparison. It also determines the bit that should be used for the odd/even page determination.

The eight bit ASID can identify 256 different processes. The ASID allows multiple processes to appear in the TLB and cache, i.e. when a context switch between processes there are no need for flushing the TLB or cache.

The corresponding data entry, odd or even, has the following parts:

- PFN: Physical Frame Number. Defines the higher bits in the physical address, only a few of the bits are used for pages larger then 4 Kbytes.
- C: Determines the cacheability attributes.
- D: Dirty or write enable bit.
- V: Valid bit. If set it indicates that the mapping is valid.

5.2.3 Virtual to physical address translation

When an address translation is requested the virtual address is compared with the virtual addresses in the TLB entries. All the entries are checked at the same time and a match occur when the virtual addresses are the same and one of the following condition are true. The global bit is set or the ASID field is the same in the virtual address and the TLB entry. If there where a hit the physical frame number are concatenated with the offset to form the physical address. The offset are not passed through the TLB.

TLB miss
If a TLB miss occurs the software refills the JTLB with the correct page from the page table resident in the memory. Selection of the TLB entry where the new mapping should be written can be done in two ways. Either the software writes over a selected TLB entry or a built in hardware mechanism is used. The hardware mechanism use a register that randomly selects the entry. The register decrease the value at almost every cycle and wrapping to the highest value when it reaches the value in the *wired register*. Through the wired register there are possible to protect a TLB entry from replacement.

When a entry should be updated by the software some register must be updated. The register contains information that will be part of the new JTLB tag and data entry.

6 Hardware memory management

6.1 System-on-chip dynamic memory management unit (SoCDMMU)

In this chapter a review carried out on the *system on chip dynamic memory management unit* by Mohamed Shalan and Vincent J. Mooney III at Georgia institute of technology. The information has been gathered from their paper about the SoCDMMU [16]. All the figures is also taken from the same paper.

6.1.1 Basic ideas

The system-on-chip contains a *global on-chip memory*, the *SoCDMMU* and a number of *process element* (PE), which can be a DSP, micro controller, microprocessor or a specific hardware. The SoCDMMU dynamically allocates memory from the global on-chip memory when a PE needs more memory. When the SoCDMMU has allocated a portion of memory for a PE, its the PE that manages the use of the memory. In [16] this type of memory management is called *Two-level Memory Management.*

Block and pages
The global on-chip memory is divided into a fixed number of blocks with the same size. Each memory block get a unique *Physical memory block number,* i.e. a block can only have one physical address. Each PE in the system has its own address space with blocks of the same size as the blocks in the global on-chip memory. The blocks in a PE also get a unique number, *Virtual block number.* The two types of block numbers are used for mapping virtual addresses to physical addresses with a *Address Converter*. Each memory block can be accessed by one or more PE's, therefor a memory block can have many virtual addresses but only one physical address.

A memory page contains one or more memory blocks. Each page can be in three different modes depending on what command the PE send to the SoCDMMU when the memory is allocated.

- *Exclusive Memory.* The PE that allocates the memory owns it and no other PE's can access the page.
- *Read/Write.* Only the PE that allocates the page can write to the memory, it can also read from the memory. No other PE can allocate the memory as *Exclusive or Read/Write.* If another PE wanted to read from the memory it must allocate the page as *Read Only.*
- *Read Only.* The PE can only read from the page, and a different PE must allocate the page as *Read/Write*

Interface on the chip
Figure 17 show the interface between the PE's, SoCDMMU and the global on-chip memory. Each PE has its own bus to the SoCDMMU, this should improve the performance compared with an architecture with only one bus connecting the PE's to the SoCDMMU according to Shalan and Mooney.

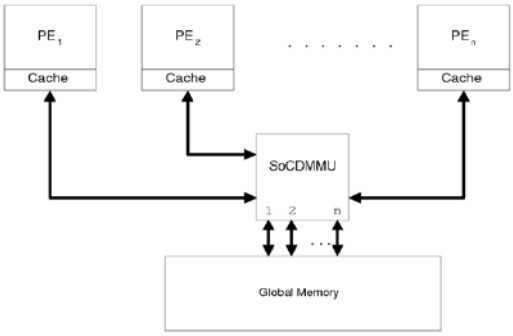


Figure 17 Interface on the chip

The Memory bus from a PE is connected to the SoCDMMU. This make the global memory access totally controlled by the SoCDMMU, i.e. all the mapping between physical and virtual addresses are handled by the SoCDMMU.

6.1.2 SoCDMMU architecture

The SoCDMMU contains of building blocks that together manages the global memory on the chip, Figure 18

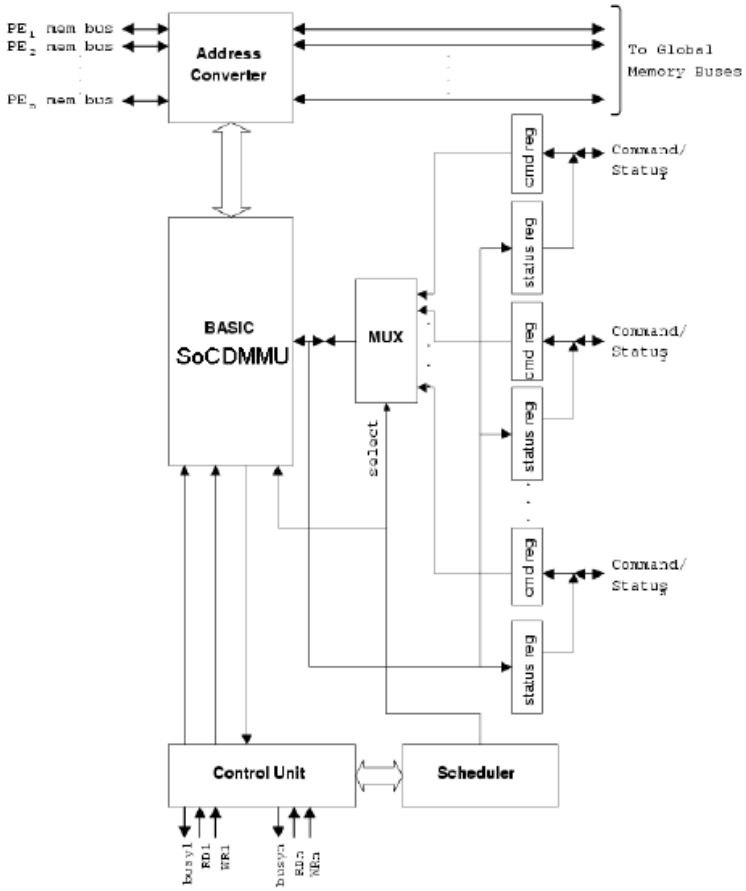


Figure 18 SoCDMMU architecture.

Address Converter
The address converter translates the virtual PE address to a physical address. The PE and physical addresses are stored in a *Lookup Table,* and each PE has its own lookup table.

The address from the PE consists of two parts, the virtual block number and a offset. The virtual block number is used to find the physical block number, using the lookup table for the current PE. The physical block number is then connected to the offset, obtaining the actual address to the global memory.

Command and status register
Each PE has its own status register and command register. This makes it possible to serve simultaneous requests from the PE's. The PE write their commands into the command register, and a scheduler organises the command in the right order.

Scheduler
The scheduler decides which command from a PE that should be executed when multiple request arrives to the *SoCDMMU*. The scheduler uses a round robin algorithm so the *basic SoCDMMU* is fairly shared between the PE's. According to [16] the priorities are dynamically assigned, the priorities ensures that the commands for deallocation is executed first.

Basic SoCDMMU architecture
The basic SoCDMMU have the architecture illustrated in Figure 19, the construction are the same no matter how many process elements there are connected to the SoCDMMU. Depending on the number of blocks that the global on-chip memory is divided into, changes in the size of the blocks, *allocation vector*, *deallocation unit*, *allocation unit* and *allocation table* must be made.

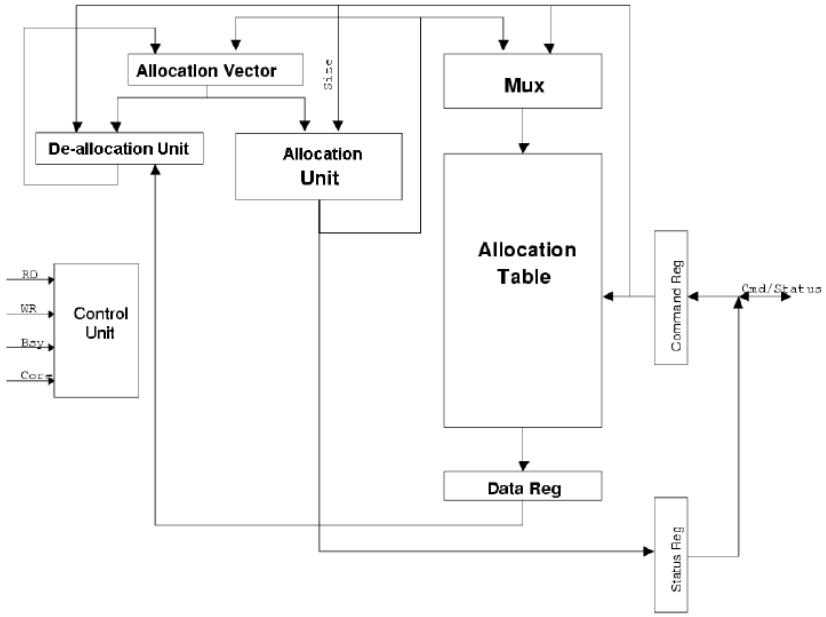


Figure 19 Basic SoCDMMU

Allocation Vector
To keep track of the free blocks in the global on-chip memory an A*llocation Vector* is used. The allocation vector is a bit vector where '1' indicates that the block in the global memory is allocated, and '0' indicates that the block is available. The number of bits in the allocation vector equals the number of blocks in the global memory.

Allocation Table
The basic SoCDMMU also use a *Allocation Table* to store the information about the allocated blocks. The allocation table word contain information about the following.

- *Size.* The number of blocks connected to the page.
- *PE.* Information about which PE that allocated the memory.
- *Mode.* Indicates if the memory is, Exclusive, Read/Write or Read.
- *SW ID.* The software ID that are assigned to the page.

Each block in the global on-chip memory have a corresponding word in the allocation table, i.e. the number of words equals the number of blocks.

Allocation Unit

The actual allocation of the memory is performed by a *allocation unit.* Inputs to the allocation unit is the number of blocks in the wanted page and the information about free blocks in the allocation vector. To allocate the requested page the allocation unit uses a first fit algorithm. With the output from the allocation unit the allocation table and allocation vector are updated. The output is also written to the status register. If an error is detected no update of the allocation table and allocation vector are performed, and an error code is written to the status register. The Allocation unit is also responsible for the update of the *Address Converter* Lookup Table.

Deallocation unit

To deallocate a memory page a *Deallocation Unit* is used. The page information from the allocation table and the allocation vector are inputs to the deallocation unit. If no error was detected the unit updates the allocation vector and deletes the page entry from the allocation table. When a deallocation is carried out a change in the address converter lookup table is also made.

6.1.3 Commands for the SoCDMMU

There are five different commands that the SoCDMMU work with:

- *G_alloc_ex.* Allocation of a page with exclusive mode. Need the virtual block number to the first block in the page and the number of blocks to be allocated.
- *G_alloc_rw.* Allocation of a page with read/write mode. Need the software ID to the page, the virtual block number to the first block in the page, and the number of blocks to be allocated.
- *G_alloc_ro.* Allocation of a page with read mode. Need the software ID to the page and the virtual block number to the first block in the page.
- *G_de-alloc* . Deallocate the wanted page. Need the first virtual block number in the page.
- *Move.* Re maps allocated memory blocks to another PE address. Need the old and new virtual block numbers

There are a couple of errors that can occur and be indicated in the status registers, during the commands on the SoCDMMU level, they are as follow.

- Allocation. Not enough memory to allocate.
- Deallocation. Trying to deallocate memory page that belongs to another PE, or trying to deallocate a non exiting memory page.
- Move. Moving a non existing memory block.

6.1.4 Performance

In [16] a comparison has been done between the hardware SoCDMMU and a software SoCDMMU, running on a *Microchip PIC, RISC microcontroller.* The software SoCDMMU have the same functionality as the hardware SoCDMMU. The clock rate in both cases were assumed to be 400 MHz. Table 1 shows the outcome of the comparison.

SoCDMMU, worst case execution time	20 Cycles
Micro controller, best case execution time	221 Cycles

Table 1 Comparison between the SoCDMMU and the micro controller

6.1.5 Conclusion and Advantages

The SoCDMMU allows a fast allocation and deallocation of the global on-chip memory. The management of the memory is completely deterministic and therefor its useful for real-time systems. A disadvantage is when a new PE should be implemented in the system, or the number of memory blocks are changed. There must be changes made in some building blocks in the SoCDMMU itself. Examples of blocks that must be changed is the, *address converter*, *Control unit* and the *Scheduler*, perhaps this could be cured with generic implementation of the blocks.

6.2 A high performance memory allocator for object oriented systems

J Morris Chang and Edward F Gehringer have developed an allocator implemented in hardware, presented in [18], most of the pictures here are copied from the paper. The base of the allocator is a modified buddy system. The system allocates memory sizes that are power of two, between a minimal size and a maximum size. A bit-map is used to mark which blocks of the minimal size that have been allocated. This bit map forms the base for a hardware binary tree, which is used to find free blocks for allocation. The binary tree itself represent the buddy system. Deallocation is made by resetting bits in the bitmap, making the need for coalescing obsolete since contiguous memory will be automatically coalesced in the bit-map. In Figure 20 , coalescing in a software in a buddy system is compared to the proposed hardware approach. Coalescing of two blocks of size 4 is made. With the software approach two free list pointers need to be changed. However with the hardware approach the bits that are freed are combined with its adjacent bits in the bit-map. Thereby coalescing is made by default when freeing bits in the bit-map, without any extra logic.

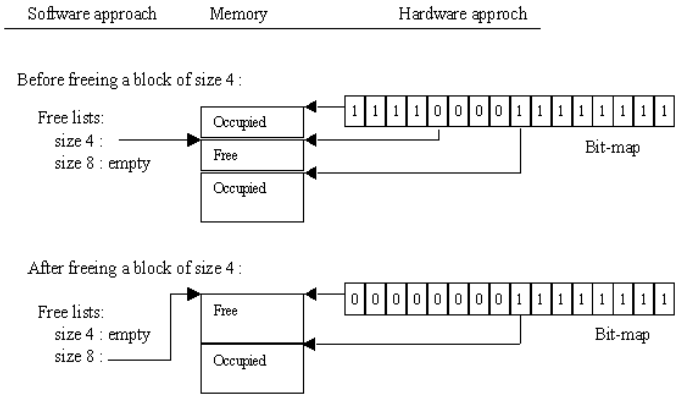


Figure 20 Example of coalescing in software buddy system and hardware buddy system using a bit-map

Also the internal fragmentation problem with the buddy system can be reduced by letting the hardware only set the used bits in a buddy, instead of the whole buddy as in an ordinary buddy system. This is called a modified buddy system. The hardware modified buddy system allocates space in three steps :

1. First it determines if there is a large enough space to serve the allocation request.
2. If so, it finds the start of the free memory space.
3. And finally it marks the corresponding bits in the bit-map.

Step 1 is performed by an or-gate binary tree, step 2 is done by an and-gate binary tree. A bit-map marking is implemented by a bit-flipper. Deallocation is simpler. A bit-flipper just resets the bits in the bit-map associated with the address-space to be deallocated.

6.2.1 Locating free memory : The or-gate tree

To determine if there is a space large enough to serve an allocation request, a complete binary tree is built on top of the bit-map. The tree consists of or-gates with two inputs as nodes and the bits in the bit-map as leafs. Figure 21 illustrates the or-gate tree. The output from a node shows the allocation status of the subtree below, 0 representing free and 1 allocated. If the output at the root is 0 the whole memory is free. The system serves allocation requests of sizes represented by the levels in the tree. For a certain size request, only the outputs of the corresponding level need to be considered. By making an AND operation of the outputs at this level, it could be determined if there is any available memory of the corresponding size, 0 there is and 1 there isn't. The or tree will require 2^N-1 or-gates to handle a allocation space of 2^N blocks of minimum size.

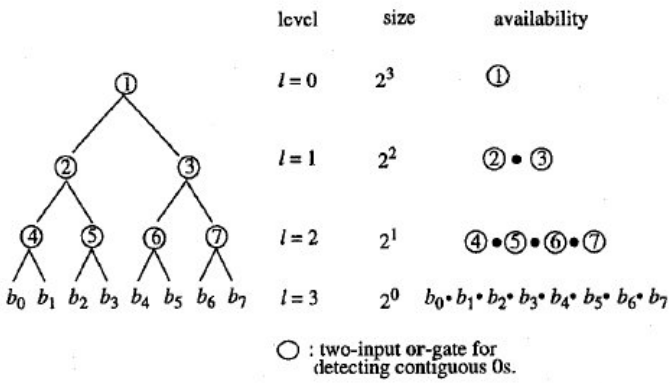


Figure 21 Example of a or-gate tree

For example Figure 21 shows an or-gate tree for a eight bit bit-map. The output of a node at level *l* represent the allocation status of a buddy of size 2^{N-l} blocks. Where 2^N is the number of blocks that can be allocated

6.2.2 Finding the address of the free memory : the and-gate tree

To find the address of a free space found at level *l*, the first zero of level *l* (in the or tree) need to be found. This is done by feeding an and-gate tree with the or-gate outputs from level *l*. If the address of the first zero of level *l* in the or-gate tree can be determined to be *n*. Then the address of the of the block to be allocated is $n*2^{N-l}$ (eq 6.1).

The and-gate output at each node is called propagation bit (P-bit) since it propagates allocation status upwards trough the tree. To search for the first zero valued P-bit at a level would require backtracking in the tree. To avoid backtracking a extra address-bit (A-bit) is used to guide the search. This A-bit is a copy of the P-bit of a nodes left child. An A-bit value of one guide the search to the right and zero to the left. For example as shown in Figure 22, the A-bit in the root node has value 0 thereby indicating that the next step is to the left child node. At node 2 the A-bit is 1, indicating a next step to the right, node 5. The A-bit thereby directs the search to a subnode that have sufficient free space. The sequence of A-bits during the traversal from the root through the tree forms *n* in eq 6.1, thereby forming the address to the block to be allocated. It should be observed that the input to the and-gate tree is not the bit map.

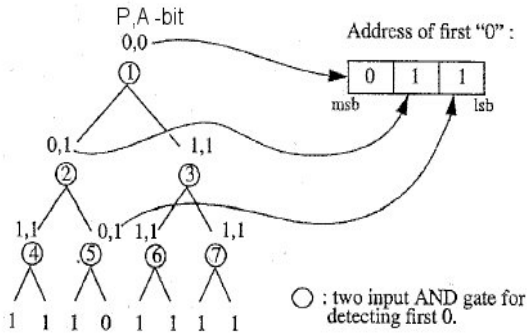


Figure 22 Searching for the first "0" in he and-gate tree

There is a logical connection between the or-gate and the and-gate tree shown Figure 23.This make it possible to have only one and-gate tree instead of having one for each level.

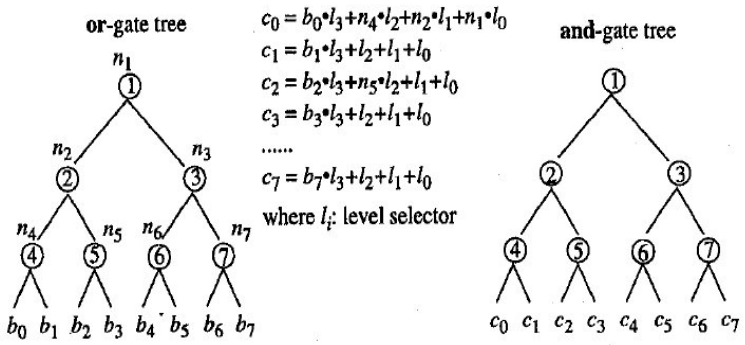


Figure 23 The connection between the or-gate and the and-gate binary tree

6.2.3 Marking allocated memory: the Bit-flipper

To perform step 3 in the allocation process Chang and Gehringer purpose a bit-flipper mechanism to flip bits in the bit-map, from 0 to 1 if allocation and from 1 to 0 if deallocation. As input it takes the start address of the bits to be flipped and the number of bits to flip. The modified buddy system only marks as many bits necessary to fulfil the allocation request, not a whole buddy as in a regular buddy system. The bit flipper is a binary tree in which the signal propagates from the root to the leafs, the leaf output tells which bits in the bitmap to invert. Each node in the tree have four inputs and four outputs as shown in Figure 24.

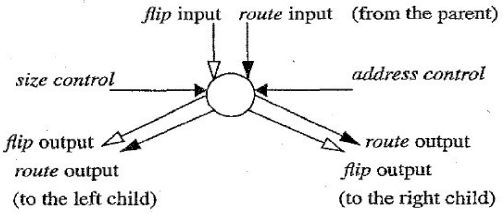


Figure 24 A bit-flipper node

The flip input has the highest priority, when set (=1) the whole subtree of nodes should have their flip output set. If the flip signal is not set, and the route input signal is set. The subtree will be partially allocated and flipping will depend on the size and address control bit. If none of the flip or route input signals is set no flipping will occur in the subtree. The address control signal directs output to be set to the left or right child node. If the size control signal is set the node flip output should be set according to the address control signal. This is better explained by looking at the example of Figure 25. The flip input to the root must always be cleared, the route input to the root node should be set. A leaf flip output of 1 means that the corresponding bit in the bit-map should be inverted. The truth table for a bit-flipper node is shown below.

Input				Output			
Flip input	Route input	Size control	Address control	Flip left	Route left	Flip right	Route right
1	X	X	X	1	X	1	X
0	0	X	X	0	0	0	0
0	1	0	0	0	1	0	0
0	1	0	1	0	0	0	1
0	1	1	0	1	X	0	1
0	1	1	1	0	0	1	X

X means don't care

Table 2 The truth table for the bit-flipper

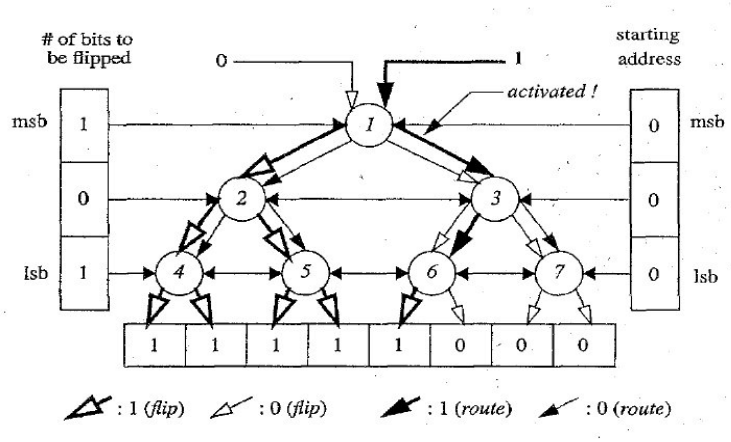


Figure 25 Example of flipping, when five block is to be allocated at address 0

6.2.4 Conclusion

The main advantages of the presented method is that allocation is made in constant time and probably a lot faster then a corresponding software approach. In the general case the hardware allocator is also said to suffers less from internal fragmentation then a ordinary buddy system. On the downside it might not scale very well to large memories due to the rapid growth of the hardware needed.

The method presented here has been expanded into a dynamic memory management unit [18], which support garbage collection. Also there has been a realloc implementation in hardware that could be combined with this allocator [19].

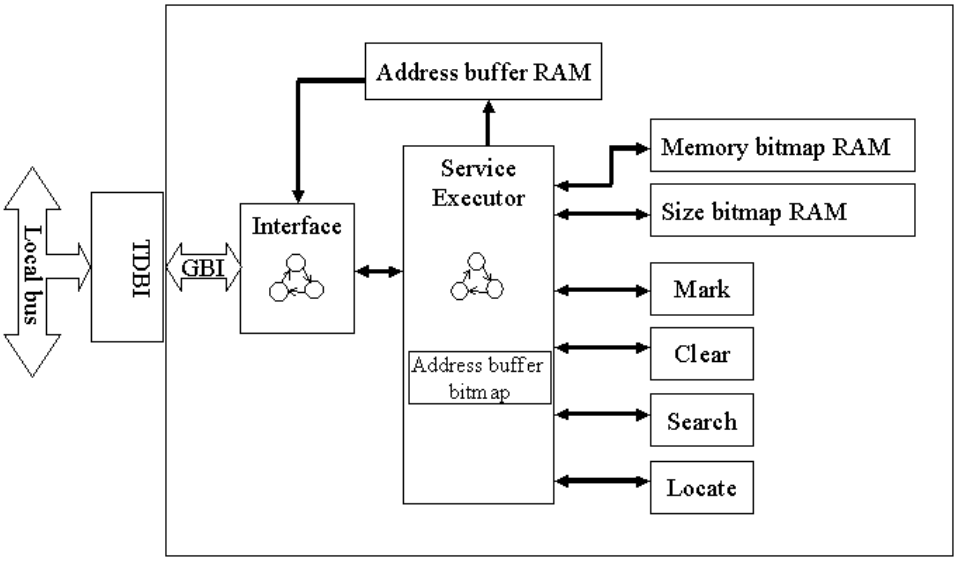
7 Conclusions

This paper will serve as the foundation for implementing a hardware memory allocator. The reason for implementing a hardware memory allocator is to gain determinism and speed compared with a software approach.

8 Bibliography

[1] *VRTX32/68K Real – Time Kernel, Programmers Guide and Reference*, Microtec
[2] *Tradeoffs in supporting two page sizes*, M. Talluri, S. Kong, M.D. Hill, and D.A Patterson, in Proc. 19th Annual international Symposium on Computer Architecture(ISCA-19), May 1992
[3] *Virtual memory in contemporary microprocessors*, Bruce Jacob and Trevor Mudge, IEEE Micro, Vol. 18, No. 4, July/August 1998
[4] *PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors*, IBM Microelectronics Division, G522-0290-01, 21, Februari 2000
[5] *An overview of Memory Protection under eCos.* www.3glab.org
[6] *Dynamic storage allocation: A survey and critical review*, P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. In Proceedings of the 1995 International Workshop on Memory Management, volume 986 of Lecture Notes in Computer Science, Kinross, United Kingdom, Sept. 1995. Springer-Verlag.
[7] *eCos 1.3.1 Documentation*, Red Hat, Inc, http://www.redhat.com/.
[8] *Operating systems: Internals and design principle, 3ed*, William Stallings, ISBN 0-13-887407-7, 1998, Prentice Hall.
[9] O*perating system concepts*,4ed, A. Silberschatz and P.B Galvin, ISBN 0-201-50480-4, 1994, Addison-Wesley.
[10] VxWorks programmers guide 5.4, edition 1, Windriver systems, http://www.windriver.com.
[11] VxWorks reference manual 5.4, edition 1, Windriver systems, http://www.windriver.com.
[12] Tornado user's guide 2.0, edition 1, Windriver systems, http://www.windriver.com.
[13] OSE kernel, Reference manual R 1.0.4, Document No: 420e/OSE93-1, Enea OSE Systems AB, http://www.ose.com.
[14] OSE kernel, User's guide R 1.0.4, Document No: 420e/OSE93-2, Enea OSE Systems AB, http://www.ose.com.
[15] OSE Memory Management System (MMS), User's guide and reference manual R 1.0.5, Document No: 420e/OSE75-7, Enea OSE Systems AB, http://www.ose.com.
[16] A Dynamic Memory Management Unit for embedded real-time System-on-a-Chip, Mohamed Shalan, Vincent J Mooney III, Georgia Institute of technology.
[17] *A High-Performance Memory Allocator for Object-Oriented Systems*, J. Morris Chang, Edward F. Gehringer, IEEE Transactions on Computers, March 1996 (Vol. 45, No. 3), pp. 357-366.
[18] *Architectural support for dynamic memory management,* J. Morris Chang, Srisa-an, Chia-Tien Dan Lo, *Proceedings of IEEE International Conference on Computer Design*, Austin, Texas, Sep. 17-20. 2000, pp.99-104.
[19] *A Hardware Implementation of Realloc function*, J. Morris Chang, Srisa-an, Chia-Tien Dan Lo, VLSI '99. Proceedings. IEEE Computer Society Workshop On , 1999, Page(s): 106 -111
[20] *Modern operating systems*, Andrew S. Tanenbaum,ISBN 0-13-595752-4, 1992, Prentice-Hall Inc.
[21] *The Art of computer programming, Vol1, Fundamental algorithms,2ed*, Donald E Kntuh, *ISBN* 0-201-89683-4, Reading, Addison-Wesley.
[22] *An analysis of the effects of memory allocation policy on storage fragmentation*, Michael Shannon Neely, citeseer.nj.nec.com/58412.html, 1996.
[23] *Programming languages – C*, ISO/IEC 9899:1999
[24] Fast-Fit: A new hierarchical dynamic storage technique, Masters thesis, UC Irvine, Computer Science department 1978
[25] Data structure techniques, Thomas Standish, Addison-Wesley, Readin, Massachusetts, 1980
[26] The Memory Management Reference, http://www.memorymanagement.org/
[27] *Design of a general-purpose memory allocator for the 4.3 BSD UNIX kernel*, Marshall Kirk McKusick and Michael J. Karels, In Proceedings of the Summer 1988 USENIX Conference, San Francisco, California, June 1988. USENIX Association
[28] *OSE – Heap Manager Reference*, Document no : 420/OSE116-1R1.0.2, Enea OSE Systems AB
[29] *VRTX Real-Time Operating System, Product Brochures*, Mentor Graphics Corp, www.mentor.com
[30] *VRTX 5.0 user guide*, Doc Id: 102456-001, 12/1998, Mentor Graphics Corp, www.mentor.com
[31] *MIPS32 4K processor core family software user's manual, revision 1.12*, MIPS Technologies inc, www.mips.com
[32] *MIPS32 4Kc processor core datasheet, revision 01.03*, June 2000, MIPS Technologies inc, www.mips.com

A Dynamic Memory Allocator in hardware



Authors :

Karl Ingström
E-mail:kim99001@student.mdh.se
Phone: (+46)070-5586023

Anders Daleby
E-mail:ady99002@student.mdh.se
Phone: (+46)021-129884

Karl Ingström
kim99001@student.mdh.se

Anders Daleby
ady99002@student.mdh.se

Abstract

Software allocation is a time consuming and usually unpredictable task. Development of larger chips makes it possible to implement more and more software functions in hardware there by increasing execution speed.

In this thesis we present a novel approach for performing memory allocation and deallocation in hardware, a dynamic memory allocator, DMA. The DMA uses a ***pre-search*** *approach (before allocation request) for finding free objects. The pre-search will reduce the response time. A predictable minimum timing restriction between two service requests to the DMA is introduced, which guarantees a pre search time for the DMA. The dynamic memory is represented by a bitmap.*

The DMA has been implemented in hardware using VHDL. The implementation is generic and can be mapped to different environments.

The DMA is shown to reduce allocation time, compared to a software allocator, in a test application. The allocation response time is fast, less the 15 clock cycles. Resulting in significant speed up for the application, up to 90 % compared to the software allocator. The DMA has predictable behaviour, allocation always take the same amount of time, making it useful in real time systems.

Keywords

fast predictable generic IP hardware dynamic memory allocation deallocation FPGA SoC

Karl Ingström
kim99001@student.mdh.se

Anders Daleby
ady99002@student.mdh.se

Karl Ingström
kim99001@student.mdh.se

Anders Daleby
ady99002@student.mdh.se

Table of contents

Karl Ingström
kim99001@student.mdh.se

Anders Daleby
ady99002@student.mdh.se

1 Introduction

In this report we develop an Intellectual property (IP) module, called DMA, for performing dynamic memory allocations in hardware. The memory allocator is able to carry out two types of services, allocation and deallocation. The module shall have a well defined architecture that can be scaled by generic parameters to serve a broad range of memory sizes, block sizes etc. The performance of the module will be evaluated against a comparable software allocator.

This report is a second part of our master degree project, performed at the Department of Electronics ,Mälardalens University, Västerås, Sweden. The project was initiated and conducted in co-operation with RealFast AB, www.realfast.se and The Department for Computer Science and Engineering Mälardalens University.

The first part of the project was a state of the art report which gives an overview of memory management [18]. Focus was on memory management in real-time operating systems .

Structure of report
The remaining of this section gives the motivation for developing a dynamic memory allocator in hardware. Section 2 will give background to memory allocations in software and to previous work done in the field of hardware allocators. Section 3 contains our analysis of the problem, the section analysis which hardware features that can be exploited in an hardware allocator. In section 4 our approach for a hardware allocator is presented and how the unit should be used. Section 5 introduces the DMA architecture, our developed dynamic hardware allocator. Section 6 will go into depth with the DMA implementation. Section 7 defines a test system for compression between a software allocator and the DMA, The software allocator used is also presented here. Section 8 gives the measured results. Section 9 concludes by giving a discussion around the results. It also states future work that can be done to enhance the DMA.

1.1 Motivation

In a report from Gartner [7] it stated that the market for third-party intellectual property is extremely important to the semiconductor industry. The market for System-level integration (SLI) or system-on-a-chip (SoC) devices was about $20 billion in 2000, and it is expected to grow to $60 billion by 2004.

As chip sizes grow large the problem is not how to make large chips, but rather what to incorporate on them. As functions performed in hardware generally are fast, hardware could be used to remove system bottlenecks introduced by software. This follows the well known *Amdahl's Law – make the common case fast* [2]. Amdahl's law tells us that the benefit of an improvement is affected by how much the improvement is used. Thus, making the common case fast will tend to enhance performance better then optimising rare cases. A lot of research points at dynamic memory allocations being a rather common case. Measurements on real programs made by B. Zorn el. at [15][4][5] shows that allocation time corresponds to somewhere between 1 –40 % of the total runtime of the measured applications. As the usage of object oriented programming will increase. The amount of dynamic allocations will increase, up to 10 times [6], making allocation even a more common case.

Allocation / deallocation performed by hardware can be used in a type of device called IRAM (Intelligent RAM). According to D. Patterson et al [8] this type of devices which integrate main memory on the same chip as the processor, might be necessary to bridge the CPU main memory gap. Research in this direction are currently taking place [19].

Lots of research is aimed at using high-level languages, e.g. ANSI C, as input for hardware designs. As shown in [3] a hardware allocator can be used to efficiently map C/C++ models (malloc / free) into hardware. The design process would be accelerated by the use of C/C++ or a subset of C/C++ to describe both hardware and software. The designers could describe the system in C. Compilers and synthesis tools could then do partitioning of the system into hardware and software blocks [9]. Such a development would be aided by the possibility to use a hardware allocator.

Karl Ingström
kim99001@student.mdh.se
Anders Daleby
ady99002@student.mdh.se

It's a well know problem that dynamic allocation in software is slow and non-deterministic. A software allocator usually involves a searching in a heap area. The searching is often performed in a sequential fashion (i.e. linked list search). As the number of existing objects to be searched grows, the search time will grow linearly longer as well. This limits the use of dynamic memory allocation in real-time applications where deterministic behavior, i.e. predictable execution time, is wanted. As it is believed, and shown, that a hardware implementation of a dynamic memory allocator can achieve predictable allocation time [12][13]. Such an allocator could enhance the range of real-time applications, i.e. make new types of real-time implementations possible.

2 Background

Memory allocation techniques has been implemented in software for a long time. This chapter will describe the work a memory allocator should perform and how this usually is done in software. Resent development and research have migrated dynamic memory allocation from software to hardware. Two such hardware allocator will be presented.

2.1 The purpose of a memory allocator

The purpose of a memory allocator is to supply an application with the service of allocation and deallocation of memory from a heap. That is supply utilities for an application to request an amount of memory to be used by the application. And a way for the application to return the memory to the heap when it no longer needs it.

The allocator must register which part of the memory that is used (allocated by a application) and free (available for allocations). The goal of a memory allocator is to minimise memory waste inside reasonable time frame. The ideal memory allocator is one that would spend negligible time managing the memory and cause negligible waste of memory space.

An allocator has not any control over the number, nor the size, of objects being allocated. This is determined by the application(s) that utilises the allocator. Neither has the allocator any control of when an allocated object is going to be returned to the heap (deallocated), or the order of which allocated objects are deallocated. The problem introduced is that holes emerge in the heap. These holes can be to small to individual serve allocation requests but together they could sum up to a large amount of memory. This is called fragmentation.

2.2 Related work

The existing number of software allocators developed through the years is out of the scope of this report to present. The software allocator chapter will give pointers where to find information on software allocators. The two hardware allocators described in this chapter are the :

- SoCDMMU by Mohamed Shalan and Vincent J. Mooney III [12].
- DMMX by J. Morris Chang, Edward F. Gehringer [13], [16].

2.2.1 Software allocators

There exists a large variety of software allocators which all have there advantages and disadvantages. A good survey of the topic is made by Paul Wilson, Mark Johnstone, Michel Neely and David Boles, Dynamic Storage and Critical Review [14]. Also in [18] some basic allocator mechanisms are described. Some often used basic allocator mechanism is first fit, best fit and buddy systems.

There will always be some application that will cause sever fragmentation for any allocator [14]. In practice there has been shown [10] that fragmentation is really a problem of a poor allocator implementation. For most applications there exists well known allocator principles which causes no severe fragmentation in the general case. Good policies are for example address-ordered first fit and best fit. For most allocators, the problem of simple overheads is more significant than the problem of fragmentation itself.

2.2.2 Hardware allocators

The previous work on performing dynamic allocations with hardware have focused on performing allocations and deallocations in constant time. Examples of such allocators are the, SoCDMMU by Mohamed Shalan and Vincent J. Mooney III [12] and the DMMX by J. Morris Chang, Edward F. Gehringer [13], [16].

The main principle used in both allocators are the use of a bitmap for representing the memory that can be allocated. The use of a bitmap removes the problem of coalescing [18] as it will be carried out automatically as seen in Figure 1. Coalescing is the merging of adjacent free objects when performing deallocation.. An allocator that immediately coalesce memory has been shown to perform better, concerning fragmentation, then one that doesn't [10].

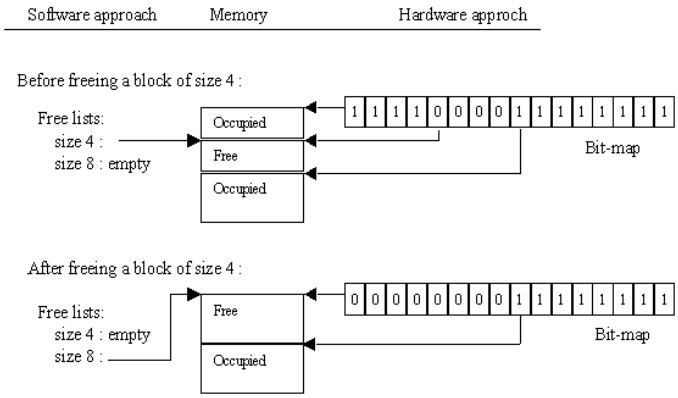


Figure 1: Automatic coalescing with bitmap representation

A bitmap representation is one of the smallest representation there is regarding how much information that must be stored [14].

Another feature in common between the two allocators is that all job is performed after a command is given to the allocator. The allocator search for a suitable free object in the bitmap and then returns an address to a free object if one is found. This means that the allocator will be inactive most of the time, as allocation time is usually some where between 5 – 30 % of application run time in allocation intensive c-programs [15].

System on Chip Dynamic Memory Management Unit - SoCDMMU
The SocDMMU [12] is a hardware allocator developed to perform dynamic memory allocations in a multi-processor environment. The SocDMMU is intended to handle requests of large blocks of memory which will be assigned to the requesting processor. A processor can then allocate smaller amount of memory from the larger allocated block by the use of a software allocator.

The SocDMMU allocator uses a first fit algorithm for performing memory allocations. The allocation procedure is made by combinatorial logic and takes just one clock cycle. A bitmap is used to represent the memory that can be dynamic allocated. The size of allocated objects is stored in a table. The table also holds information of which process element that did the allocation , which mode the object was allocated to (exclusive, read / write, read) and a software ID.

The main principle of the allocator is to have a counter logic associated with each bit in the bitmap (a bit representing a block of memory that can be allocated). A search is then performed by counting number of free blocks, a sequence of bits having value zero. If an allocated bit is encountered the counter is cleared. The counting progress until a free space of sufficient size is found. The address to the free space is then returned. In the case of 256 possible blocks that can be allocated and the maximum size that can be allocated is the whole space. Each logic unit will have a counter of 8 bits and there will be 256 of these counters after each other. This will give quite a long data path.

An implementation using Synopsys Design Compiler™ and the AMI 0.5-micron Logic library.
With 256 bits in the bitmap where each bits represents 64 Kbytes, and support for 4 processors. Requires a equivalent gate area of 41,561.5 gates [12].

Dynamic memory management extension - DMMX
The DMMX [16] is a hardware allocator / garbage collector developed for use in a single processor environment. The DMMX allocation scheme is a modified buddy system. The modification is that the hardware marks only the bits that are being allocated instead of whole buddies. In most cases this should lead to less internal fragmentation compared to an ordinary buddy system (there are cases where the fragmentation is worse). A bitmap is used for representation of the memory that can be allocated. The size of allocated objects is also stored in a bitmap.

The principle for the allocator part of the DMMX [13] is divided into three steps

a) Determine if there is a large enough free space to serve the request and if so
b) Find the address of the memory suitable for serving the request and
c) Mark the bits in the bitmap as allocated.

All three steps is performed by combinatorial logic, that is a allocation request can be served in just one clock cycle. The combinatorial logic is built in a tree structure which should give quite a good scalability.

Deallocation of allocated objects is described in [17].

The allocator has been implemented in a Xilinx Vertex V800HQ246-6 device for 256 and 512 blocks [22]. In this article the allocator is called AMMU (Active memory management unit). The implementation used 14.594 and 28.786 gates. The max frequency of the implementation was 9.649 MHz and 7.992 MHz respectively.

3 Analysis

The section analysis which hardware features that can be exploited in an hardware allocator. Example of such a feature are Parallelism. The chapter starts with setting up some restrictions on the allocator that shall be developed.

3.1 Restrictions

There where some design goals / restrictions set up for this project. The hardware allocator should:

- Have a fast response time compared to a software allocator.
- Use a small chip area
- Be as customisable as possible regarding memory space, object sizes, etc.
- The allocator is manly targeted for a small range of object sizes. That is the number of different object sizes that can be allocated is quite limited (1 –16).
- Be transparent for the user.
- Be aimed at single processor system, maybe adaptable to multiprocessor system.
- Use no memory protection, i.e. all allocated memory is shared.

3.2 Definition of the job to be performed

Allocation
A typical allocation service can be broken down into the following steps :

1. A search step for finding a suitable free object to serve the request.
2. Mark such an free object as allocated.
3. Return the address to the found object.

Deallocation
A typical deallocation service can be broken down into the following steps :

1. Determine if the object to be deallocated is an valid object, i.e. is an allocated object.
2. Change the status of the allocated object to being free.

3.3 Hardware features to be exploited

When constructing in hardware there are several hardware features that might be used to increase speed.

- Internal representation of memory space (in the hardware allocator) → faster search times as memory access time is reduced, i.e. no main memory accesses is needed when searching for free objects.
- Combinatorial → much job can be done in just one clock cycle.
- Parallelism → things can be done at the same time.

3.4 Different Approaches (idea's)

Several ideas on how to exploit the hardware features where developed. Theses ideas are presented here.

3.4.1 Compactation

Compactation is like the golden goal in memory management. With compactation allocated memory is shifted so that all free memory is located in one large chunk. This would minimise memory waste due to fragmentation. The problem is when an object is made free all other objects must be shifted so the hole created by the recently freed object is moved to the free chunk. In software this is an unreasonable approach as the time consumption is very large. Performing compactation in hardware is not believed to improve the time consumption much, as the number of memory accesses needed would be the same.

3.4.2 Internal representation

With internal representation, the representation of the memory space is located inside the allocator. This gives flexibility on where the memory space itself is located. No memory access will be needed when performing allocation or deallocation. This gives fast access to the memory representation for the allocator. Operations on the memory representation can be made without occupying any system bus.

An internal bitmap representing the dynamic memory that can be allocated has been used in several hardware allocators [12], [13]. A bitmap representation, in general, gives the smallest representation of the memory, 1 bit per smallest object that can be allocated. Automatic coalescing is another advantage of a bitmap approach.

A different approach might be to keep an internal ram memory for storing addresses to free and allocated objects i.e. a free and an allocated list. Such an approach [3] is believed to become larger in area. Coalescing is not automatic performed in such a solution and must be considered.

3.4.3 Combinatorial

In hardware it's possible to perform more then sub sequential tasks in a single clock cycle. The trade of being, increased number of tasks performed in sequence can give a reduction of the system frequency.

If a bitmap representation is used the task that might benefit from this is:

- Searching multiple bits in the bitmap at a single clock cycle. Searching the complete bitmap in just one clock cycle probably will result in a very slow system regarding frequency.
- Marking an object as allocated in a single clock cycle.
- Clearing an object from being allocated in a single clock cycle.

3.4.4 Parallelism

In a software allocator all job needs to be carried out between the service request and the service response. With a hardware approach to memory allocation it becomes possible to do job outside the service request – service response sequence. Studies has shown that allocation time can be up to 30% of a program execution time. An allocator that only performs its job during the request – response cycle will be inactive most of the time (70%). This time can be used to do useful stuff instead. The main time consuming task for the allocator is searching for free objects. If this search could be performed during the inactive time, the allocator response time might be cut down. However there is no guarantee for how long (or short) this inactive time frame is. At one time service requests can come close in time and at others they might be further away form each other.

This limits what can be done during the inactive time period. The allocator must not depend on work being done in long inactive time frames. But as there sometimes will exist long periods of inactivity, the allocator should use this time to prepare for active times.

One idea is to, in small inactive time frames search for objects that can be used by the largest amount of requests. As the inactive time frame grows, search for more and more specialised objects that can serve a smaller amount of requests. This concept is called pre-request search.

Another idea is to search for all different size of objects that might be allocated in parallel thereby reducing the need for long time frames. This will need a larger investment in area.

An combination of these to ideas is to use the first idea and let it pick up any other sort of free objects it might encounter on the way, not just the one's that are most useful.

Such ideas mentioned above requires some sort of storage facility for the search result. Depending on allocator mechanism, different kind of storage of free objects is needed when searching in the inactive time frame.

Other activities of the allocator that might be placed in the inactive time is marking and clearing in the bitmaps when performing allocation and deallocation. As the allocator is aimed at rather small number of object sizes and a combinatorial approach to marking and clearing can be taken. These two task are rather small in time compared to the search time. As the number of object sizes grows these two tasks will take longer time.

This kind of ideas will introduce a timing requirement for a minimum time between two sub sequential service requests. If using a internal bitmap representation, the timing requirement is determined by how long time it takes to look at each bit in the bitmap.

3.4.5 Allocator mechanism

This section will describe how common allocator mechanisms can be adopted to hardware using most of the hardware approaches presented earlier.

First fit
A first fit algorithm simply searches for the first free object that is large enough to serve the request. If a larger block then necessary is found, the block is divided and the remainder is kept as a free object. In software a linked list of free objects is usually maintained. If a similar list should be maintained in hardware the list would need to have room for the maximum number of free objects that might exist, leading to a large area. When using a bitmap, the bitmap can be searched for the first object that is large enough to serve the request.

If the pre-request approach to searching is taken, there is need to store an address for each object size that might be allocated. The search will commence at the beginning of the bitmap. When a free object is found, it is stored if none object of the found size is stored before. The search stops when an object of the largest size is found. When performing an allocation service the stored object, large enough, with the lowest address is used to serve the request. The drawback of such an approach is that the storage facility will grow with the number of object sizes that can be allocated.

The storage facility used to give an first fit mechanism might be put into better use by taking the object that best serves the requested size instead of the object that have the lowest address, thereby hopefully reducing fragmentation.

Next fit
Next fit is a "common" optimisation for first fit in software allocators. The principle is to have a pointer to where the last allocation took place and start the search for the next request at the location of the pointer. The reason is to decrease the search time. The drawback of such an approach is that it may affect the locality of the application it allocates for, by scattering objects used by certain application phases and mix them with objects used by other phases. Further such an approach will not reduce the timing requirement compared to a first fit.

Worst fit
A worst fit mechanism always allocates from the worst fitting free object, i.e. the largest free object. In practice this policy seem to work quit badly [14] since it tends to split large objects in favour of small. Leading to a small number of large blocks available. The merit of a worst fit mechanism is that the storage needed when performing pre request search only need to hold the address to one free object, the largest, there by reducing the size of the object storage facility. To further enhance allocator speed one could use cache of worst fit objects instead of just storing the address of one object when searching. In the worst case this will not reduce the allocator-timing requirement, as the cache could be empty.

In a system where the maximum object size is much less then the total memory space, the worst fit approach could work good. But as the maximum object size is close to the total memory space the performance of a worst fit mechanism will decrease, as it will tend to split the large objects when allocating small objects.

Buddy system
Buddy system are described in [18]. Buddy system are quite easy to implement, but in general has worse fragmentation properties compared to first fit or best fit. The DMMX [16] shows that buddy systems are possible to use in a hardware allocator. Existing hardware implementation of buddy system results in a low clock frequency [22].

Best fit
The main idea with a best fit mechanism is to find the smallest free object large enough to serve an allocation request. The strategy is to minimise the amount of wasted space by ensuring that fragments, the part of free objects not used when a free object is allocated, is as small as possible. Best fit has a quit good memory usage in the general case.

To use a best fit mechanism with the pre-request search there will be need for storage space of at least one address for each available object size that can be allocated. During the search the best object for each size should be located and their addresses stored. This storage space will grow large, as the number of object sizes that can be allocated grows large. The timing requirement will depend on the implementation, the minimum time that can be reached is the time it takes to examine all bits in the bitmap for each object size that might be allocated. If the sizes are searched in parallel the time will be the same as for a first fit or a worst fit, but the investment in area will be quite large compared to the worst fit approach.

4 Design

From the ideas in chapter 3 a design approach where developed. The approach uses an internal bitmap representation of the memory space, performs jobs in background using parallelism.

The user interface to the unit is specified, i.e. how the unit should be used. A service call protocol is developed for interfacing the unit. A service consists of giving commands to the unit. Commands access registers in the unit. A definition of which types of errors the unit will be able to detect is made. The unit hardware interface, generic bus interface (GBI), is specified.

4.1 Approach

A internal bitmap representation of the memory space that can be allocated is used. If the complete bitmap should be search in one clock cycle, the logic needed would grow very large if the number of blocks that can be allocated grows large. Such a search would also result in long data paths (low frequency). Therefor the bitmap is divided into sub-parts, called vectors, containing several bits.

Operations on the bitmap are performed in one vector at the time. Vector operations are combinatorial, making the most use of the hardware combinatorial feature. The search operation on the bitmap is performed before allocation requests, utilising that application doesn't always request allocations, it usually performs some actions on the allocated memory space. This will introduce a timing restriction for the user, i.e. the user will have to wait a predictable amount of time between two sequential service requests.

A type of cache, called Address buffer, is used to store addresses found in the pre-service request search. As the allocator primarily should handle small number of different object sizes, the address buffer will be rather small. The address buffer can hold one address for each object size that might be allocated.

Service response are given as soon as possible, the marking in the bitmap (when allocation service) and the clearing in the bitmaps (when deallocation) is performed after the service response. This also adds to the timing restriction.

If all sizes are searched before an allocation service request, the allocator behaviour will be an address ordered best fit behaviour. Best fit is one of the best strategies considering fragmentation. However the time restriction is for one search of the largest size. A search operation on a vector will also find smaller objects if the size currently searched isn't found in the vector. This will guarantee that a search for the largest object will store the largest object currently available. So there will always be the largest available object stored in the Address buffer after a search for the largest size.

The only need to increase the timing restriction longer, is to achieve the address ordered best fit behaviour. In our believe there will be some time where all the sizes will be searched thereby giving the good behaviour.

This main idea is summarised in Figure 2

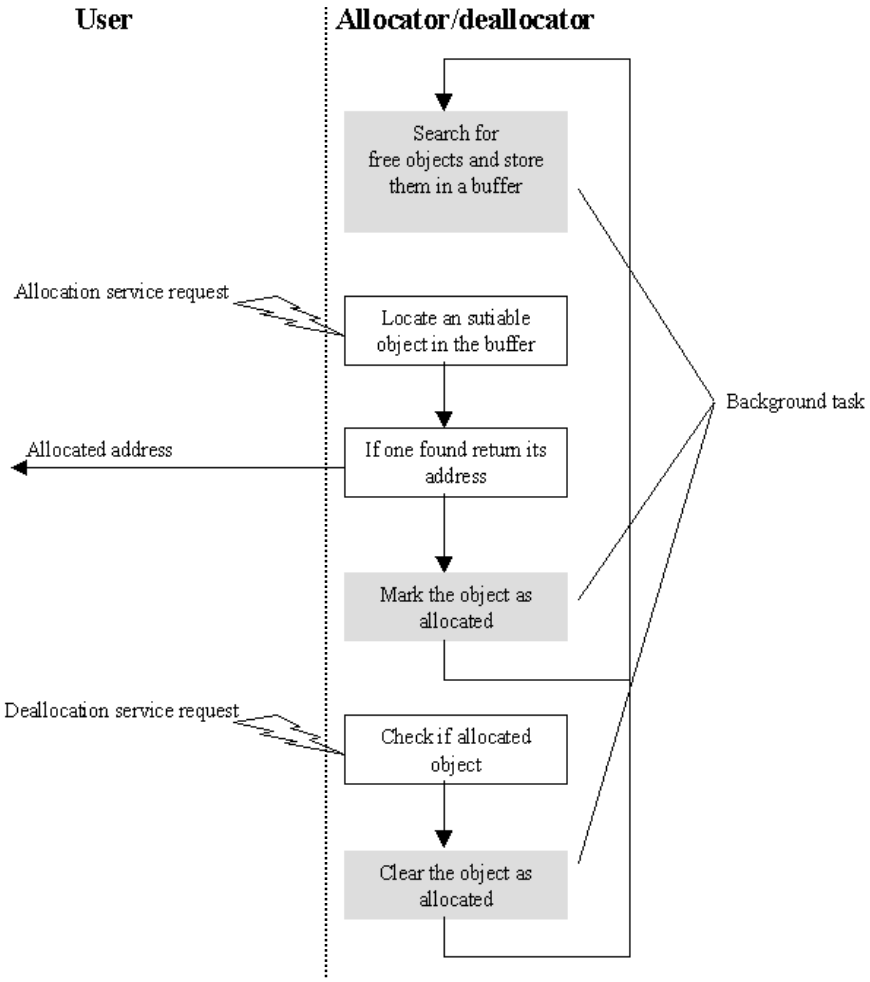


Figure 2: The Allocator / Deallocator main idea

Karl Ingström
kim99001@student.mdh.se
Anders Daleby
ady99002@student.mdh.se

4.2 Memory Representation

The memory space are represented by a bitmap, called the memory bitmap. Each bit in the bitmap represents a block. A block contain a predefined number of bytes, the blocksize. An allocated block is represented with value 1, and a free block is represented with a 0. A memory object is defined by a single block or by blocks with the same value that are beside each other. The sizes of the bitmap are depending on the total memory size and the blocksize. As an example, imagine a memory space of 512 bytes and a blocksize of two bytes. This gives a bitmap that contains 256 blocks (bits).

To be able to deallocate previous allocated blocks, the size of the allocated object must be available. This is due to the fact that a deallocation command only contains the starting address of the object to be deallocated. If the DMA only receives the starting address it is unable to separate two allocated objects that reside beside each other, instead it recognise the two object as one. To store the allocated sizes an extra bitmap are added, the size bitmap.

Definition of free objects
To define a free object in the memory bitmap there are a couple of criteria that must be fulfilled.

Start of a free object: A transition from one to zero.
A zero in the beginning of the bitmap.
End of a free object: A transition from zero to one.
A zero at the end of the bitmap.
Transition from zero to zero and the object is of max size.

A start of an object can not be a zero to zero transition as this could lead to coherence problems in the address buffer.

Figure 3 shows an example of free objects in a bitmap where the max object size is four blocks. All criteria mentioned above are fulfilled.

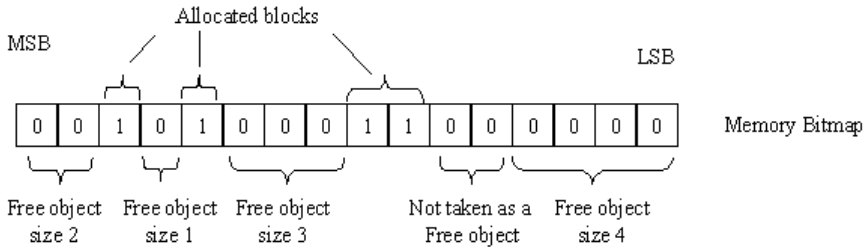


Figure 3: Definition of free objects

How the sizes should be encoded in the size bitmap
As mentioned earlier a size bitmap is needed for the deallocation to take place correctly. The size bitmap contains the same number of blocks as the memory bitmap. The connection between the memory and size bitmaps follow a couple of combinations described in Table 1.

Memory	Size	Interpretation
0	0	Not an allocated object
0	1	Should not happen, illegal combination
1	0	An allocated object continue
1	1	Start of an allocated object

Table 1: Connection between memory and size bitmaps

With the combinations described, the sizes of allocated objects can be detected. This is shown in Figure 4. The need for a size bitmap is also obvious if the allocated object of size one is studied. Without the size bitmap this object would be concatenated with the object of size three.

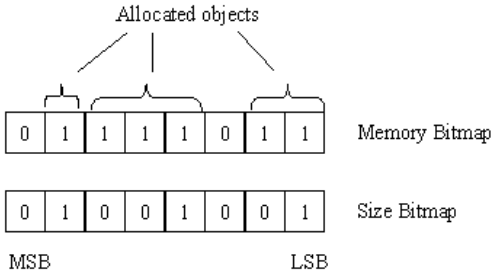


Figure 4: How the size of an allocated object is encoded in the size bitmap

4.3 Error detection

There are circumstances when allocation or deallocation can fail. The unit will be able to detect some circumstances for failure. The types of errors that will be detected are :

Allocation :

- There are not an sufficient space available to allocate the object size. The unit is out of memory.
- The wanted allocation size is zero, i.e. no operation. This is not really an error but the case must be taken care of.
- The object to be allocated is larger then the maximum allowed object size.

Deallocation :

- The address to the object that shall be deallocated is not an allocated address.
- The deallocation address belongs to an allocated object, but is not the start of the object.
- The address to be deallocated is outside the memory space that can be dynamic allocated.
- The deallocation address is not aligned according to the block borders.

4.4 How the unit should be interfaced

The unit will serve two types of services, allocation and deallocation service. Each service is divided into two or more phases. The first phase is to give a service request command to the unit, the second phase will read a service acknowledge response from the unit and the third phase is to obtain the service result. Each phase will have a command that is recognised by the unit. The commands are sent to the unit through a generic bus interface (GBI). The goal of the GBI is to make the unit easily adaptable to different kind of connections, e.g. system buses. The unit should provide a fast response to commands given through the GBI. This is done by having a set of registers. The registers can quickly be accessed by the commands. There are two types of commands, reading or writing of a certain register.

4.4.1 Service call protocol

The DMA can perform two major services, allocation and deallocation. The user will interact with the DMA with commands. Each command is connected to a register in the DMA. A sequence of commands will make up a service.

Allocation service

An allocation service is performed by using a sequence of three commands :

1. The user requests an allocation service by giving a start allocation command and supplying the wanted size to be allocated.
2. The user reads the units service response register, to see if the service was carried out and if there was any errors encountered during the execution of the service.
3. If the was no errors the allocated address can be fetched from a register in the unit. If an error have occurred the type of the error can be fetched from an error register.

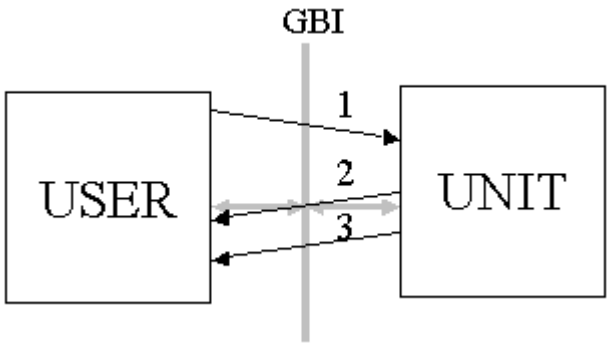


Figure 5: Allocation service

Deallocation service
An deallocation service is performed by using a sequence of three commands where the third is optional.

1. The user requests an deallocation service by giving a start deallocation command and supplying the address to the object that should be deallocated.
2. The user reads the units service response register, to see if the service was carried out and if there was any errors encountered during the execution of the service.
3. (Optional) If there where errors encountered during the execution of the service, the type of error can be fetched from an error register.

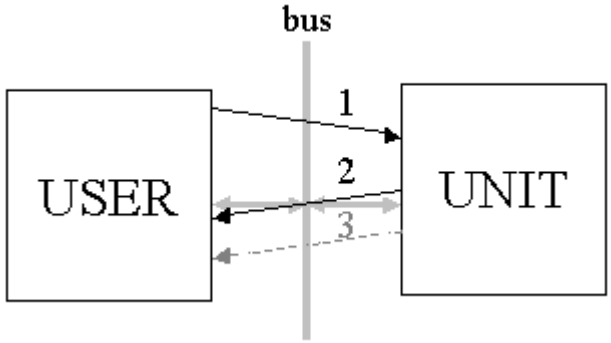


Figure 6: Deallocation service

Giving each command it's own command register instead of passing the commands as parameters will give the possibility map a larger memory space for allocation as addresses passed to the DMA can occupy the whole address space.

DMA commands
The DMA supports a set of seven user commands.

Start allocation
Initiate an allocation service.
Parameter : The wanted size in bytes.
Return value : None

Get allocation result
Used to get the result from an allocation service.
Parameter : None
Return value : The start address of an allocated object if the allocation service performed ok.

Start deallocation
Initiate a deallocation service.
Parameter : The start address of an object to be deallocated.
Return value : None

Karl Ingström
kim99001@student.mdh.se
Anders Daleby
ady99002@student.mdh.se

Read status register

Used to detect if a service request is completed without errors.

Parameter : None

Return value : 0 Was busy so the service request couldn't be accepted. This happens if service requests are given to the unit, violating the timing restriction.
1 The service request have been completed without any errors.
2 The service request have been completed with some errors.
3 The command sent was not valid.

Read error register

Used to read the error register.

Parameter : None

Return value : The error register, in which a flag indicate which error that has been detected. The types of errors the DMA detects are :

1. Out of memory, no free object of appropriate size could be found.
2. Size error, the wanted allocation size is out of range for the DMA to handle.
3. Zero size, the wanted allocation size is zero.
4. Not allocated, the deallocation address sent to the DMA is not an allocated object.
5. Object align, the deallocation address is not the start address of an allocated object.
6. Range error, the deallocation address is outside the range of the memory space that can be allocated.
7. Block align, the deallocation address is not aligned according to the block borders.
8. Command error, the DMA did not recognise the command.

Set memory offset

Used to change the offset of the memory

Parameter : The new offset for the memory. The offset should be a multiple of the memory size.

Return value : None

Get memory offset

Used to read the offset of the memory

Parameter : None

Return value : The current offset for the memory

Device driver example

Allocation service

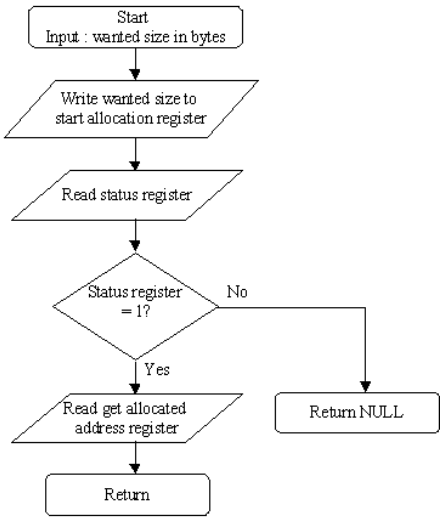


Figure 7: Device driver flowchart for allocation

Karl Ingström
kim99001@student.mdh.se

Anders Daleby
ady99002@student.mdh.se

This device driver corresponds to an ANSI-C [11], *malloc()* implementation where no error handling what so ever is needed. More error handling could be put in the allocation device driver by reading the error register when the status register is unequal to 1.

Deallocation service

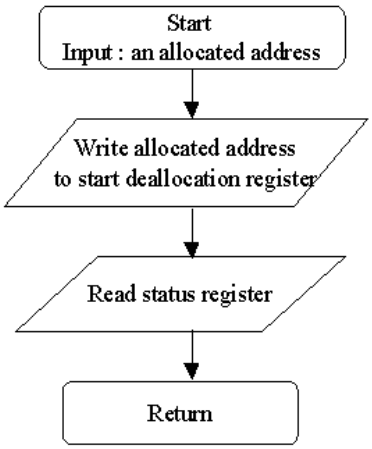


Figure 8: Device driver flowchart for deallocation

This device driver corresponds to an ANSI-C [11], *free()* implementation where no error handling what so ever is needed. More error handling could be put in the deallocation device driver by reading the error register when the status register is unequal to 1.

4.4.2 Generic Bus Interface

The hardware interface to the DMA consists of a generic (bus independent) interface, GBI. The GBI maps the DMA to the environment of the user. There are two types of transactions on the bus, read and write. The generic bus interface consists of the signals described in Table 2. The bus protocol use a handshake behaviour with the two signals, chip_select_n and ack.

Name	Direction	Description
clk	In	System clock
reset_n	In	Reset signal (active low)
command(3:0)	In	Used to give commands to the DMA
data_in(Address_space_Bits-1:0)	In	Data bus in, command parameters
data_out(Address_space_Bits-1:0)	Out	Data bus out, command return values
chip_select_n	In	Chips select (active low), activate detection of commands
ack	Out	Acknowledge of received command

Table 2: GBI Signals

The parameter Address_space_Bits are the bits needed to represent the address space in which the memory reside, see Generic inputs, chapter 5.2.1.

The parameters that are used by the DMA are stored in a set of registers inside the DMA. The registers are read from or written to, depending on the given command.

Timing for GBI

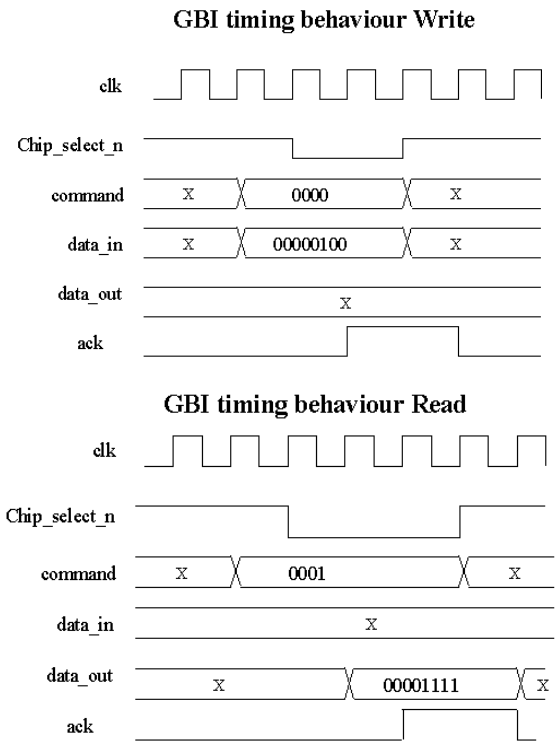


Figure 9: GBI timing

The timing behaviour for the GBI during a write and read cycle are described in Figure 9. In writing mode it is recommended that the command and data-in signals is set one clock period before the chip select signal is activated. This ensures that the signals are stable when chip select is activated. In read mode the command should also be set out one clock period before chip select. The memory unit responds to the read command by setting the data-out signal one clock period before the acknowledge signal is activated. Just as in the write mode this ensures that the data signal is stable.

5 DMA architecture

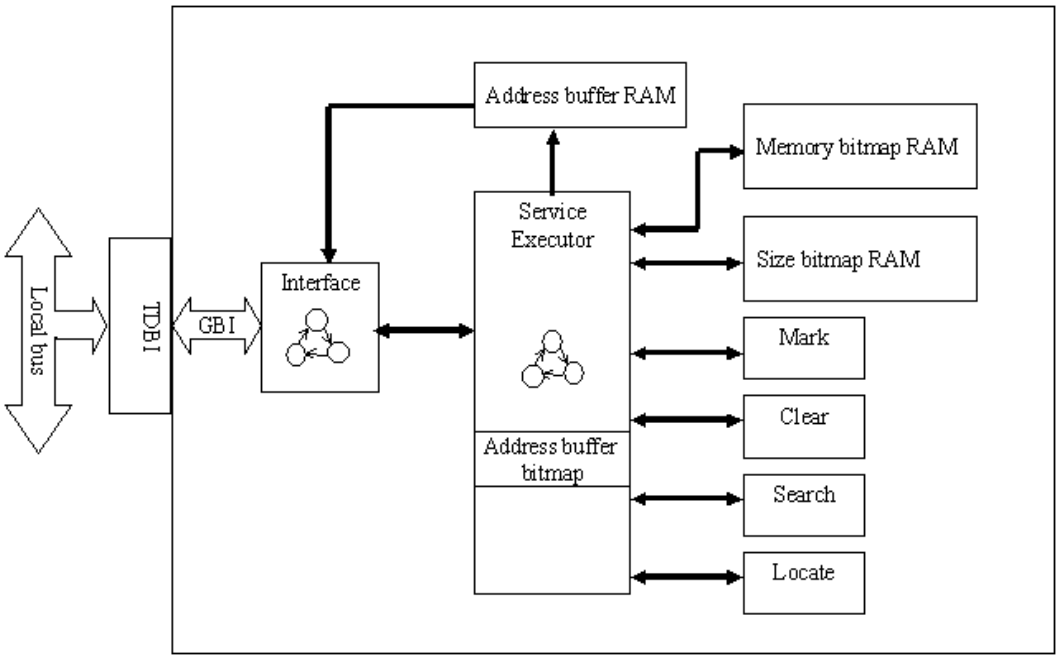


Figure 10: The DMA architecture

The DMA consists of a bus interface, which decodes the DMA commands, a service executor which can carry out the two services provided by the unit, allocation and deallocation, RAM memories for the bitmap vectors, an address buffer for fast allocations and combinatorial units for bitmap operations. The dynamic memory space where the actual dynamic allocable memory is located, resides not in the unit itself and can thereby be located anywhere in the system.

The size of the unit depends on the settings of the generic parameters. The parameters also determines the time restrictions between two service requests.

5.1 Component descriptions

5.1.1 Interface

The interface unit receives commands passed to it from the user. The commands are given to the interface unit in a register fashion way, i.e. the commands either reads from a register or write to a register. The registers used by the interface is described in Table 3. It is the interface responsibility to update the registers. If the command is a valid service request, the interface stores the allocation size or deallocation address in the corresponding register and then activates the service executor.

The communication between the user and the interface unit follows the GBI handshake behaviour, the GBI is described in chapter 4.4. A wanted behaviour is that the bus connected to the DMA is not occupied any longer then needed. The interface therefor have a fast response to actions taken on the GBI.

The interface unit is also responsible for the memory offset. The unit should use the memory offset register to add the offset to the allocated address before it is sent to the user. For a deallocation service the memory offset should first be removed from the address before the address is stored in a register.

Register	description	Mode
error_register	The different errors that can occur	R
allocation_size	The wanted size to allocate	W
deallocation_address	The address to deallocate	W
allocated_address	The address that have been allocated	R
status_register	Status of the service request	R
memory_offset_register	The offset where the memory reside	R/W

Table 3: Registers in the interface

5.1.2 Memory Bitmap RAM

The memory Bitmap RAM unit stores the memory bitmap. Each bit in the bitmap corresponds to the allocation status of a block, with 1 being allocated and 0 free accordingly. The purpose of the unit is to divide the memory bitmap into smaller groups. One group in the memory Bitmap RAM is called a memory vector. The memory vectors are accessed by the Service Executor unit. The advantage of this is that any action on the bitmap can be done on a smaller part. This means that the blocks that operates on the memory vectors that the service Executor has loaded can be made smaller and faster.

Figure 11 shows an example with a memory bitmap with sixteen blocks that are divided into four memory vectors. The start of the memory bitmap is stored at RAM address zero, and the end of the bitmap is stored at the highest RAM address.

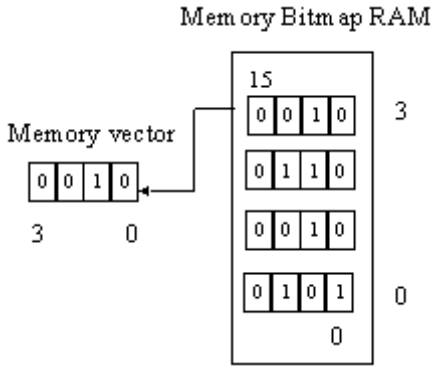


Figure 11: Memory bitmap RAM

The memory Bitmap RAM is constructed, with synchronous writing and asynchronous reading. And the size of memory Bitmap RAM depend upon the values of the generic parameters. The total number of blocks divided with the length of a vector gives the total number of vectors. The number of vectors stored in the RAM memory have a great impact on the worst case time which is presented in chapter 5.3 and 6.12.

5.1.3 Size Bitmap RAM

The size bitmap RAM is constructed in the same way as the memory bitmap RAM, with the difference that it is the sizes of the allocated object that are stored in the RAM memory. How the sizes is coded into the bitmap is discussed in chapter 4.2.

5.1.4 Address buffer

In the address buffer the block addresses obtained by the search is stored for quick access during an allocation service request. The address buffer is constructed as a RAM memory with synchronous writing and asynchronous reading, it also have separate data buses for read and write. The unit can store one address for each block size that can be allocated. The address for the block sizes that can be allocated is stored at the index that correspond to their size, i.e. the smallest size have index zero.

To keep track of which object sizes that have an stored address in the address buffer, an address buffer bitmap in the service executor is used. The index in the Address buffer bitmap is equal to the index in the address buffer. The values in the address buffer is not cleared when the addresses are allocated. Instead a zero in the address buffer bitmap indicates a invalid address while a one indicates a valid address. Se Figure 12 where the three smallest sizes have valid addresses. It is the service executor block that activate all actions on the address buffer. The addresses that should be stored are provided by the service executor, while the interface reads the addresses.

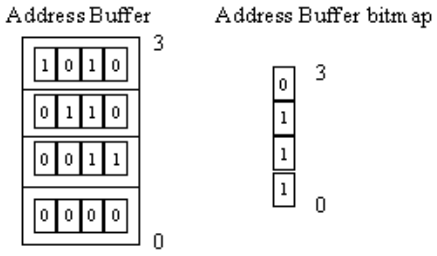


Figure 12: Connection between the address buffer and the bitmap

5.1.5 Search

The search unit searches for free objects in a memory vector provided by the service executor. The object that can be found are of the wanted size or a smaller size. This means that if no objects of the wanted size is found, it is possible that the search unit return a smaller object instead. Free objects are defined in Figure 13 below, in the figure the max object size is four. Definitions of free objects are further discussed in chapter 4.2

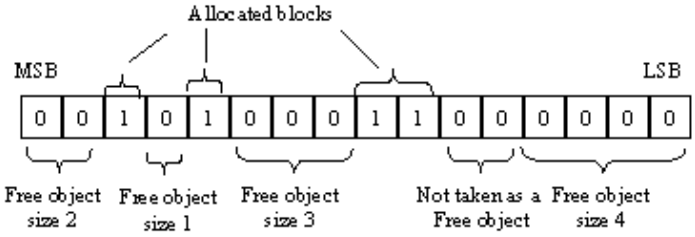


Figure 13: Definition of free objects

If a free object is found, the size of the object and the index to the start of the object in the memory vector are returned. If two objects of different sizes are found the largest of them are returned. If two objects of the same size is found the object with the lowest index is chosen.

Since the memory bitmap are divided into smaller memory vectors an object can be located in one or more vectors. To deal with this, there is a counter that holds the number of contiguous zeros at the end of the vector. There is also a counter that are used to continue counting the size of an object starting in the previous memory vector. If the counter from the previous vector have a nonzero value and the current vector have the MSB marked as allocated, the object is found in the old vector. The outcome of this can be explained by Figure 13, the free object of size three is returned as the found object and the object of size four is transferred to the next vector to be searched. The service executor unit must take special care of the end vector. If the found object is in the old vector a found in old signal is set high, otherwise an ordinary found signal is set high. The two different found signals is never set at the same time.

5.1.6 Locate

The locate unit searches for an object with a valid address that are stored in the address buffer. To search for a valid address it uses the address buffer bitmap in the service executor. The unit locates a valid index in the bitmap, i.e. bit value equal to one. The unit reads the Address buffer bitmap and clear all the bits with an index smaller then the wanted allocation size. It then finds the smallest of the remaining valid indexes. If the search for a valid index is successful the found size is returned and a found signal is set high. If the search for an index fails there are no object large enough for the wanted allocation size.

This block is totally combinatorial and it delivers the answer to the service executor within one clock cycle.

5.1.7 Mark

The unit marks the memory and size bitmaps so that an object currently taken from the address buffer will be marked as allocated. The service executor loads the memory and size vectors where the marking should start. The mark unit receives a vector index that indicates where the marking should start in the vectors. The marking continues as long as the size provided by the service executor is not reached. To mark an object as allocated the bits in the memory and size vector is set according to Table 4. This unit is totally combinatorial and the marking in one vector is done within one clock cycle.

Memory	Size	Interpretation
0	0	Not an allocated object
0	1	Should not happen, illegal combination
1	0	An allocated object continue
1	1	Start of an allocated object

Table 4: Combinations of the memory and size bitmaps

If the marking should take place over several vectors, the service executor load new vectors and supply the mark unit with a new size and vector index.

If the illegal combination, 01, for some odd reason should occur, there are two ways to mark that index. If it is the start of the allocated object the unit marks the combination 11. If the illegal combination occurs after the first bit the combination 10 is marked.

In Figure 14 there are an example where an allocated object of size four should be marked. The left picture show the original memory and size vectors with an illegal combination at index 2. In picture A the size vector have been cleared and the illegal combination have been ignored. In picture B the size vector have been untouched except for the start of the object. The result of this is that there are two objects of size two, where the object starting at index two doesn't have any reference to it. So by letting the unit clear the size vector in case A, a memory leakage is avoided.

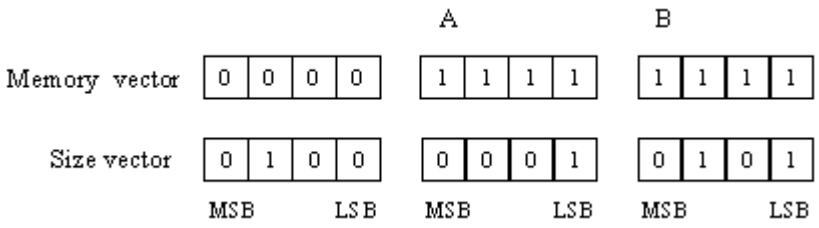


Figure 14: Illegal vector combination

5.1.8 Clear

The job of the clear unit is to clear objects that previous have been marked as allocated in the size and memory bitmap. This is a combinatorial unit and it clears the memory and size vector in just one clk. To know the size of the allocated object the unit uses the information that the combinations of the memory and size vector give, se Table 5. The service executor provide the clear unit with an index to the start of the object to be deallocated. Before the index is sent to the clear unit the service executor check the vectors to ensure that the index is a start of an allocated object.

The unit starts its operation by clearing the bit in the size vector that are indicated by the vector index, giving the combination 10. The unit then starts at the vector index and continue clearing the vectors as long as the combination 10 appears. If some other combination appears no clearing of the vectors are done, instead the clearing is finished and a done signal is sent to the service executor. If the illegal combination 01 occurs the unit set the memory bit high to achieve the 11 combination. This could lead to memory leakage but the situation should not appear under normal circumstances.

Memory	Size	Interpretation
0	0	Not an allocated object
0	1	Should not happen, illegal combination
1	0	An allocated object continue
1	1	Start of an allocated object

Table 5: Combinations of the memory and size bitmaps

If the object to be cleared stretches over several vectors the service executor will handle it, by loading new vectors and a new index. When the new vectors are loaded the service executor doesn't make any check on the vector combinations. The new index will have the value zero and this could lead to some problems if the new vectors starts with the combination 11, start of an allocated object. The unit usually starts its operation by clearing the size bit corresponding to the index. Which in this case means that the start of the new object would be missed, and the clear unit then accidentally deallocate the next object. Therefor there is a signal that indicates if the current vector is the one where the object have its starting address.

If the object that have been cleared is followed by a free object a signal is send to clear the address buffer bitmap. The buffer bitmap is cleared to avoid coherence problems in the memory bitmap.

5.1.9 Service executor

The service executor have contact with all the other units in the DMA, Figure 10. When the interface receives a service request the service executor are enabled by a signal. The service executor then uses the other units that are needed to perform the service request. If there are no command sent from the interface, the service executor still works in the background, with the search for free objects in the memory bitmap. The memory bitmaps is searched form the top to the bottom. This will simplify address calculations when a new free object is found

When the unit responds to a command from the interface it can detect three different errors. Out of memory, the deallocation address doesn't match the start of an previously allocated object or the deallocation address points to a block that is not allocated.

For the service executor to deliver data to the mark, clear, locate and search unit it uses some registers. The memory vector and size vector registers, stores a part of the memory and size bitmaps that are read by the units. The size register, stores the wanted size to allocate or to mark as allocated. The address buffer bitmap, indicates which indexes in the address buffer that are valid.

The interconnections between the service executor and the other units in the construction are described below. Some of the units connected to the service executor are combinatorial, so it is up to the service executor to decide which response signal it should take care of. The service executor should be able to perform one vector operation each clock cycle. That is as one vector is processed by either the mark, clear or search unit, a new vector should be loaded from the memory bitmaps and the result from the previous operation stored. This is a type of pipeline operation for the results.

5.1.10 TDBI

Stands for Target Dependent Bus Interface. As the name point out it is dependent on the technology of the system. Towards the MA the TDBI should be constructed to follow the GBI interface. This makes it possible to interface the DMA towards different applications. As the TDBI is target dependent it is not a part of the basic DMA architecture. The TDBI used in the test application is described in chapter 7.3.4.

5.2 DMA Configuration

The DMA have been developed to be adaptable to different settings. These settings are the size of the memory space to be allocated, the maximum number of bytes that can be allocated in a service, the size of a block, he number of blocks that are searched per clock cycle and the number of bits needed for the data in and out signals to the unit. Through these setting the developed architecture can be reused in different environments. The settings are given to a DMA construction as generic inputs.

5.2.1 Generic inputs

The user have five different generic values to set. The inputs and their default value are described in Table 6. The generic inputs are used to configure the system so it match the wanted behaviour.

Name	Description	Default value
Memory_space	The total number of bytes in the dynamic memory space that can be allocated.	8192
Address_space_Bits	This value sets the width of the data_in and data_out signals.	16
Block_size	The number of bytes in each block.	8
Max_allocation_size	Max number of bytes that can be allocated	32
Search_width	Search width, the number of blocks that are searched simultaneously	8

Table 6: Generic inputs

All values except Address_space_Bits the should be a power of 2. This will simplify implementation. The settings of the parameters effects the timing behaviour of the DMA unit, chapter 5.3 and 6.12. The generic parameters also determine the area of the module, se chapter 8.2.

5.2.2 Restrictions on the generic inputs

There are some restriction that must be considered when setting the DMA generic parameters. One is that all generic input except Address_space_Bits should be a power of two. The other restrictions are described in Table 7. If the restrictions are ignored the unit behaviour is not guaranteed.

Parameter	Restriction	Description
Address_space_Bits	>= 8	The data_out bus must be large enough so the error register can be delivered on the data_out bus.
Memory_space	<= $2^{Address\_space\_Bits}$	The memory can not occupy a larger address space then the Address_space_Bits parameter indicates.
Max_allocation_size	<= Memory_space / 2	A restriction is made so only half of the memory space can be allocated at one time.
Block_size	<= Max_allocation_size	The size of a block must be smaller then or equal to the largest size that can be allocated.
Search_width	>= 2 <= (Memory_space / Block_size)/2	No implementation have been made for the case of a search width of one. This is mainly due to the fact that the hole system could have been implemented in a different more efficient fashion. The search width can't be larger then half the total number of blocks available in the memory.

Table 7: Restrictions on the generic parameters

5.3 Timing restrictions

The timing restrictions for how fast two sequential services can be carried out depends on the generic settings of the DMA. Generally the timing restriction when performing a allocation service can be divided into :

- The time for the interface to pass the command to the service executor.
- The time to locate an suitable object in the Address buffer.
- The time to mark the object as allocated.
- The time is takes to do a search for an object of the largest size in the memory bitmap

The restriction for deallocation service is similar :

- The time for the interface to pass the command to the service executor.
- The time to clear the object.
- The time is takes to do a search for an object of the largest size in the memory bitmap.

Formulas for the final timing restrictions can be found in the implementation chapter, 6.12.

6 Implementation

In the following chapter the construction specification for the different units in the DMA is presented. Implementations of the DMA parts are presented. Since the memory bitmap RAM, size bitmap RAM and the address buffer are made up of the same type of RAM memory they are described together in one chapter. Some of the units is totally combinatorial and this could lead to timing problems depending on the values of the generic inputs. Much time where therefor spend on optimisation of the combinatorial units regarding the timing behaviour.

6.1 Internal generic values

The different units in the DMA need some generic parameters that are used internally. The parameters can be user supplied or internal generated generic parameters. The internal generic parameters are calculated from the user supplied generic parameters. The calculation of the internal generic values and their default value is described in Table 8. The calculation of the internal generic parameters is done when the different units are mapped together to form the DMA architecture.

Parameter	**Calculation and description**	**Default value**
Mem_address_Bits	The $\log_2$ of the Memory_space, must be at least one. The number of bits that are needed to represent the memory space.	13
Max_object_size	Max_allocation_size / Block_size The maximum number of blocks that can be allocated.	4
Max_object_size_Bits	The $\log_2$ of the Max_object_size, must be at least one. The number of bits needed for the Max_object_size parameter to be represented.	2
Search_width_Bits	The $\log_2$ of the Search_width, must be at least one. Bits needed to represent the Search_width.	3
Blocksize_Bits	The $\log_2$ of the Block_size. Bits needed to represent the Block_size.	3
Block_address_Bits	Mem_address_Bits – Blocksize_Bits Bits needed to represent the total number of blocks that can be allocated.	10
Bitmap_address_Bits	Block_address_Bits – Search_width_Bits Bits needed to address the memory bitmap and size bitmap RAM memories.	7

Table 8: Internal generic parameters

6.2 Interface

6.2.1 Specification

The interface unit should receive commands passed to it from the user. The communication between the user and the interface unit should follow the GBI handshake behaviour. The GBI is described in chapter 4.4. The interface should respond to any action started by the GBI as fast as possible. The commands are given to the unit in a register fashion way, so during a write operation the unit should store the parameters given on the data_in signal in the appropriate register. During a read operation the unit should return the value from the wanted register on the data_out signal. The different registers are shown in Table 13. If the command sent requires some actions from another unit the service executor should be enabled.

The interface unit is responsible for the memory offset. The unit should use the memory offset register to add the offset to the allocated address before it is sent to the user. For a deallocation the memory offset should first be removed from the address and then stored in the deallocation_address register.

The interface unit has some generic inputs, these are shown in Table 9. The names and description of the input and output signals that the interface should use are described in Table 10 and Table 11.

Name	default valu
Mem_address_Bits	13
Address_space_Bits	16
Blocksize_Bits	3
Block_address_Bits	10
Max_object_size_Bits	2
Max_allocation_size	32

Table 9: Generic inputs for the interface

Name	Direction	Description
clk	In	System clock
reset_n	In	System reset (active low)
chip_select_n	In	Chips select (active low), activate detection of commands
ack	Out	Acknowledge of received command. Goes high when chip_select_n goes low. Logical low when the command is executed or an error is detected. It should not go low before chip_select_n goes high.
command(3:0)	In	Used to give command to the interface
data_in (Address_space_Bits -1:0)	In	Data bus in, command parameters
data_out (Address_space_Bits -1:0)	Out	Data bus out, command return values

Table 10: Signals between the interface and GBI

Name	Direction	Description
enable	Out	Logical low as long as no new command is to be performed. If a valid allocation or deallocation command has been received and busy is logical low the signal goes high. The signal shall remain logical high at least to the time when Service executor starts executing, busy goes high.
busy	In	Logical low as long as the service executor is not executing. As soon as the service executor receives a command it will go logical high and stay high until the search for the largest object size is done. When Busy goes high enable should go low.
cmd	Out	Command for the service executor to execute 0 = allocation, 1 = deallocation.

		The signal shall remain stable from when enable goes logical high and at least to the time when busy goes logical high.
allocation_size (Max_object_size_Bits-1:0)	Out	Used to read out the content in the allocation_size register. The value should only be changed by the start_allocation command.
deallocation_address (Block_address_Bits-1:0)	Out	Used to read out the content in the deallocation_address register. The value should only be changed by the command start_deallocation.
done	In	Logical high for one clock period when a start allocation or start deallocation command has been finished by the service executor. During this period the allocated_address register should be updated. Also the done flag in the done register should be turned to logical high.
out_of_mem	In	Logical high for one clock period when out of memory is detected by the service executor. During this period the flag out_of_mem in the error register should be set logical high.
deallocation_error(1:0)	In	Logical high for one clock period when a deallocation error is detected by the service executor unit. BIT 0: Deallocation of a not allocated object. BIT 1: Missaligned (not start of an allocated object) address During this period the corresponding flag in the error register should be set logical high.
allocated_address (Block_address_Bits-1:0)	In	Used to update the allocated_address register. The register should be updated when the done signal is logical high.

Table 11: Signals between the interface and the other units

Commands
The interface should be able to detect seven different commands, they are described in Table 12 below. For the start allocation and start deallocation commands the command should be passed on to the service executor and the enable signal go high. If an unknown command is received, the status register should be set to indicate that case. To be able to receive a start_allocation or start_deallocation command, the done bit in the status register must be clear. This should be accomplished with the command read_status which also returns the status register.

Command	Value
START_ALLOCATION	0000
GET_ALLOCATION_RESULT	0001
START_DEALLOCATION	0010
READ_STATUS	0011
READ_ERROR_REGISTER	0100
SET_MEMORY_OFFSET	0101
GET_MEMORY_OFFSET	0110

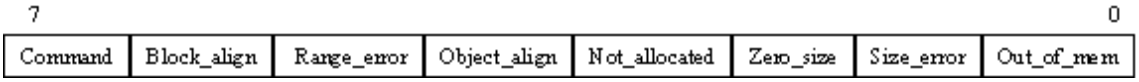
Table 12: Commands

The registers
All the registers should be cleared during reset. It is the interface responsibility to handle the registers accessed by the user commands and other blocks in the DMA. The register is shown in Table 13.

Register	Mode
error_register	R
allocation_size	W
deallocation_address	W
allocated_address	R
status_register	R
memory_offset_register	R/W

Table 13: Registers in the interface

error_register
The error register contains all the errors that can be detected during a service request for an allocation or deallocation. The error bits 0, 3 and 4 are detected by the service executor unit. The other five errors should be detected by the interface unit. The error_register should be cleared either when, start_allocation, start_deallocation or read_error_register command is received. This gives the user a choice of reading or not reading the error register. The command read_error_register reads the register and then clears it.



Command : Indicates not implemented command.
0 = no error
1 = error

Block_align: Indicates if the deallocation byte address is not a multiple of the block size.
0 = no align error
1 = block align error

Range_error: Indicates if the deallocation address is outside the memory region.
0 = no address error
1 = offset error

Object_align: Indicates if the deallocation block address is the start of an allocated object, comes from deallocation_error (1).
0 = no align error
1 = object align error

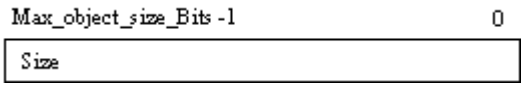
Not_allocated: Indicates if the address to deallocate is allocated, comes from deallocation_error (0).
0 = allocated
1 = not allocated

Zero_size: Indicates an allocation service request for a object of zero size.
0 = requested allocation service with size larger then 0 byte
1 = requested allocation service with size 0 byte

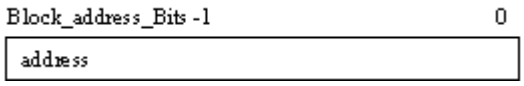
Size_error: Indicates error in the size parameter for allocation service request
0 = requested allocation of allowed size
1 = requested allocation with size larger then maximum allowed size

Out_of_mem: Indicates out of memory
0 = ok
1 = could not find enough memory space to serve the request

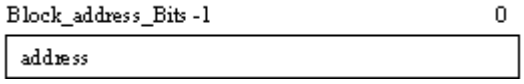
allocation_size
Store the size that should be allocated. Should only be changed by the start_allocation command.



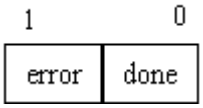
deallocation_address
Store the block address that should be deallocated, the memory offset should first be removed. The start_deallocation command should update the register



allocated_address
The block address to the latest allocated object. The register should be updated when the done signal is high



status_register
Contains the status of the latest service request. The status register is summarised in Table 14.

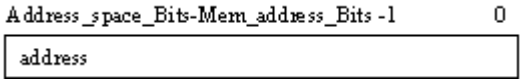


The done bit should be set when the done signal goes high, the bit should be cleared when the status register are read. The error bit should be set if some error in the error register occurs, it should be cleared when the error register is cleared. If the error bit in the status register is high the error register must be cleared before the bit goes low.

Error	Done	Meaning
0	0	The unit was busy and could not receive the given service request
0	1	The service request have been carried out without any errors
1	0	The service request have been carried out with some errors
1	1	A incorrect command was given to the unit.

Table 14 The meaning of the status register

memory_offset_register
Store the bits needed to give the memory offset in the address space. Default offset is zero.



6.2.2 Construction

The interface unit is constructed with a state machine to respond to the actions on the GBI, Figure 15. The state machine is constructed to meet the smallest timing requirements for the interaction with the GBI.

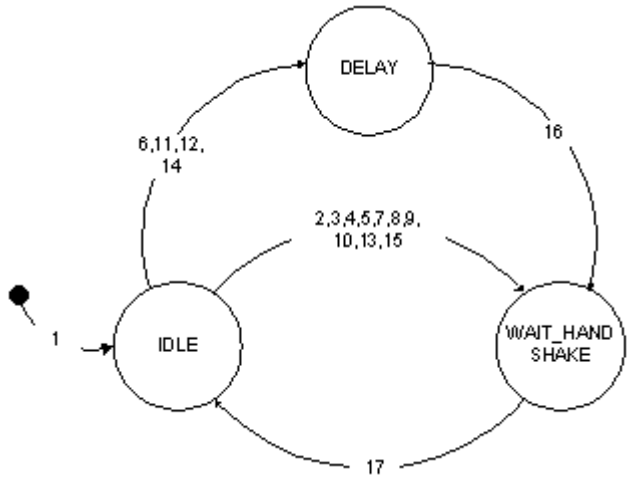


Figure 15: state machine for the interface

The different transition are described in Table 15. In the table the events that causes a transition and the actions taken when a transition occurs is listed. The transition between the idle and delay states corresponds to a read operation. The delay state guarantees that the data_out signal is stable before the ack signal is returned. When going from idle to wait_handshake a write operation to the interface is wanted. For all transitions from the idle state the chip_select_n signal must be low.

Transition	Event	Action
1	Reset released.	All signals and registers are cleared.
2	Start_allocation command. The wanted allocation size is zero.	Set ack signal high and error_register 2 high. Set a signal to clear old errors from the service executor.
3	Start_allocation command. The wanted allocation size is to large.	Set ack signal high and error_register 1 high. Set a signal to clear old errors from the service executor.
4	Start_allocation command. A valid allocation request.	Store the wanted allocation size, convert the number of bytes to number of blocks with blocksize. Set ack signal high. Set allocation commando to the service executor and enable it. Set a signal to clear old errors from the service executor.
5	Start_allocation command. The service executor is busy or done flag in status register is not cleared.	Set ack signal high and go to the state wait_handshake. Set a signal to clear old errors from the service executor.
6	Get_allocation_result.	Set out the address on data_out. Convert from block address to memory address with the memory offset register.
7	Start_deallocation command. Wrong memory offset on the deallocation address. Only detected when offset exists.	Set ack signal high and error_register 5 high. Set a signal to clear old errors from the service executor.
8	Start_deallocation command. Deallocation address is not a multiple of the blocksize. Detected when blocksize is larger then 1.	Set ack signal high and error_register 6 high. Set a signal to clear old errors from the service executor.

9	Start_deallocation command. Valid address for deallocation.	Store the wanted deallocation address, if needed convert to block address with memory offset and blocksize. Set ack signal high. Set deallocation commando to the service executor and enable it. Set a signal to clear old errors from the service executor.
10	Start_deallocation command. The service executor is busy or done flag in status register not cleared.	Set ack signal high and go to the state wait_handshake. Set a signal to old clear errors from the service executor.
11	Read_status command.	Set out the status register on data_out. Set a signal to clear the done bit in the status register next clk.
12	Read_error_register command.	Set out the error register on data_out. Set a signal to clear the error register next clk.
13	Set_memory_offset command.	Store the delivered offset in the memory offset register. The offset is stored in the offset register. Set the ack signal high.
14	Get_memory_offset.	Set out the actual memory offset value on data_out.
15	Unknown command.	Set ack signal high and error register 7 high. Set a signal so the done flag in the status register is set high.
16	No event required.	Set ack signal high.
17	Chip_select_n = 1.	Set ack signal low.

Table 15: Description of the interface state machine

To update registers or part of registers that get the values from other units in the system there are five small processes.

Update of the allocated address register. The value is only updated when the done signal from the service executor is high. The done signal is high for one clk.

The error register have three different flags that are set by the service executor, out of memory, not allocated and object align. They are in three different processes but the behaviour is the same. If the internal signal to clear old errors are high the flag is cleared. When a flag has been set high, it is kept high although the error from the service executor only is high for one clk.

The done flag in the status register is cleared when the signal clear_done is set, i.e. the status register have been read. It can be set high either when the done signal from the service executor is high, or when the internal set_done signal is high. The flag keep its value until any action is taken.

6.3 Service executor

6.3.1 Specification

The service executor should use some of the other units to perform a command, the units to be used are, search, mark, clear and locate. To allow the units to perform their task the service executor must deliver data to each unit. The data needed by each unit are described in Table 17. The search, mark and clear units works with the same signals provided by the service executor, this means that the service executor must decide which response signal it should use.

When the unit responds to a command from the interface it should be able to detect three different errors. Out of memory, the deallocation address doesn't match the start of an previously allocated object or the deallocation address points to a block that is not allocated. If no errors are found the done signal should be set high as soon as possible and be kept high for one clk. The busy signal should be set high when the unit detects the enable signal. It should remain high until the search for an object of the largest size is finished or an error is detected.

The service executor uses some generic inputs, these are shown in Table 16. The names and description of the input and output signals that the service executor should use are described in Table 17.

Name	Default value
Block_address_Bits	10
Max_object_size_Bits	2
Max_object_size	4
Search_width	8
Search_width_Bits	3
Bitmap_address_Bits	7

Table 16: Generic inputs for the service executor

Name	Direction	Description
clk	In	System clock.
reset_n	Out	Reset signal, active low.
enable	In	Logical low as long as no new command is to be performed. If a command is to be performed the signal goes high. The signal is logical high at least to the time when busy goes high
busy	Out	Logical low as long as no command is executing. As soon as a command is received it will go logical high and stay high until a search for the largest size is completed.
cmd	In	Command to execute: 0 = allocation, 1 = deallocation. The signal is stable from when enable goes logical high and at least to the time when busy goes logical high.
allocation_size (Max_object_size_Bits-1:0)	In	Give wanted number of blocks to allocate. The signal is stable between two commands.
deallocation_address (Block_address_Bits-1:0)	In	Give the block address to the object to be deallocated. The signal is stable between two commands.
done	Out	Logical high for one clock period when a correct deallocation or allocation service response is available.
out_of_mem	Out	Should be set logical high for one clk when out of memory is detected.
deallocation_error(1:0)	Out	Logical high for one clock period when a deallocation error is detected. BIT 0: Deallocation of a not allocated object. BIT 1: Missaligned address (not start of an allocated object)
ABuffer_we_n	Out	Sets the address buffer RAM write mode, 1= no update, 0 = update, read mode by default.
ABuffer_address_write (Max_object_size_Bits-1:0)	Out	Where to store the address of a found object.
ABuffer_data_write (Block_address_Bits-1:0)	Out	Used to store the address to a free object found
Abuffer_address_read (Max_object_size_Bits-1:0)	Out	The address where to obtain an object from the address buffer.
allocated_address (Block_address_Bits-1:0)	In	The address to a object newly allocated by the locate unit. Used to update the memory and size bitmap RAM.
MemoryB_we_n	Out	Enables update of the memory bitmap RAM. 1 = no update, 0 = update.
MemoryB_address_write (Bitmap_address_Bits-1:0)	Out	Address for writing a memory vector to the memory bitmap RAM.
MemoryB_data_write (Search_width-1:0)	Out	Used to update a vector in the memory bitmap RAM.
MemoryB_address_read (Bitmap_address_Bits-1:0)	Out	Address to a memory vector in the memory bitmap RAM that shall be read.
MemoryB_data_read (Search_width-1:0)	In	Used to read a vector from the memory bitmap RAM.

SizeB_we_n	Out	Enables update of the size bitmap RAM. 1 = no update, 0 = update.
SizeB_address_write (Bitmap_address_Bits-1:0)	Out	Address for writing a size vector to the size bitmap RAM.
SizeB_data_write (Search_width-1:0)	Out	Used to update a vector in the size bitmap RAM.
SizeB_address_read (Bitmap_address_Bits-1:0)	Out	Address to a memory vector in the size bitmap RAM that shall be read.
SizeB_data_read (Search_width-1:0)	In	Used to read a vector from the size bitmap RAM..
memory_vector (Search_width-1:0)	Out	Give access for the clear, search and mark units to the memory vector that have been loaded from the memory bitmap RAM.
size_vector (Search_width-1:0)	Out	Give access for the clear and mark units to the size vector that have been loaded from the size bitmap RAM.
vector_index (Search_width_Bits-1:0)	Out	Index to the memory and size vectors where the clear and mark operation should start.
size (Max_object_size_Bits-1:0)	Out	Give the size to be used by the locate, search and mark units.
buffer_bitmap (Max_object_size-1:0)	Out	Give access to the address buffer bitmap for the locate unit.
Locate_found	In	Result is valid one clk after the outputs to the Locate unit has been set. 1: indicates that an available object has been found 0: no object of sufficient size can be found
Locate_found_size (Max_object_size_Bits-1:0)	In	The index to a found object, if one exists. The signal is valid when the Locate_found signal is high.
Clear_done	In	High when the deallocation is completed, else low
Clear_clear_bitmap	In	Signal from the clear unit that the address buffer bitmap should be cleared.
Clear_status_in	Out	High signal tells the clear unit that current vector is the vector in which the object starts.
Clear_memory_vector_new (Search_width-1:0)	In	The updated memory vector from the clear unit. Result is valid one clk after the signals to the clear unit have been set
Clear_size_vector_new (Search_width-1:0)	In	The updated size vector from the Clear unit. Result is valid one clk after the signals to the Clear unit have been set
Mark_size_mark_start	Out	High signal tells the mark unit that this is the vector where the object starts.
Mark_memory_vector_new (Search_width-1:0)	In	The updated memory vector from the mark unit. Result is valid one clk after the signals to the mark unit have been set
Mark_size_vector_new (Search_width-1:0)	In	The updated size vector from the Mark unit. Result is valid one clk after the signals to the Mark unit have been set
Search_old_counter (Max_object_size_Bits:0)	Out	Indicates how many contiguous zeros there where at the end of the previous memory vector. Used to find free objects crossing vector borders.
Search_found	In	Set high if a free object of wanted or smaller size is found.
Search_found_in_old	In	Set high if a free object of wanted size or smaller was found entirely in the previous vector.
Search_found_index (Search_width_Bits-1:0)	In	Give the index to the start of the found object in the memory vector. Only valid if Search_found is high.
Search_found_size (Max_object_size_Bits-1:0)	In	Give the size of a found object. Valid only if Search_found or Search_found_in_old is high.
Search_current_counter (Max_object_size_Bits:0)	In	Indicates how many contiguous zeros there where at the end of the memory vector. Used to find free objects crossing vector borders.

Table 17: Input and output signals for the service executor

During a search operation the memory should be searched form the top to the bottom, this means that the memory vectors should be loaded from the highest address first.

When the service executor are enabled by the interface it should respond by setting the busy signal high. If an allocation service request arrives the service executor first should use the locate unit to receive an index to the address buffer. If the locate_found signal is high the done signal should be set high and the mark unit should mark the address delivered on the allocated_address signal as allocated. The unit should also clear the bit in the buffer_bitmap that corresponds to the found index. When the marking is done a search for the largest size should be performed by the search unit. When this search for the largest size is finished the busy signal should be set low and search for the rest of the sizes should be performed. If the locate_found signal is low the out_of_mem signal should be set high and the busy signal low.

If a deallocation service request arrives the service executor first should check for deallocation address errors. If an error is found the appropriate error signal should be set, the busy signal should be set low. If no error is found the unit should set the done signal and use the clear unit to deallocate the object. When the clearing is done a search for the largest size should be performed by the search unit. When this search for the largest size is finished the busy signal should be set low and searches for the rest of the sizes should be performed. The buffer_bitmap should be cleared if the signal Clear_clear_bitmap is high or if the vector_index is zero and it is not the vector with the lowest address. It should also be cleared if the bit in the vector with vector_index-1 is zero.

If there are no new command sent from the interface the unit should work in the background and continue the searching for the rest of the sizes.

6.3.2 Construction

The service executor unit is constructed with a state machine Figure 16. The state transitions are described in Table 18. In the table the events that causes a transition and the actions taken when a transition occurs is listed.

Transition	Event	Action
1	Reset released	Set write enable to size and memory bitmap. Set the current bitmap address write to zero. Read the size and memory vectors with the highest address.
2	While not all memory and size vectors are cleared	Decrease the bitmap address write. Write zeros to the vectors at the current bitmap address write.
3	If all vectors cleared	Clear the write enable for the memory and size bitmap.
4	No event required.	Store the current bitmap address read. Decrease the bitmap address read. Set search size to max size.
5	While not all the memory vectors have been searched.	If a free object is found in the current vector or at the end of the previous vector, store the address in the address buffer and mark it's size in the buffer bitmap. Load the next memory vector.
6	If all vectors have been searched.	Save the current counter from the last vector. Set the read address to the highest vector.
7	If a deallocation command arrives	Set busy signal high. Set out the address to the vectors that should be cleared first.
8	If an allocation command arrives.	Set busy signal high. Store the wanted size to allocate in the size register.
9	If no new command arrives and all vectors have been searched.	Save the current counter from the last vector. Set the read address to the highest vector.
10	If no new command arrives and there are still vectors to search in.	If a free object is found in the current vector or at the end of the previous vector, store the address in the address buffer and mark it's size in the buffer bitmap. Load the next memory vector.
11	If a deallocation command arrives.	Set busy signal high. Set out the address to the vectors that should be cleared first.
12	If an allocation command arrives.	Set busy signal high. Store the wanted size to allocate in size register.

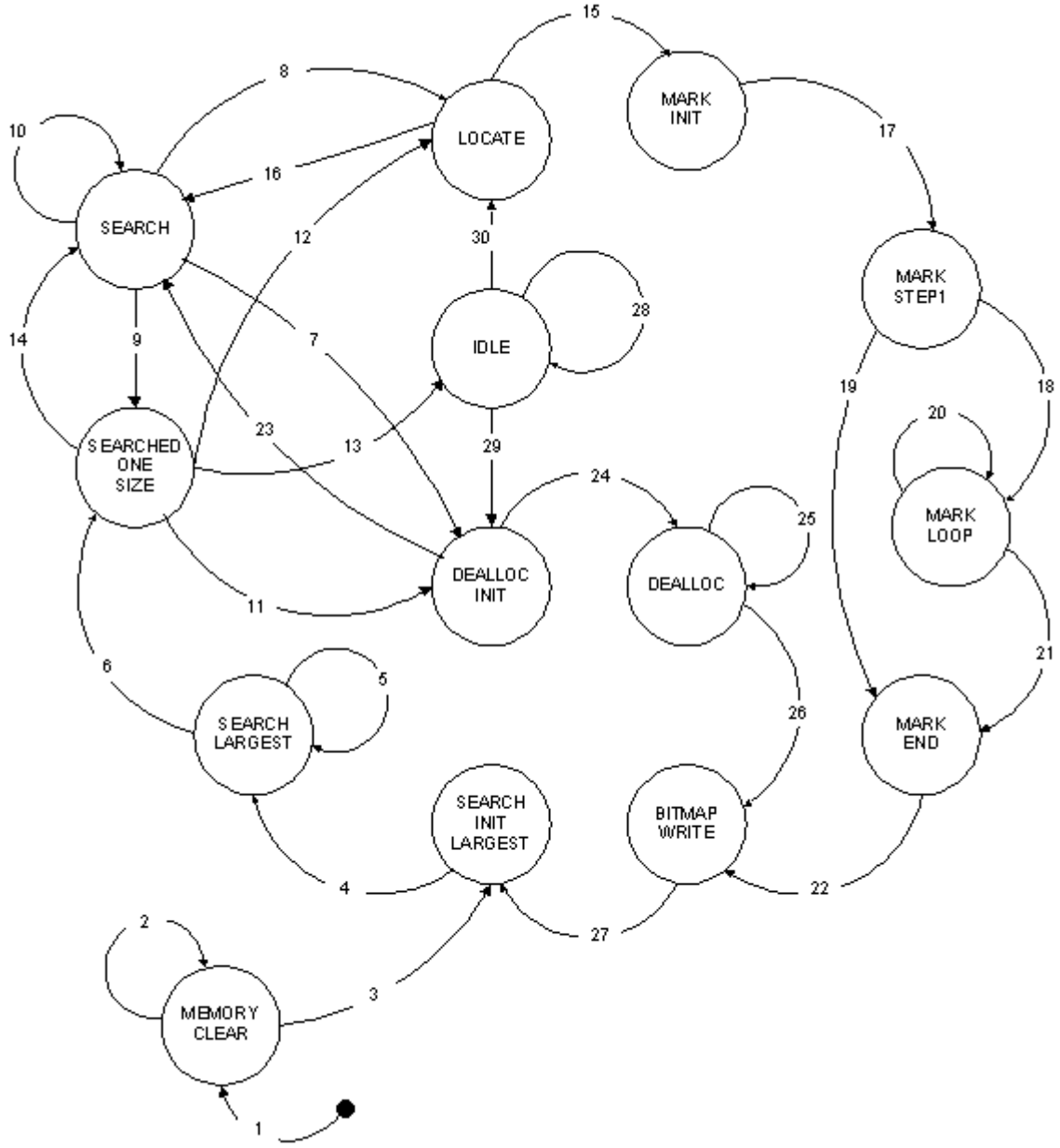


Figure 16: state machine for the service executor

Transition	Event	Action
13	If no new command arrives and all sizes have been searched for in all vectors.	If busy signal isn't low set it low. Go to state idle and wait for a new command.
14	If no new command arrives and there are sizes left to search for.	If busy signal isn't low set it low. If a free object is found in the current vector, store the address in the address buffer and mark it's size in the buffer bitmap. Load the next memory vector.
15	The locate unit found a free object in the buffer bitmap.	Set the done signal high. Clear the bit in the buffer bitmap, Set the read address to the address buffer equal to the locate_found_size.
16	The locate unit didn't find a free object in the buffer bitmap.	Set the out of memory signal. Set the wanted search size to the next highest.
17	No event required.	Set done flag low. Set out the read address to the vectors that should be marked first.

18	If the marking should take place in multiple vectors.	Calculate the number if blocks to the end of the vector, store the number in the size_reg. Store the remaining part of the size in a temporary size register. Set the vector_index and the signal Mark_size_mark_start to the mark unit. Update the read address to the next vectors.
19	The marking should take place in just one vector.	Set the vector_index and the signal Mark_size_mark_start to the mark unit.
20	The remaining marking should take place in multiple vectors.	Set the size_reg equal to the search width. Store the remaining part of the size in a temporary size register. Set vector index to zero and clear the Mark_size_mark_start signal. Update the read address to the next vectors. Write back the marked memory and size vectors.
21	The remaining mark operation should take place in just one vector.	Set the size_reg equal to the temporary size register. Set vector index to zero and clear the Mark_size_mark_start signal. Write back the marked memory and size vectors.
22	No event required.	Prepare to write back the last marked memory and size vectors.
23	The deallocation address is not valid.	Set the corresponding deallocation error, not allocated 01, or object align 10. Set the wanted search size to the next highest.
24	The deallocation address is valid	Set the done signal. and the clear_status_in signal. Set the vector index that the clear unit needs. Update the address to the next vectors.
25	If the deallocation should take place in multiple vectors.	Set vector index to zero and clear the status in and done signals. Update the read address to the next vectors. Write back the cleared memory and size vectors. Clear the buffer bitmap if coherence problem can occur.
26	If the clear_done signal is high or it is the highest vector, the clearing is done.	Clear the done signal and the clear_status_in signal. Prepare to write back the last cleared memory and size vectors.
27	No event required.	Write back the last marked or cleared vectors. Update the read address to the highest memory vector.
28	No new command from the interface.	Set the write enable signals to the address buffer and memory and size bitmaps so no writing is allowed. Clear busy, done, out of memory and deallocation error signals.
29	If a new deallocation command arrives.	Set busy signal high. Set out the address to the vectors that should be cleared first.
30	If a new allocation command arrives.	Set busy signal high. Store the wanted size to allocate in the size register.

Table 18: Service executor state machine transitions

6.4 Locate

6.4.1 Specification

The locate unit should look in the buffer bitmap for valid bits with an index equal to or larger then the wanted size. The bit is valid if it have a logical high value. If a valid bit is found, the index should be returned and the found signal go high. If no valid bit is found the found signal goes low. The index value indicates the size of a valid object. The unit should deliver the answer within one clock cycle.

The unit needs two generic parameters. The generic parameters are shown in Table 19. The signal names and their description in Table 20 should be used for the locate unit.

Name	Default value
Max_object_size_Bits	2
Max_object_size	4

Table 19: generic inputs for the locate unit

Name	Direction	Description
size (Max_object_size_Bits-1:0)	In	Give the wanted size to look for. The size equals the index in the buffer bitmap.
buffer_bitmap (Max_object_size-1:0)	In	A bitmap where 1indicates an available object of size given by the position in the bitmap. A zero indicates that no object of the particular sizes is available.
found	Out	1 indicates an available object has been found 0 no object of sufficient size can be found
found_index (Max_object_size_Bits-1:0)	Out	Gives the index to the found object. The signal only needs to be valid when found is high.

Table 20: Input and output signals for the locate unit

6.4.2 Construction

Because of the restriction that the answer should be delivered within one clk this unit is made totally combinatorial. The realisation of the unit is divided into a sequence of three different steps. The different steps are described in Figure 17.

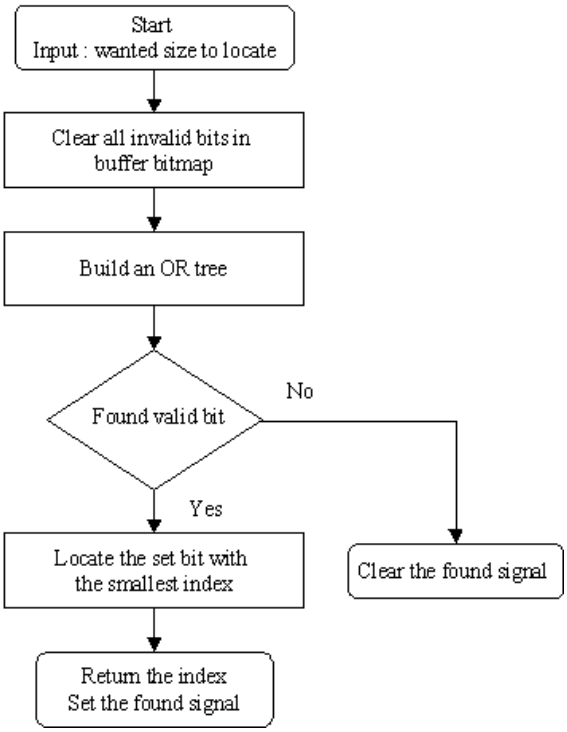


Figure 17: Flowchart of the locate unit

The invalid bits in the buffer bitmap are all bits with an index that are smaller then the wanted size to locate. The resulting bitmap from this operation is then delivered to the next step.

An OR-tree is built on the bitmap from the previous step, the tree nodes are stored in a vector. The number of bits in the vector are, the size of the bitmap multiplied with two and then subtracted with one. The bitmap values are stored in the lowest part of the vector. The connection between the OR-tree and the vector are described in Figure 18

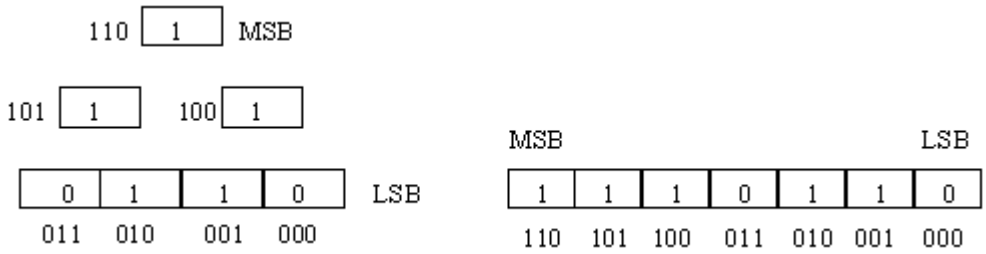


Figure 18: Connection between OR-tree and the vector

To indicate if a valid bit is found, the most significant bit in the vector is checked. If the bit equals one there exists a valid bit in the bitmap, otherwise the locate operation have failed and the found signal is cleared.

If at least one valid bit was found, the OR-tree are also used to determine the smallest index. To do this the OR-tree is traversed backwards until the index is found. The explanation for the traverse are made for the case when the bitmap are of size four, Figure 18. The same procedure are used independent of the size of the bitmap, the only difference are number of iterations that must be performed.

Start with the index to the highest bit in the vector, index 110. The value in index 110 comes from the index 101or 100, which only differs in the index LSB. If the LSB equals to zero the index belongs to the lower part of the bitmap, otherwise the index belongs to the higher part. Since it is interesting to find the lowest index with a valid bit, the lower part of the bitmap must be investigated first, i.e. the value in index 100 should be examined. This is accomplished by shifting the start index one step to the left and concatenate it with 0. If the value in this new index is equal to one, a step to the lower part is wanted and the index 100 is set as the new starting index. If the value is zero the shifted start index is concatenated with 1 to get a new start index in the higher part of the bitmap, in this case index 101. The procedure are then repeated until an index in the bitmap is found. The highest bit in the found index is not part of the final index.

As an example the Figure 18 is studied. The Or-tree have tree levels which give the fact that two iterations should take place.

1 Start index 110
Shifted 10 & 0
Value 1
New index 10 & 0

2 Start index 100
Shifted 00 & 0
Value 0
New index 00 & 1

Returned index 01

6.5 RAM memories

The three units that uses a RAM memory are, memory bitmap RAM, size bitmap RAM and the address buffer. The three memories should have the same behaviour and therefor only one type of memory is constructed, the different units are created when the hole system is mapped together. The only thing that differs between the units are the size of the memory, the size depend on the generic values for the block. The generic parameters that decides the size of the RAM memory are displayed in Table 21 and Table 22.

To allow both a read and a write operation during the same clock cycle the memory is constructed as a dual port RAM memory. The memory is constructed with asynchronous read and synchronous write. The signal names that should be used for the memory are shown in Table 23.

Name	Mapping	Default value
address_bits	Max_object_size_Bits	2
width	Block_address_Bits	10

Table 21: Generic inputs to address buffer

Name	Mapping	Default value
address_bits	Bitmap_address_Bits	7
width	Search_width	8

Table 22: Generic inputs to memory and size bitmap RAM

Name	Direction	Description
we_n	In	Sets the RAM write mode: 1= no update, 0 = update.
address_write(address_bits-1:0)	In	Address in the RAM that shall be written.
data_in(width-1:0)	In	Parallel data in to the RAM
address_read(address_bits-1:0)	In	Address in the RAM that shall be read.
data_out(width-1:0)	Out	Parallel data out from the RAM

Table 23: Input and output signals for the RAM memories

6.6 Mark

6.6.1 Specification

The mark unit should mark blocks in the memory and size vectors so it corresponds to an allocated object. To mark an object as allocated the bits in the memory and size vector should be set according to Table 26. The mark unit should return the new memory and size vectors in one clk. The unit is not responsible for detection of errors with the address that should be allocated or the size of the object. The generic inputs used by the mark unit are shown in Table 24. The signals that should be used by the unit are described in Table 25.

Name	Default value
Search_width_Bits	3
Search_width	8
Max_object_size_Bits	2

Table 24: Generic inputs for the mark unit

Name	Direction	Description
memory_vector (Search_width-1:0)	In	The part of the memory bitmap in which the object to be marked is located.
size_vector (Search_width-1:0)	In	The part of the size bitmap in which the object to be marked is present.
vector_index (Search_width_Bits-1:0)	In	The start index in the vectors where marking should begin.
size (Max_object_size_Bits-1:0)	In	The size of the object to be marked.
size_mark_start	In	Indicates if it is the first vector to mark in, 1 if first vector to mark in else 0
memory_vector_new (Search_width-1:0)	Out	The new memory_vector after marking
size_vector_new (Search_width-1:0)	Out	The new size_vector after marking.

Table 25: Input and output signals for the mark unit

Memory	Size	Interpretation
0	0	Not an allocated object
0	1	Should not happen, illegal combination
1	0	An allocated object continue
1	1	Start of an allocated object

Table 26: Combinations of the memory and size vectors

For the size signal it should be noticed that a value of zero indicates that one block should be marked.

For the mark unit to know where the marking should start an index to the vectors shall be used. The mark operation should then continue until the object is completely marked.

Both the memory and size vector must be marked all of the size, this is necessary so the problem with memory leakage is avoided. The memory leakage problem is described in chapter 5.1.7.

If the marking should take place over several vectors, the size_mark_start signal should be used so the mark unit doesn't mark the first bit in the size vector, when marking in other vectors then the vector where the object starts.

6.6.2 Construction

To meet the response demand of one clk the mark unit is constructed as a combinatorial unit. The flowchart in Figure 19 was used to implement the mark unit.

To determine when all of the object is marked the start index and the size of the object is used. If the sum of the index and the size are smaller then or equal to the temporary index counter, the marking continues.

The check if the current vector is the first one is done with the size_mark_start signal. If the signal is high it is the first vector and the size vector is marked at the start index. This give the vector combination 11 at the start index. If the signal is low the size vector is cleared for the hole size.

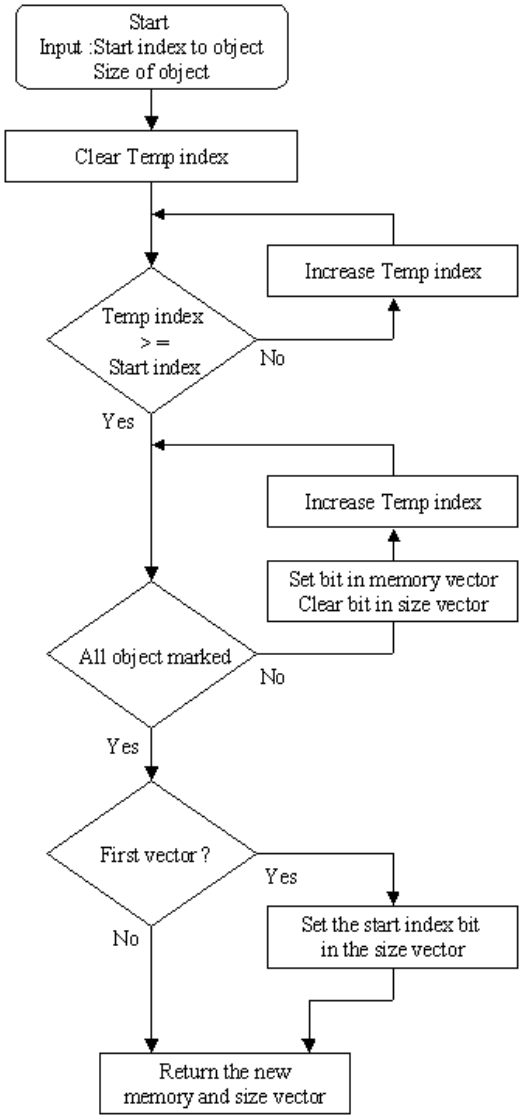


Figure 19: Flowchart of the mark unit

6.7 Clear

6.7.1 Specification

The clear unit should clear objects in the size and memory vectors, i.e. the bits in the vectors should be cleared. This operation should be performed in one clk. The unit is not be responsible for detection of errors with the deallocation address. The generic inputs used by the clear unit are shown in Table 27. The signal that should be used by the unit are described in Table 28.

Name	Default value
Search_width_Bits	3
Search_width	8

Table 27: Generic inputs for the clear unit

Name	Direction	Description
memory_vector (Search_width-1:0)	In	The part of the memoryB bitmap in which the object to be cleared is marked.
size_vector (Search_width-1:0)	In	The part of the sizeB bitmap in which the size of the object to be cleared is encoded.
vector_index (Search_width__Bits-1:0)	In	The start index in the vectors where clearing should begin.
status_in	In	If the object to clear occupy bits in several memory vectors this signal should be high when the first vector is loaded, else it should be low.
done	Out	The clearing is done.
clear_bitmap	Out	Set high if the buffer bitmap should be cleared.
memory_vector_new (Search_width-1:0)	Out	The new memory_vector after clearing
size_vector_new (Search_width-1:0)	Out	The new sizeB vector after clearing.

Table 28: Input and output signals for the clear unit

For the clear unit to know the size of the allocated object it need the information that the combinations of the memory and size vector give, Table 29. The clear unit should use the index signal to know the start of the object, and make a clear operation as long as the vector combination 10 appears. If the combination 00 or 11 combination appears, no more clear operations should take place on the vectors. If the illegal combination 01 occurs the unit should set the memory bit high to achieve the 11 combination. The done signal should be low for the combination 10, for all the other combination the done signal should be high.

If the object that have been cleared are followed by a free object, the signal to clear the address buffer bitmap should be set high. This happens if the clearing is stopped by the combination 00.

Memory	Size	Interpretation
0	0	Not an allocated object
0	1	Should not happen, illegal combination
1	0	An allocated object continue
1	1	Start of an allocated object

Table 29: Combinations of the memory and size bitmaps

If the object to be cleared stretches over several vectors the status_in signal should be used. The signal is high if the current vector is the one where the object have its starting address.

6.7.2 Construction

To meet the demand for a response time of just one clk the unit is combinatorial. To implement the unit the flowchart in Figure 20 was used.

To prepare the size vector the signal status_in is used. If the signal is high then the size vector is cleared at the start index position, this operation produce the vector combination 10. This allows the unit to clear the memory vector at the start index. Without the clearing the original size vector is used and the unit detects the vector combination 11, i.e. the clearing is finished. If there are indexes left when clearing is done, the unit does not perform any action on the memory vector, the vector combinations are ignored.

If the vector combination is 10 at the last index the clearing is not finished and the done signal is low. In this case the unit will receive new parameters, where the status_in is low.

If some of the other combinations are detected the appropriate action is taken.

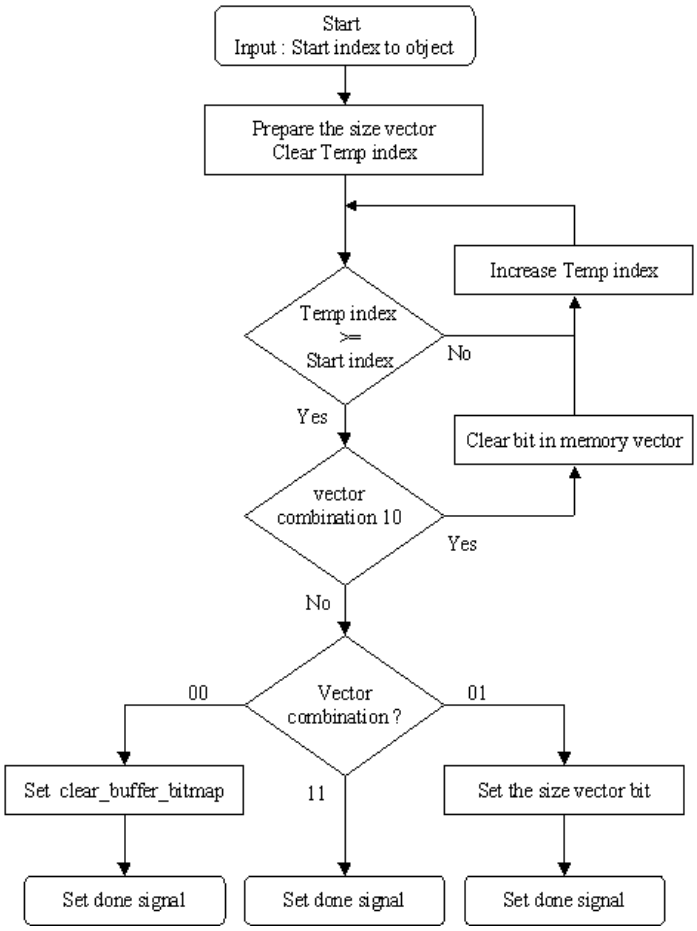


Figure 20: Flowchart for the clear unit

6.8 Search

6.8.1 Specification

The search unit should search for free objects in the memory vectors provided by the service executor. The object to search for can be of sizes equal to or smaller then the wanted size. This means that if no free object of the wanted size is found, the search unit should be able to pick up and return a smaller size then wanted. This operation should be performed in one clk. The generic inputs used by the search unit are shown in Table 30 and the signal used by the unit are described in Table 31.

Name	Default value
Search_width_Bits	3
Search_width	8
Max_object_size_Bits	2
Max_object_size	4

Table 30: Generic inputs for the search unit

Name	Direction	Description
memory_vector (Search_width-1:0)	In	The part of the memoryB bitmap in which a search should be performed.
search_size (Max_object_size_Bits-1:0)	In	The size to search for.
old_counter (Max_object_size_Bits:0)	In	Tells how many contiguous zeros there where at the end of the previous memory vector. Used to find free objects crossing vector borders.

found	Out	Set high if a free object of search_size or smaller is found else low
found_in_old	Out	Set high if a free object started in the previous memory vector is found. i.e. the first bit in the new memory vector is one and no larger objects are found in the current vector.
found_index (Search_width_Bits-1:0)	Out	Give the index to the start of the found object in the memory vector. Only valid if found is high.
found_size (Max_object_size_Bits-1:0)	Out	Give the size of a free object if one is found. Valid only if found or found_in_old is high.
current_counter (Max_object_size_Bits:0)	Out	Tells how many contiguous zeros there where at the end of the memory vector. Used to find free objects crossing vector borders.

Table 31: Inputs and output signals for the search unit

If a free object of a size that equals to search_size or smaller is found, the found_size and found_index should be set to the corresponding values of the object in the memory vector. The largest of the found object should be returned, if two objects of the same size is found the object with the lowest index should be chosen.

If the search for free objects stretches over several vectors the unit receives a new vector each clk. In this case the old_counter signal should be used to continue counting the size of an object starting in a previous memory vector. The new vectors are loaded from the top of the memory bitmaps down to the bottom, so the previous vector contain higher memory addresses than the current vector. If an object is found and it reside in the previous memory vector the signal found_in_old should be set high. The object is defined to reside in a previous memory vector if the most significant bit in the current vector equals to one, and the old_counter value is the largest found size. The two different found signals should never be set at the same time. The signal current_counter should hold the number of contiguous zeros at the end of the memory vector. This two counter signals should make it possible to search for free objects over vector boundaries.

6.8.2 Construction

To meet the demand for a response time of just one clk the unit is combinatorial. To simplify the request that the lowest index should be chosen the memory vectors are searched from the MSB down to LSB. To implement the unit the flowchart in Figure 21 was used.

The unit starts its operation by copying the old counter, set the temp size to zero indicating that no object have been found. It also set the index so it starts at the most significant bit in the memory vector. If the indexed bit in the bitmap is zero and the size counter has not reached the maximum object size, the size counter is increased. If the indexed bit have a value of one, the size counter is always cleared. The action taken before the counter is cleared depends on the condition of the counter.

The first condition that must be fulfilled for any further action is that the size counter are smaller than the wanted size. For the condition found in old, the index must be the highest in the vector and the size counter not equal to zero.

The purpose of the temp size signal is to deny smaller objects to overwrite larger, but an object of a size equal to the previously largest is made the new found object. This allows the unit to meet the demands that the largest of the found object should be returned, and that the object with the lowest index should be chosen if two objects of the same size is found.

The operation continue until all indexes have been investigated, it then sets the current counter signal equal to the size counter. It also set the found signals according to the object that was found, if no object was found the two found signals are low. The signals, found index and found size, are always set to a value, it is up to the service executor to decide if the values should be stored or not.

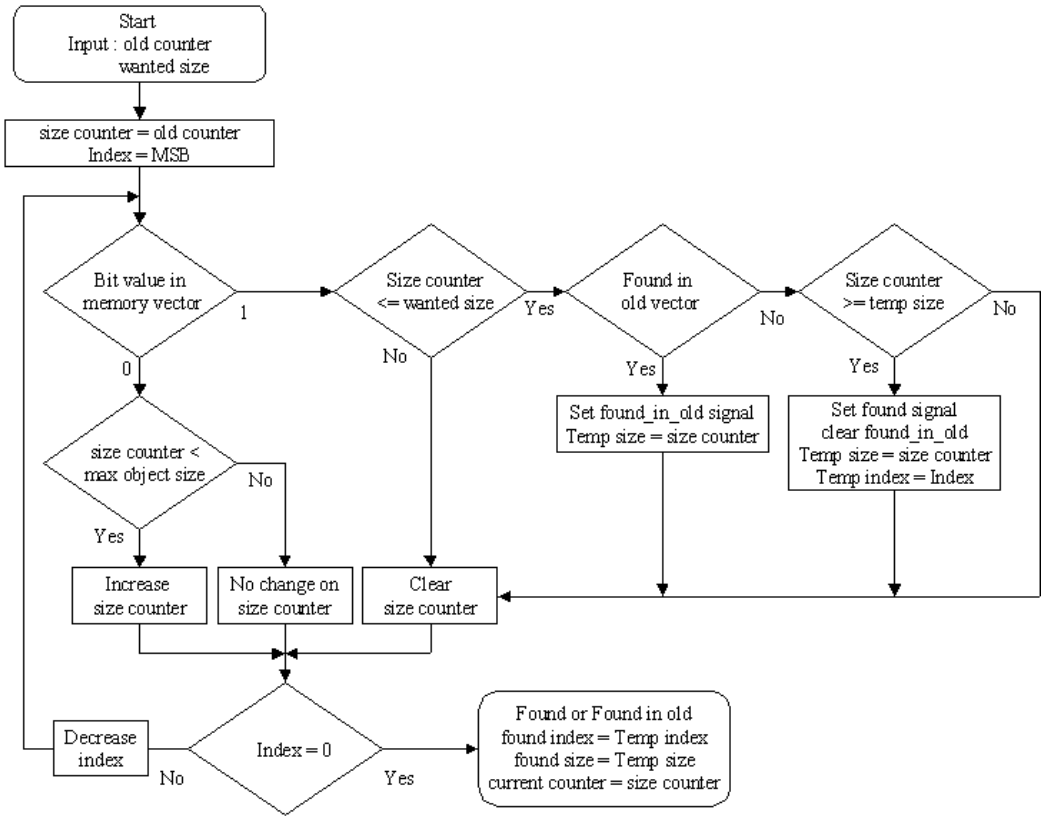


Figure 21: Flowchart for the search unit

6.9 Top level

The nine different components defined by the DMA architecture in chapter 5 are mapped in the top level , using structural VHDL. The mapping signals between the components follows a name convention which simplifies the understanding. The name convention is shown in Figure 22.

The Call and RDV signals works as a handshake communication between the components. The call signal is constructed as follows, Call _receiving component_ sending component. The respond signal is constructed as RDV_ sending component_ receiving component. The data signals follow the same name convention as the call signals, giving Data_ receiving component_ sending component. In some cases an explanation is put last in the signal name to improve the readability.

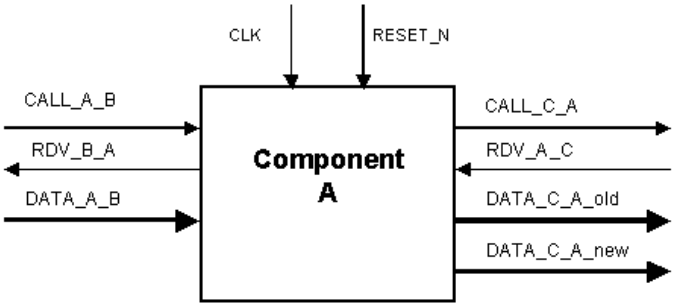


Figure 22: Name convention for signals between components

All the mappings between the components and their names can be read in appendix [1]. In some cases the receiving component name is defined by a X, which means that more than one component uses the signal.

6.10 Testbench

An extensive testbench for verification of the DMA functionality where developed. The testbench interfaces the DMA and request services from it by the use of a commando file. This makes it easy to perform large allocation/deallocation test. Running the testbench will produce an output file. For each command in the command file there will be a corresponding output displaying the result of the command. Timing is verified by simulation graphs. The syntax used in the command file is as follows.

Command	Parameter	Description
a	s (byte)	allocation of s bytes
d	address -1 -2 –3	deallocation of supplied address deallocation of last allocated object deallocation of first allocated object deallocation of random allocated object
o	wanted offset in bytes	Set the offset to the dynamic memory space.
f	number of loops	the next command, a or d, (must be a or d) will be carried out number of loops time.
e	None	Sets read of error register on or of, 1 = read, 0= no read, default is read.
p	None	Prints the current allocated addresses
#	None	line comment, the rest of the line will be ignored, must be followed by a space

Table 32: Testbench command file syntax

The command and the parameter should be separated by a space. Empty lines are not allowed. When giving a deallocation command, the memory offset will be added automatically in the testbench to a supplied address.

6.11 Verification

The DMA functionality has been verified by the use of an commando file. The command file and the resulting output is given in appendix [5]. Verification of certain key implementation details is done by examine simulation graphs generated by the testbench.

Examinations of the DMA concerning error detection and service call protocol has been performed with the settings of Table 33. Examinations of the units inside the DMA have been performed with the settings of Table 34.

Name	**Value**
Memory_space	256
Address_space_Bits	16
Block_size	2
Max_allocation_size	32
Search_width	8
Base address to DMA	7F0040

Table 33: The DMA settings used in verification of error detection and service call protocol

Name	**Value**
Memory_space	256
Address_space_Bits	16
Block_size	8
Max_allocation_size	32
Search_width	2
Base address to DMA	7F0040

Table 34: The DMA settings used in verification of the DMA units

6.11.1 Error detection

To verify that the DMA detects all types of errors as it should the following command file where used.

```
#set offset
o 4608
#allocated zero bytes
a 0
#allocate to large size
a 36
#OK allocation
a 32
#deallocate address 2
d 2
#deallocate address 8
d 8
#deallocation of not allocated address
d 128
#deallocate address 300
d 300
#fill memory
f 7
a 32
#allocation out of memory
a 32
```

This command file should generate all possible errors that might occur, except unrecognised command error. The output when running the testbench with the command file is as follows :

```
Starting Test
-------------------------------------------------
command #1
Allocation of :0 bytes
  - Allocation service service returned with error :
    type :2 , Allocation of zero size
-------------------------------------------------
command #2
Allocation of :36 bytes
  - Allocation service service returned with error :
    type :1 , Size error
-------------------------------------------------
command #3
Allocation of :32 bytes
  - Allocation service performed OK
   Address: 4608
-------------------------------------------------
command #4
Deallocation of address:2
  - Deallocation service returned with error :
    type :6 , Block align error
-------------------------------------------------
command #5
Deallocation of address:8
  - Deallocation service returned with error :
    type :4 , Object align error
-------------------------------------------------
command #6
Deallocation of address:128
  - Deallocation service returned with error :
    type :3 , Deallocation of not allocated address
```

----
command #7
Deallocation of address:300
- Deallocation service returned with error :
type :5 , Range error
----
command #8
Allocation of :32 bytes
- Allocation service performed OK
Address: 4640

.
.
.
----
command #14
Allocation of :32 bytes
- Allocation service performed OK
Address: 4832
----
command #15
Allocation of :32 bytes
- Allocation service service returned with error :
type :0 , Out of mem

As one can se all types of errors are detected.

6.11.2 Service call protocol

Figure 23 shows the DMA interface unit behaviour when performing an allocation service. This behaviour follows the specification in chapter 4.4.

Timing requirement formulas
Using the timing restriction formula for allocation, chapter 6.12, the applied DMA setting should have a timing restriction of 27 clock cycles. The simulation shown in Figure 23 shows the actual timing requirement when simulating the DMA. The timing requirement is the same as predicted by the formula.

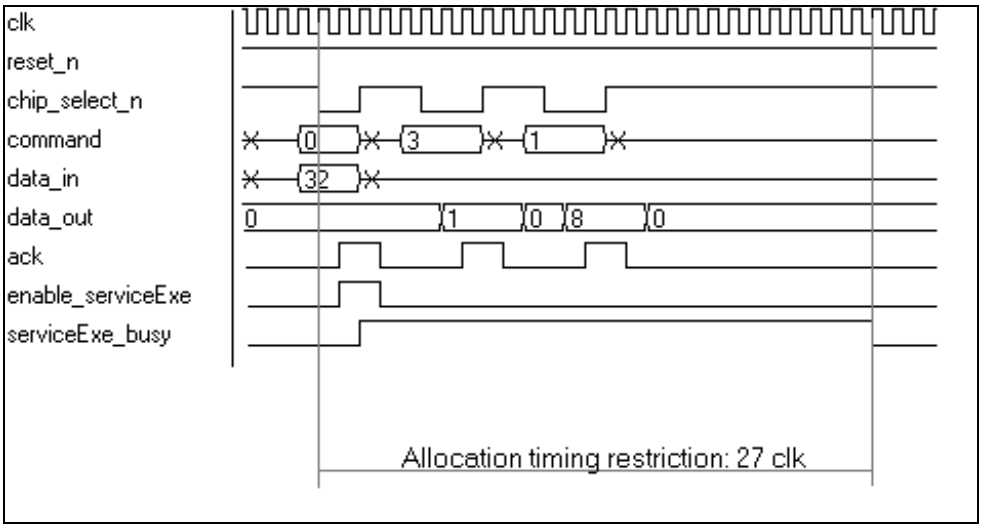


Figure 23: Simulation showing the timing restriction for allocation

6.11.3 Interface

Figure 24, shows the interface unit in/out-put signals when performing two allocations (8 and 32 byte) and a deallocation of the 32 byte object.

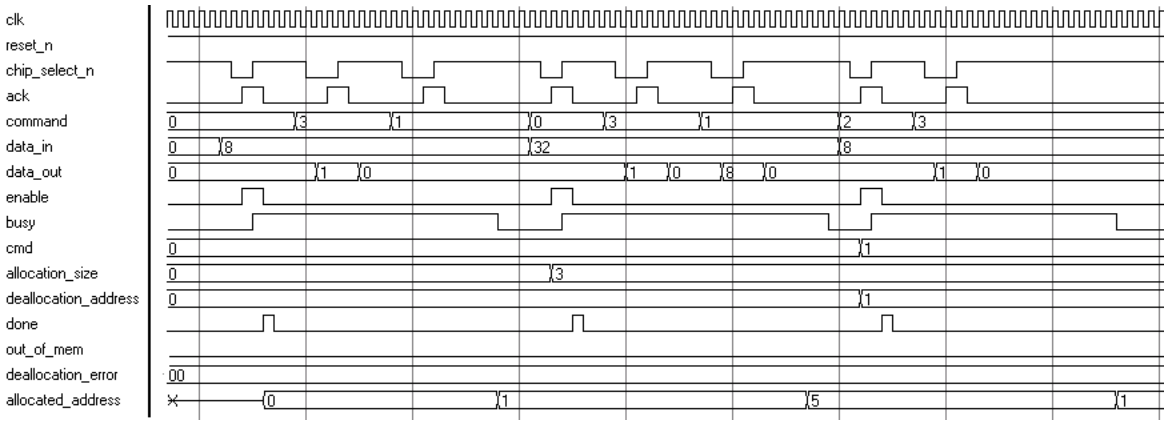


Figure 24: Simulation graph showing interface unit signals

Observe that the allocated_address signal is in block addresses, the interface scales it with the block size before returning it on the data_out signal (no offset is used). Also notice that the allocation size is scaled down to block sizes and that allocation of one block gives an allocation size of zero.

6.11.4 Mark

The Mark simulation graphs shows how the mark unit respond when the DMA is performing an allocation of 8 byte (1 block), followed by another allocation of 32 byte (4 block). No memory is allocated before the first allocation so the marking will begin at the beginning of the first vector.

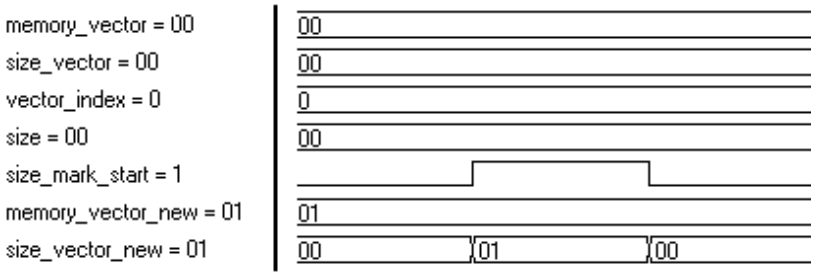


Figure 25: Marking first allocation

When size mark start goes high the mark unit marks in the size and memory vector where the object starts.

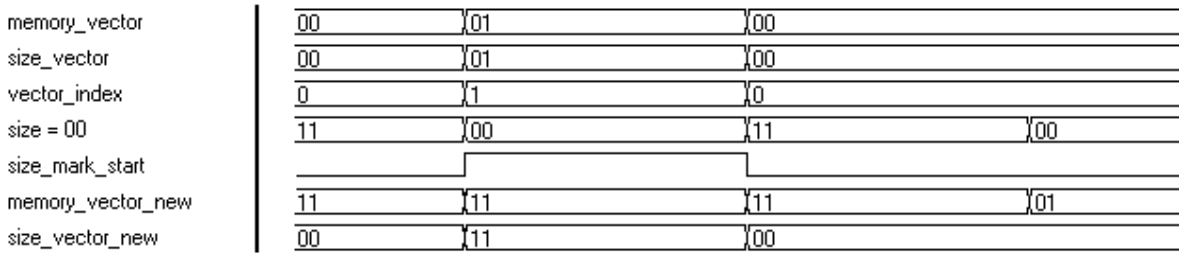
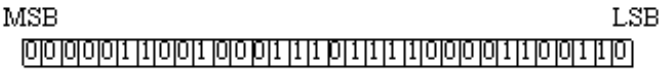


Figure 26: Marking the second allocation

With the second allocation the marking is over three vectors. The service executor feed the mark unit with the vectors to mark in. Size mark start is high during marking in the first vector of the allocation. It is only in this vector the size is marked. The mark unit is always active and always produces result. It is up to the service executor to set the input signals and use the output at appropriate times.

6.11.5 Search

The search simulation graph, Figure 27, shows a search for the largest object (4 blocks) in the following memory bitmap :



There are tow bits per vector in the bitmap (Search width = 2). The service executor feeds the search unit with parts of the bitmap, starting with the MSB vector.

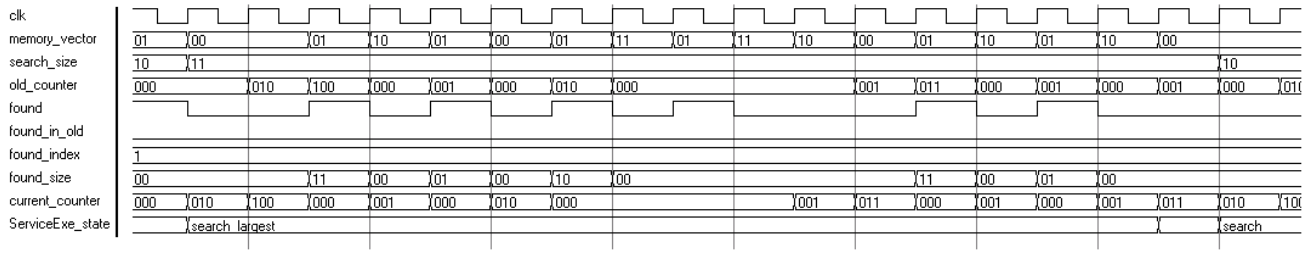


Figure 27: Simulation showing search behaviour

The search unit is always active and always produces result. It is up to the service executor to set the input signals and use the output at appropriate times.

6.11.6 Locate

The locate simulation graph, show location of an object of blocksize 1, when the buffer bitmap says that the only available objects are of sizes 3 and 4 blocks, so the 3 block object will be located.

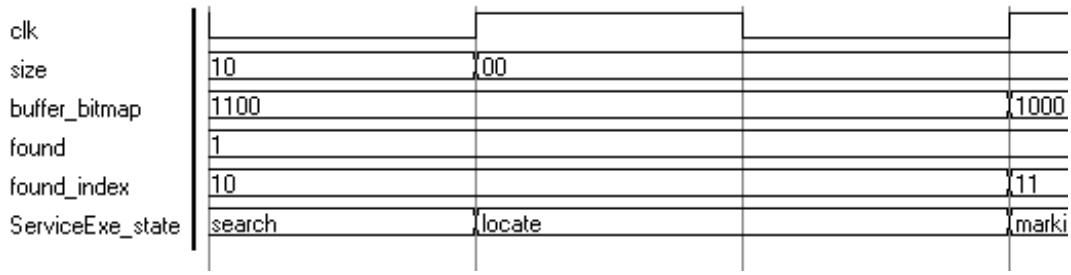


Figure 28: Locate simulation graph

The locate unit is always active and always produce results. It is up to the service executor to set the input signals and use the output at appropriate times.

6.11.7 Clear

Figure 29 shows the clearing unit activities when deallocation of the second allocation for the allocation simulation graph, Figure 26. The service executor will give the clearing unit the three vectors associated with the object the should be deallocated, starting with the lsb vector. The unit signals through the clear bitmap signal that the buffer need to be cleared as otherwise a coherence problem in the address buffer might occur. The done signal from the clear unit goes high when the object are completely cleared.

The clear unit is always active and always produces result. It is up to the service executor to set the input signals and use the output at appropriate times.

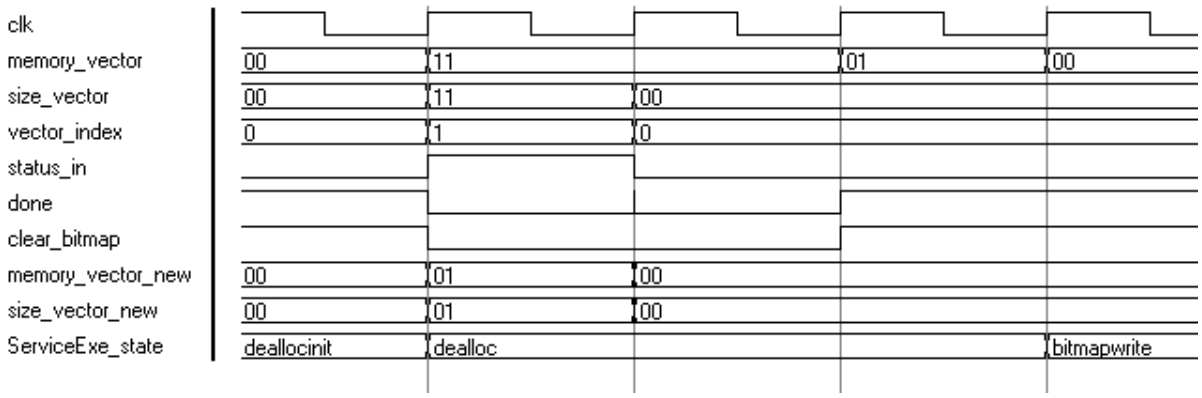


Figure 29: Simulation graph for clearing

6.12 Timing restrictions

The DMA has a predictable behaviour as long as there is any available memory space left to do allocations in. This means that an allocation or deallocation will always be carried out in a specific time frame. This time is dependent on the generic settings of the DMA. The unit has a fast response time when a service request is given to it. Meaning that the answer to a service request will be available within a few clock cycles. However between two service calls the unit must do some background work (searching, marking, etc.) which will take some time. This time, called minimum timing requirement, is dependent on the generic inputs. The time is proportional to the maximum number of blocks that can be allocated and the total amount of dynamic allocable memory. It is inversely proportional to the number of blocks that are searched at a single clock cycle and the number of bytes per block. The minimum timing requirement formulas that will be derived below, will describe the time from a start of a service request to a start of the next service request.

Definition of the parameters used to calculate the worst timing behaviour between two service requests.

Memory_space	m = number of bytes in the allocable memory
Block_size	b = the number of bytes in each block [byte / block]
Max_allocation_size	a = max number of bytes that can be allocated each time [bytes]
Search_width	s = the number of blocks that are searched each time [block]

6.12.1 Start up time

The DMA can not be access directly after reset. There is a set up time for the internal memory bitmaps.

This time is :

$$T_{Startup} = Z + S \quad [clk]$$

Where :

$Z = m / (b * s)$, time to clear the memory bitmaps

$S = 2 + m / (b * s)$, time to do a search for a object of maximum size

Which gives :

$$T_{Startup} = 2 + (2 * m) / (b * s) \quad [clk]$$

Karl Ingström
kim99001@student.mdh.se

Anders Daleby
ady99002@student.mdh.se

6.12.2 Service request response time

The service request response time is fast and independent on the generic inputs i.e. it is constant. The time represent how fast a service request is taken care of and how long time it takes until any return to the user can be accessed, i.e. the interface holds the result.

The service request response time is :

$T_{ResponseAllocation}$ = One write and two read access [clk]
= 15 clk if the GBI bus protocol is strictly followed.

$T_{ResponseDeallocation}$ = One write and one read access [clk]
= 10 clk if the GBI bus protocol is strictly followed.

6.12.3 Minimum timing requirement

Allocation service
The minimum timing requirement is calculated with the formula below.

$$T_{minAlloc} = I + L + M + S \quad [clk]$$

Where the parameters, M and S depends on how the generic inputs are chosen. While the I and L have constant values.

I = 2 , time to interface the unit and get the service executor going
L = 1 , time to locate a suitable block address

M = 4 + x , time to mark object as allocated
where
x is the maximum number of vector borders that can be passed during marking

$$x = \begin{cases} 1 & \text{if } 1 < (a/b) < s \\ \lfloor a/(b * s) \rfloor & \text{else} \end{cases}$$

S = 2 + m / (b * s) , time to do a search for a object of maximum size

which gives : $T_{minAlloc} = \mathbf{9 + x + m / (b * s)}$

Examples of the minimum timing requirement for allocation of various configurations are given in appendix [3].

Deallocation service
The minimum timing requirement is calculated with the formula below.

$$T_{minDealloc} = I + C + S \quad [clk]$$

In this case the parameter I have a constant value, while C and S are dependent on how the generic inputs are chosen.

I = 2 time to enable the deallocation command
C = 3 + x time to clear the object
where
x is the maximum number of vector borders that can be passed during clearing

$$x = \begin{cases} 1 & \text{if } 1 < (a/b) < s \\ \lfloor a/(b * s) \rfloor & \text{else} \end{cases}$$

S = 2 + m / (b * s) time to do a search for a object of maximum size

which gives $T_{minDealloc} = \mathbf{7 + x + m / (b * s)}$

7 Test system

For evaluation of the DMA performance a test system was developed. The test system consisted of :

- A FPGA chip (XILINX Spartan XCS40XL) to which the DMA was downloaded. The FPGA also holds a timing circuit for making time measures.
- Evaluation board (M68KEV Board Computer)
- MCU-card (Motorola 68322 Card-Computer). On which a test application and a software allocator / deallocator was executed.
- PC for printing results of measurements.
- Serial-card (RS232), For communications between the MCU and the PC.

7.1 Hardware development tools

The following tools where used :

- LeonardoSpectrum v2001_1d.45, Exemplar Logic, Inc.- Synthesis
- ModelSim EE/Plus 5.2e, Mentor Graphics - Simulation
- Xilinx design manager, Xilinx Inc. - Place & Route

7.2 Software development tools

All development tools, editor, assemblator, linker, make and debugger are shareware programs. On the evaluation board there is a MCU, RAM, serial port, and in ROM a so-called monitor. The monitor is a program that can execute certain commands that it receives by a serial port. With help of the monitor, the debugger can download and debug the application. The debugger sends commands to the monitor that performs them and sends back the result of the command. The board is also connected to a terminal program on the PC, Minicom, through another serial port. Which enables port printouts from the application.

7.3 Test system architecture

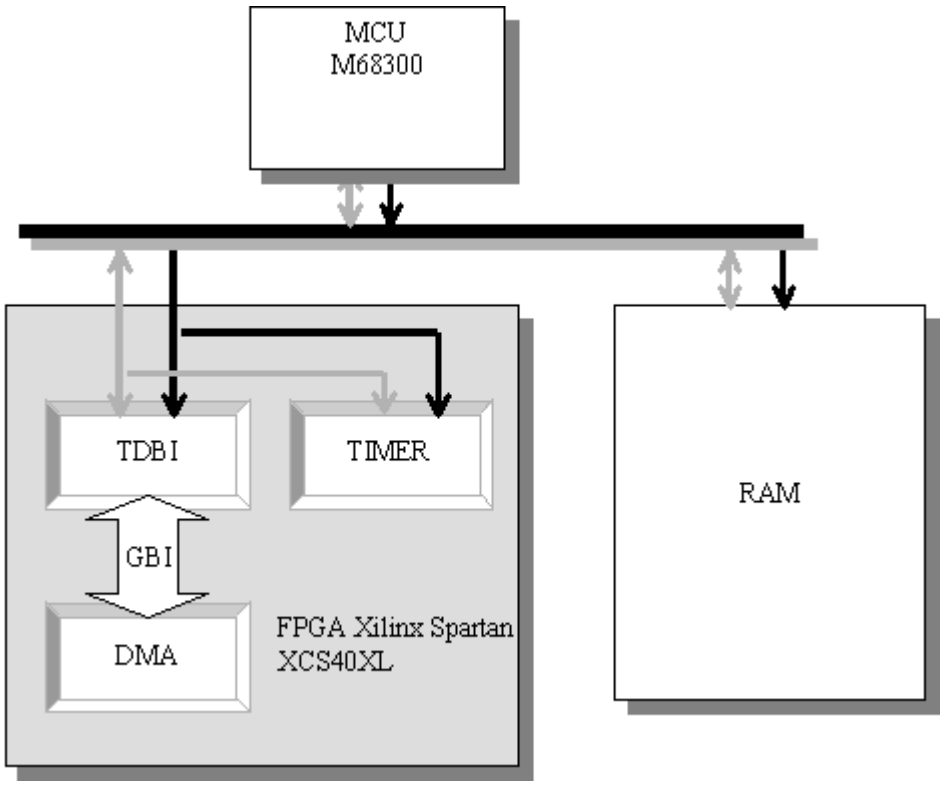


Figure 30: Test system architecture

7.3.1 The DMA settings

In the test system a total of 8192 bytes is set aside for dynamic allocations. Objects sizes from 8 byte to 32 byte are used. The search width parameter was set to 8 block. Using the formula for the minimum allocation timing requirement between to sequential service requests, 6.12, gives :

a = 32 b= 8, s = 8, m = 8192

=>

$T_{minAlloc} = 9 + 1 + 8192 / (8 * 8) = 138$ clk

The FPGA runs at frequency 10 MHz which gives $T_{minAlloc} = 138 * 1 / 10\ MHz = 13.8\ \mu s$. This is the minimum timing requirement between two allocations that must be met. In the test system the total device driver time was 21,8µs so no special care for fulfilling the time requirements was needed. The device driver was constructed according to Figure 7 and Figure 8.

7.3.2 MCU

Documentation about the MCU can be found in [1], M68300 Family - MC68332,User's Manual, Motorola 1996.

7.3.3 Memory space map

When using the DMA two things need to be taken care of from the software development viewpoint, where in the address space should memory for dynamic allocations be set aside (DMA_memory_offset) and at which addresses should the DMA commando registers be placed (DMA_base_address). The DMA base address is decoded in the TDBI. The DMA memory offset is set by the software and can hence be set to the DMA during system initialisation.

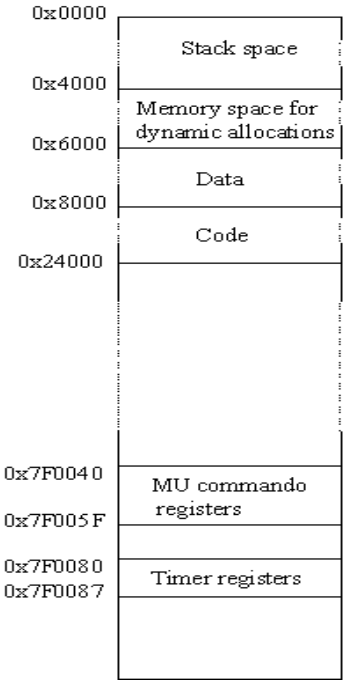


Figure 31: An address space map showing the address locations for the DMA in the test system

7.3.4 Target Dependent Bus Interface (TDBI)

The TDBI is responsible to map the DMA to the address space of the system so a command can be given to the DMA. The TDBI also translates commands and parameters to the DMA to support the GBI. The input and output signals to the TDBI is shown below.

Name	Direction	Description
clk	In	System clock
reset_n	In	Reset signal (active low)
address_bus(23:0)	In	The address of the word to be transferred during the bus cycle. The MCU place the address on the bus at the beginning of a bus cycle. The address is valid while as_n is logical low.
data_bus(15:0)	InOut	A bi-directional parallel bus that transfers data to or from the MCU. During read cycle the data is latched to the MCU on the last falling edge of the clock for that bus cycle. The MCU places the data on the data bus one half clock cycle after as_n is set low in a write cycle.
as_n	In	Address strobe. A timing signal that indicates the validity of an address on the address bus. It is set low one-half clock after the beginning of a bus cycle.
ds_n	In	Data strobe. A timing signal. For a read cycle, the MCU sets ds_n low to signal that an external device can place data on the bus. ds_n is asserted at the same time as as_n during a read cycle. for a write cycle, ds_n signals that data on the bus is valid. The MCU sets ds_n low one full clock cycle after the assertion of as_n during a write cycle.
we_n	In	Read/write signal. Determines the direction of a transfer during a bus cycle. The signal change state, when required, at the beginning of a bus cycle. The signal is valid when as_n is set low. we_n only change when needed i.e. if a write cycle is followed by a read cycle or vice versa. 1 = read, 0 = write.
size(1:0)	In	indicates the number of bytes remaining to transfer during an operand cycle. They are valid while the as_n is set low. size(1) |size(0) | transfer size 0 | 1 | byte 1 | 0 | word 1 | 1 | 3 byte 0 | 0 | long word
dsack_n(1:0)	Out	Data and size acknowledge signals. During a read cycle these signals tell the MCU to terminate the bus cycle and to latch data. During a write cycle the signals indicate that the data has been received correctly. In the case of word to 16-bit port they should be assigned 10.
berr_n	Out	Bus error signal. Should be set low if an bus error is detected in an access to the unit. See [1] for bus error explanation
command(3:0)	Out	Give commands to the DMA
data_in(Address_space_Bits-1:0)	Out	command parameters to the DMA
data_out(Address_space_Bits-1:0)	In	command return values from the DMA
chip_select_n	Out	Chips select (active low), activate detection of commands
ack	in	Acknowledge of received command

Handshake protocol between MCU and TDBI
Follows the protocol as described in [1], M68300 Family - MC68332,User's Manual, Motorola 1996.

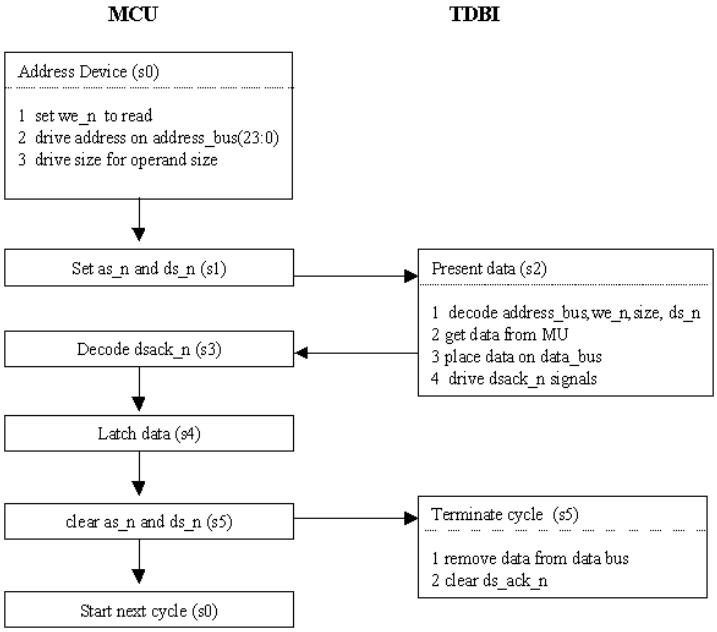


Figure 32 Word read cycle flowchart

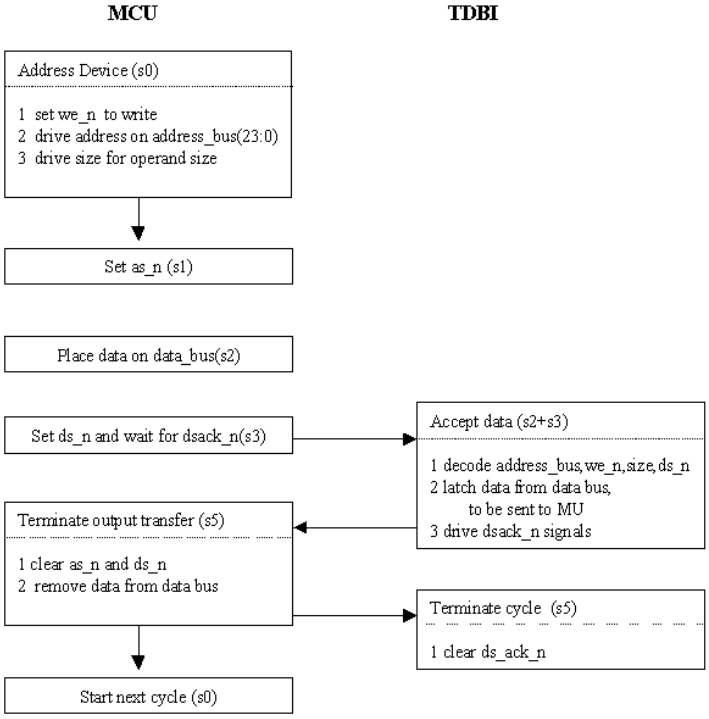


Figure 33: Write cycle flowchart

DMA command registers
The DMA supports a set of seven commands, start allocation of size X, start deallocation of address Y, read status register, get result from allocation, read the error register, set the memory offset and get the memory offset. Allocation and deallocation are made up of a sequence of these commands.

The commands given to the DMA, see chapter 4.4.1,can not be directly mapped to the system bus. The TDBI will provide a mapping. That is the TDBI will provide an address to command register mapping. Each command register will have it's own word aligned address on the system bus as summarised in Table 35. All accesses to the DMA in the test system will use word accesses meaning that the parameter/return value when writing/reading to a command register should be of word size.

Command	Mode	Address Offset	Parameter / Return depending on mode
Start Allocation	W	$00	Wanted size in bytes
Get allocation result	R	$02	Allocated address
Start deallocation	W	$04	Deallocation address
Read status register	R	$06	The DMA status register
Read error register	R	$08	The DMA error register
Set memory offset	W	$0A	Sets the DMA memory offset
Get memory offset	R	$0C	Reads the DMA memory offset

Table 35: DMA commando register with address offsets

7.4 Timer Module

The purpose of the timer module is to provide a stop watch feature. This is used for time measurements in the test application. The timer module will take four commands, start, stop, read high, read low, which are given in the same fashion as commands to the TDBI. The timer consists of a 32-bit counter which is increased each clock cycle when running. The base address used for the timer in the test system is 7F0080 hex. To access the 32 bit timer value two word accesses is used. One for reading the high order bytes and one for reading the low order bytes.

Command	Mode	Address Offset	Parameter / Return	Description
START	R/W	$000	None	Will start the timer, continuing from the current time value.
STOP	R/W	$010	None	Stops the timer
READ_HIGH	R	$100	The high order bytes of the current time value.	Used to read the high order bytes of the current time value.
READ_LOW	R	$110	The low order bytes of the current time value.	Used to read the low order bytes of the current time value. Also sets the current time value to zero.

7.5 The software allocator

The software allocator developed for comparison with the DMA uses a first fit strategy, with a bitmap memory representation. Each bit in the bitmap represents 1 byte, but the bitmap is searched in groups of 8 bits giving a minimum block size of 8 byte. The search always starts at the beginning of the bitmap. A size bitmap is used to encode the size of the allocated objects in a similar fashion as used with the DMA. This allocator is in now way any attempt to be an "optimised" allocator.

Figure 34 gives a flow chart of the software allocator and deallocator.

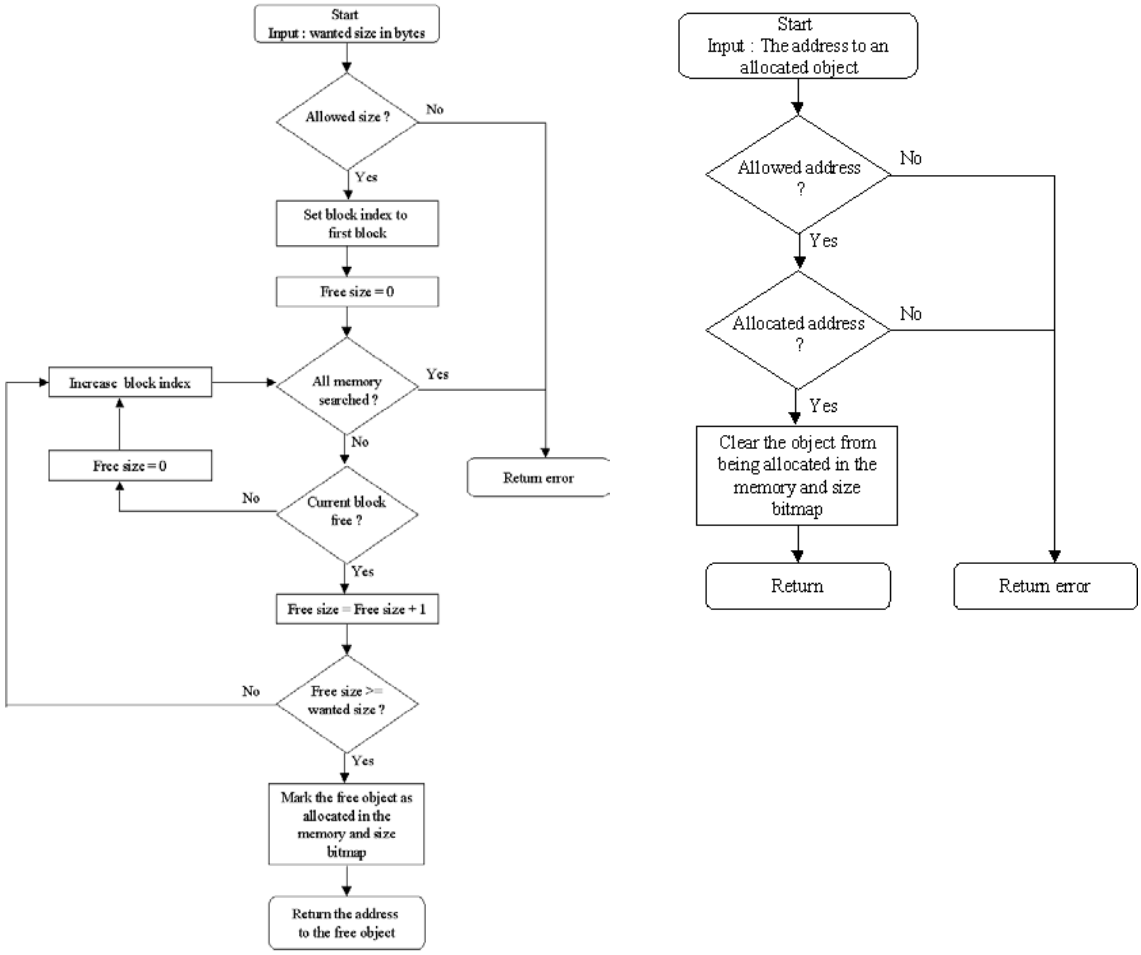


Figure 34: Flow chart of the software allocator and deallocator

7.6 The test Application

7.6.1 Tower of Hanoi problem

In 1883 the French mathematician Edouard Lucas invented the Tower of Hanoi problem [20]. A problem set up could look like the one in Figure 35. Here there are four disks building up a tower on peg one.

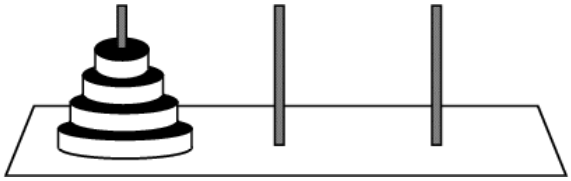


Figure 35: Tower of Hanoi

The problem consists of moving the tower, the four disks, from peg one to peg three. Disks is moved according to the following rules:

1. After each move, the disks will all be stacked on one, two, or three pegs, in decreasing order from the base to the top.
2. The top disk may be lifted from one of the three stacks of disks, and placed on a peg that is empty.
3. The top disk may be lifted from one of the three stacks and placed on top of another stack, provided that the top disk on that stack is larger.

The problem is always possible to solve. Anyone who knows how to solve the problem for eight disks, for example, moving the tower from peg number 1 to peg number 2, also knows how to solve it for nine

disks. First the eight top disks is moved to peg number 3, then the ninth disk moved to peg number 2, and then the eight disks is moved from peg number 3 to peg number 2.

The number of moves required to solve the problem for a tower of *n* disks, is $2^n - 1$ [21].

7.6.2 Usage of dynamic memory in the application

The developed test application solves the tower of Hanoi problem with three pegs and an arbitrary number of disks. Each peg has a linked list associated with it. In the linked list the disks currently located on peg *i* is stored. An element in the list store only the disk size and uses 8 byte. The order of the elements in the list tells how the disks are stacked on the corresponding peg. This linked list is dynamically allocated and for each move of a disk there is one deallocation and one allocation. The total number of elements in the lists will be the number of disks.

The moves are stored in a separate linked list. Each element in this list will hold the size of the moved disk, from which peg it where moved and to which peg it where moved. An element will use 32 bytes. When the problem is solved this list will have $2^n - 1$ elements, where n is the number of disk.

8 Results

Time measurements from executing the test application with the DMA and the software allocator are compared. The quantities measured are, total run time, total allocation time, total deallocation time, maximum allocation time and maximum deallocation time. From these measurements the speedup when using the DMA instead of the software allocator has been calculated.

The chip area required by the DMA for different generic settings have been investigated.

8.1 Time Measurements

The test application was executed using both the DMA and the software allocator for different tower heights. Execution of the application with a total of 6 different tower heights where measured with the time module. The quantities measured are, total run time, total allocation time, total deallocation time, maximum allocation time and maximum deallocation time. The hardware allocator generic settings where set to their default values :

Generic parameter	**Value**
Memory_space	8192
Address_space_Bits	16
Block_size	8
Max_allocation_size	32
Search_width	8

That is the total heap is 8192 Byte and object sizes from 8 Byte to 32 Byte can be allocated.

The software used an equal sized heap and a minimum block size of 8. A complete lists of the measurements are given in appendix [2].

8.1.1 Allocation times

Figure 36 shows the time spent on allocation in the application for different settings of tower height. The number of allocations increases with the tower height.

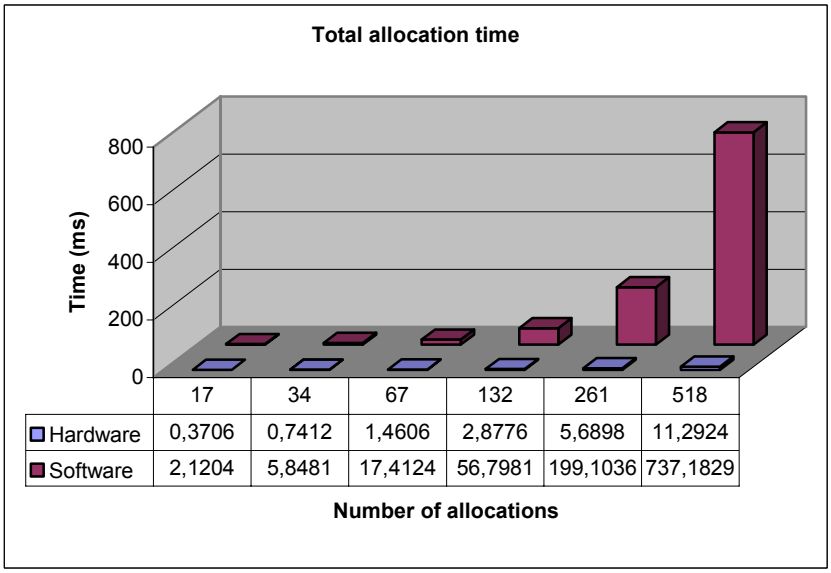


Figure 36: Total allocation time

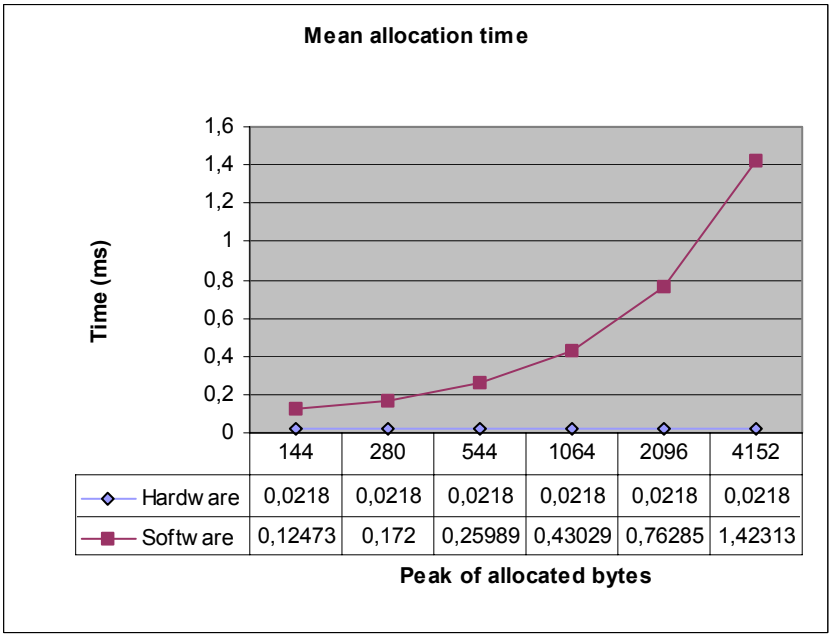


Figure 37: Mean allocation time

Figure 37 displays the mean allocation time for all allocations performed in the applications. The mean allocation time when using the DMA is substantial lower then the software allocation time. The hardware allocation time is constant and independent of the number of allocations made. Whereas the software allocator is dependent on the number of objects that are allocated when performing an allocation. So as the peak of allocations rises, the search time involved in the software allocator increases.

8.1.2 Deallocation times

Figure 38 shows the time spent on deallocation in the application for different settings of tower height.

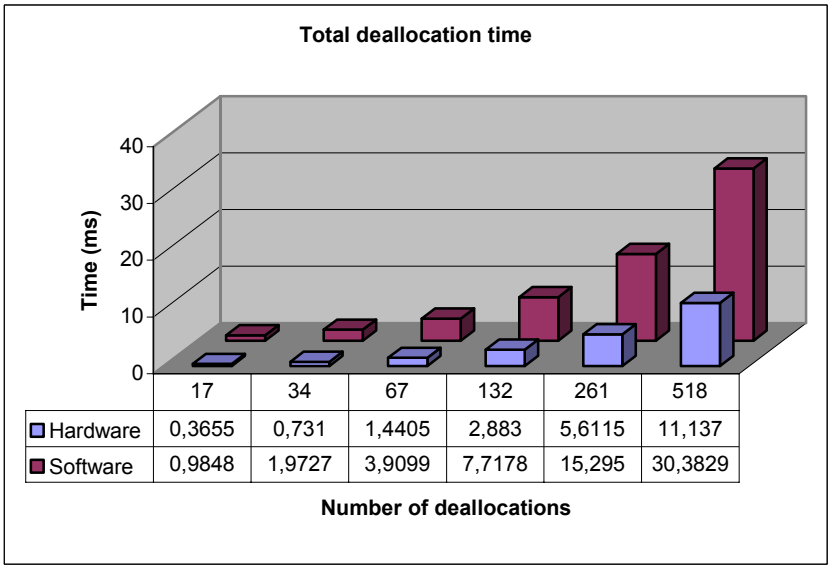


Figure 38 Total deallocation time

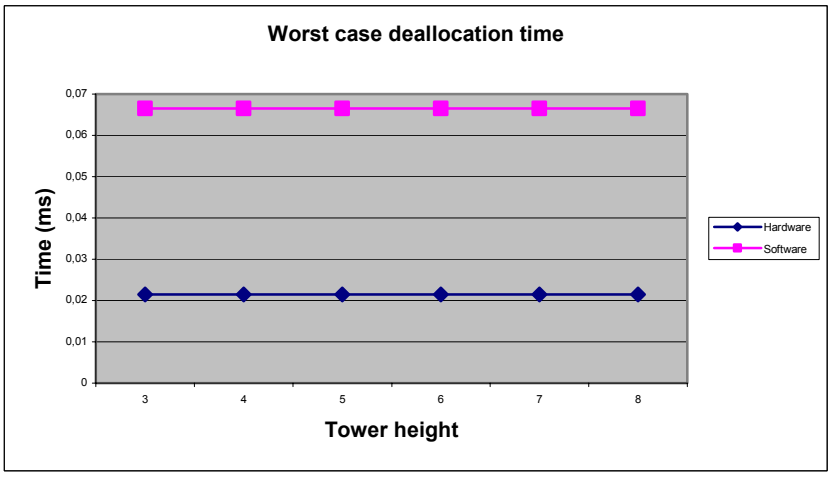


Figure 39: Worst case deallocation time

Worst case software deallocation time depends on the size of the object that shall be deallocated. In the application the maximum object sizes doesn't depend on tower height so worst case deallocation time is constant. As with the allocation time the hardware deallocation time is constant, and is roughly 3 times lower then the software deallocation time.

8.1.3 Run time

Figure 40 shows the total runtime of the application with different height settings

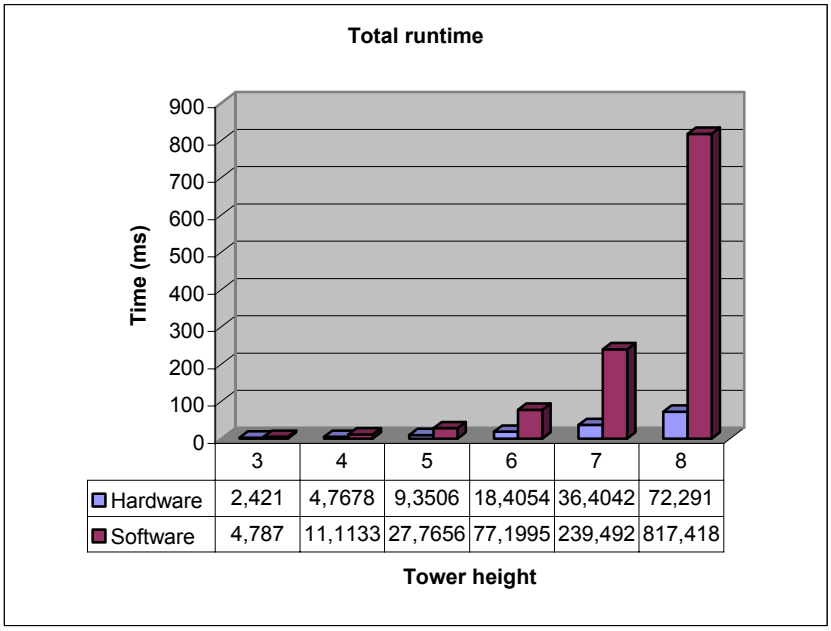


Figure 40: Total runtime

The total execution time drops dramatically for the hardware allocator in application executions involving a larger amount of allocations.

8.1.4 Speedup

By improving some part of a computer a performance increase is gained. The performance can be calculated using *Amdahl's law*. The Amdahl's law states that the improvement in performance by using an improvement (enhancement) is limited by the fraction of time the enhancement can be used. The speedup gained by using an enhancement is defined as :

$$\text{Speedup} = \frac{\text{Execution time for the task without using the enhancement}}{\text{Execution time for the task using the enhancement when possible}}$$

Worst case Allocation speedup

The speedup for the worst case allocation time is given in the Table 36, all measurements are given in appendix [2].

Worst case Allocation speedup							
peak of allocated bytes		144	280	544	1064	2096	4152
Worst case time (ms)	Software	0,254	0,4258	0,7624	0,14254	2,7412	5,3626
	Hardware	0,0218	0,0218	0,0218	0,0218	0,0218	0,0218
	Speedup	**11,57798**	**19,53211**	**34,97248**	**65,38532**	**125,7431**	**245,9908**

Table 36: Worst case allocation speedup

As the software worst case allocation time increase fast as the peak of allocated bytes increases, but the hardware remains its constant allocation time, the speedup grows large corresponding to the software increase in the search time.

Worst case deallocation speedup
Worst case deallocation speedup is constant, 3,093 times, as the maximum deallocation time don't change with the tower height, Figure 39, neither for the software and hardware allocator.

Application speedup
The speedup for the entire application is given in Figure 41. As the application is rather allocation intensive (50 – 90 %) the speedup is significant. In the case of tower height 8, the application speedup was almost 12 times, reducing the application execution time over 90 %.

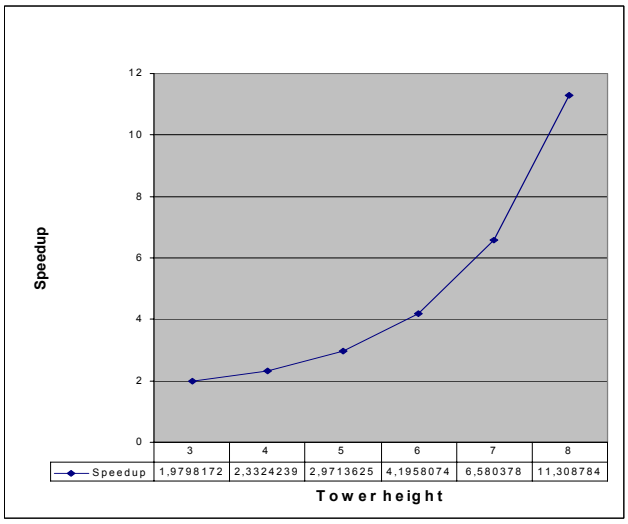


Figure 41: Application speedup

8.2 Gate count

To investigate the area of the architecture for different generic parameters, several synthesis where made. A complete list of the synthesis gate count result are given in appendix [3]. All synthesis where made using LeonardoSpectrum for synthesis and Xilinx design manager for place and route at default settings. The target architecture was a XILINX Spartan XCS40XL.

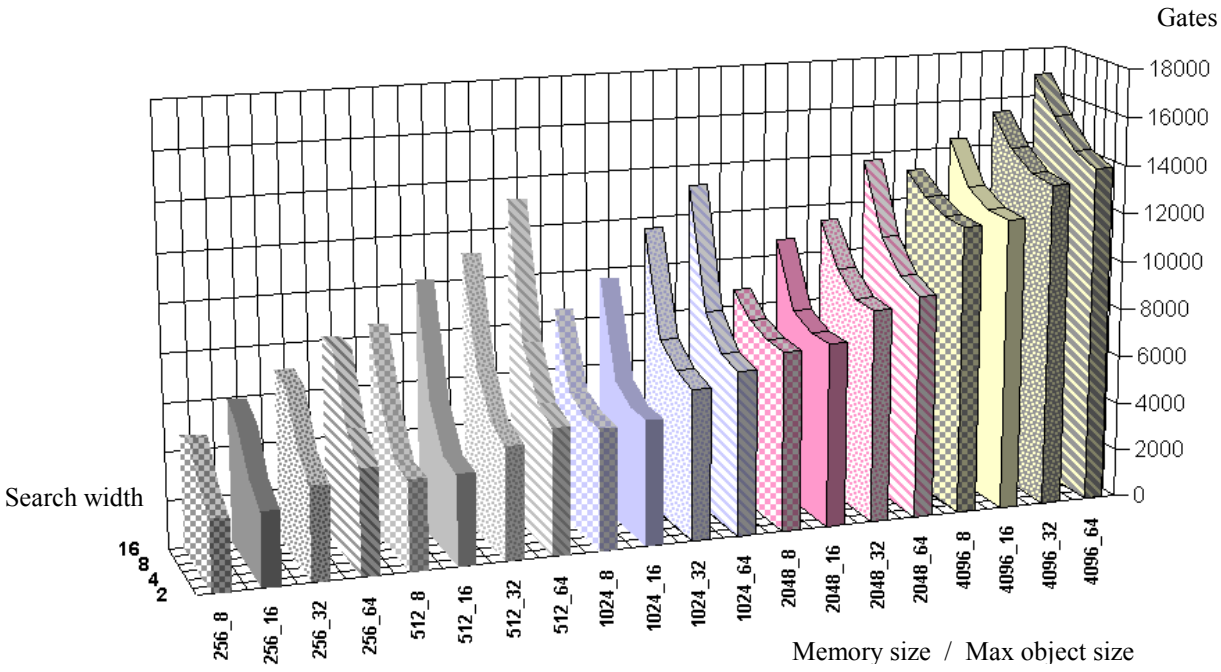


Figure 42: Number of gates

Karl Ingström
kim99001@student.mdh.se
Anders Daleby
ady99002@student.mdh.se

The investigated settings where all made with a block size setting of 8 byte per block. The memory space parameter where varied between 256 Byte and 4096 Byte. The search width parameter between 2 and 16 blocks and the maximum allocation size parameter between 8 and 64 byte. Some investigations where made on larger memory sizes as well.

The measurements indicate that the size of the architecture grows roughly linear with the max allocation size parameter, if the other parameters are kept constant. Increase of the max allocation size parameter will enlarge the address buffer RAM and add bit lengths to counters in the combinatorial units, search, locate and mark. This is seen in Figure 43, where the gate count for different maximum allocation settings have been plotted, for different search widths and a memory space of 1024 bytes.

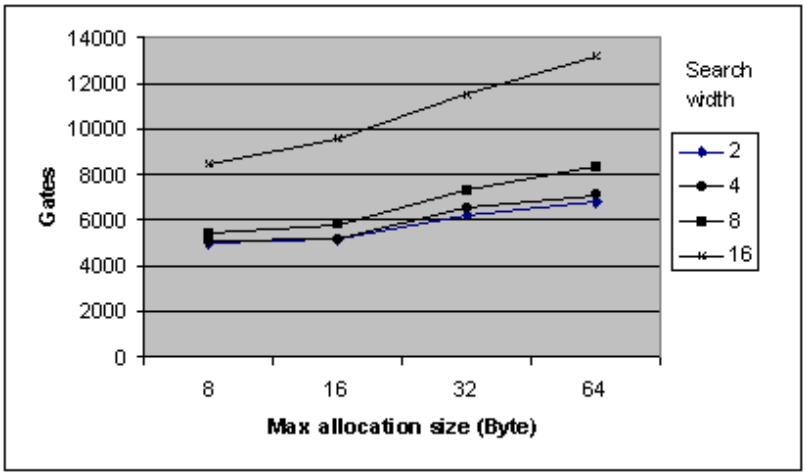


Figure 43: Gate counts depending on maximum allocation parameter

Increase of the search width parameter indicates some sort of exponential growth of area. That is because an increase of the search width parameter will add the amount of counters used in the search unit. This can be seen in Figure 44, where the dependence on the search width parameter has been plotted for a memory size of 1024 byte and varied max object size settings.

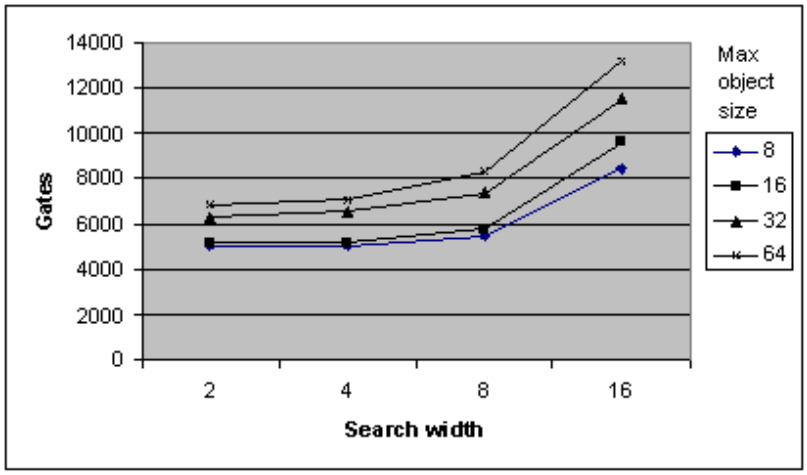


Figure 44: Gate count depending on search width parameter

When the memory sizes are small, increase of search width parameter seems to contribute more to the size of the architecture. As the memory space increase more and more area will be used to represent the bitmaps i.e. RAM memories.

8.2.1 Architecture scalability

If the proportion between the parameters, block size, memory size, max object size and search width, are kept intact, the architecture roughly utilises the same amount of gates. This means that the architecture can be used for allocation of large blocks as well as for small blocks. An example of this is shown in Table 37. Where an allocator in which the total memory size is 16 Mb and 256 blocks of 65kb can be allocated, is compared to an allocator with a total memory space of 2048 byte and a block size of 8 byte. The allocators was synthesised with a search width of 8 and 16 blocks. Both allocators have the same number of vectors, i.e. the internal RAM memories will be of the same size. The number of blocks for both allocators is the same, leading to that the same amount of logic will be used from the interface and inward in the architecture as all calculations is performed in block units.

Memory Size	Block size	Maximum object size	Search width	Number of gates
16777216	65536	262144	8	**9881**
2048	8	32	8	**9705**
16777216	65536	262144	16	**12005**
2048	8	32	16	**11344**

Table 37 Architecture scalability

9 Conclusions

This paper presents the DMA, an approach for performing dynamic allocation and deallocation in hardware. The main idea in the DMA is to search for available free objects before they are requested. Thereby utilising the inactive time, i.e. the time the allocator is not used. This reduces the response time when a service is requested. The pre-search adds a timing constraint between sequential services. The timing restriction is dependent on the generic parameters of the architecture. If the time is fulfilled the DMA will have a predictable behaviour. The allocation service response is guaranteed to be correct, that is if there are available free objects they will be found. Deallocation services has a slightly shorter timing restriction then allocation.

Through generic parameters the DMA architecture can be adopted to a variety of memory sizes, block sizes, number of different object sizes and timing restrictions. The chip area a DMA will use is dependent on the generic parameters. The DMA is mainly constructed for the use of a small amount of different object sizes.

As the developed DMA is technology independent, CPU independent, and is transparent for S/W designers no restriction is laid upon in which type of system the module could be used in, as long as there is some sort of hardware where the DMA can be placed and some sort of memory for doing dynamic allocation in. It could even be a hardware only system.

By the use of a device driver the DMA can replace the ANSI C malloc and free functions for dynamic memory handling. Reusability of old source code should be possible when migrating from a software allocator to the DMA.

The DMA has been tested and compared to a software allocator, with good result. Worst case allocation speed up reached levels between 11 and 245 times faster, depending on the allocation intensity of the application The allocation time when using the DMA is constant while the software algorithm allocation time increased with the number of allocated object. The total application execution time was reduced with 50 to 90 %.

9.1 Future work

9.1.1 DMA improvements

When the unit is used hard it rarely have time to search for the smaller sizes. With the current construction it is possible that a found object replace an equally sized object that have a better location in the memory space, a better location is in the lower part of the memory space. The fragmentation behaviour in this case should be examined to decide if any measures should be done to improve the fragmentation.

- One thing that could be done is to compare the addresses in the service executor unit to see if the new address should be stored or not.

- The search for free objects is currently going from the high addresses down to the lowest. This construction force the search for one size to go through all the vectors. To be sure that all sizes have been searched the unit needs a time corresponding to, the number of vectors multiplied with maximum object size. If instead the search should go upwards in the memory and the search for one size stops when the first object of the wanted size is found. The time spent searching for a size would in general be less. This change in the construction does not change the timing requirements between two service requests, it only reduce fragmentation.

During the implementation of the DMA it was discovered that the search unit determines the maximum clock frequency for the unit. The consequence of this is that the search width parameter must be lowered, which gives an increasing number of vector to search in. According to chapter 5.3 and 6.12 the timing requirement between two service requests are mostly depending on the number of vectors. This indicates that a better implementation of the search unit should be investigated to increase the performance. Another option could be to search for several sizes at the same time, i.e. increase the level of parallelism used.

The clearing of the address buffer bitmap take place when the deallocated object is coalesced with a free object. But it is also cleared if the deallocation index to the vector is zero. This could be avoided if a control of the last bit value in the previous vector is implemented.

9.1.2 Investigations

There are some topics concerning dynamic memory management not considered in this thesis. Such are:

- Investigation of the fragmentation properties of the developed DMA compared to other allocators / allocator mechanisms.
- The possibility to extend the functionality of the DMA, for example reallocation and garbage collection.
- Memory protection.
- Support of multi users, i.e. a the use of the DMA in a multiprocessor system.

10 Bibliography

[1] M68300 Family - MC68332,User's Manual, Motorola 1996

[2] *Computer Architecture – A quantitative approach, 2 ed.* John L. Hennsley and David A. Patterson, Morgan Kuafmans publishers inc. San Francisco, California, 1996

[3] *Synthesis of Hardware Models in C with Pointers and Complex Data Structures* , Luc Séméria, Koichi Sato, Giovanni De Micheli, published in IEEE trans. on VLSI, pp. 743-756, vol. 9, num. 6, December 2001

[4] *Memory allocation costs in large C and C++ programs*, D. L. Detlefs, Al Dosser and B. Zorn, Software Practice and Experience, 24(6), 527—542, 1994.

[5] *The measured cost of conservative garbage collection*. Technical report, Benjamin Zorn, University of Colorado at Boulder, Dept. of Computer Science, Boulder, Colorado, 1992

[6] *Quantifying Behavioural Differences Between C and C++ Programs*. B Calder, D Grunwald and B Zorn, Technical Report CU-CS-698-94, University of Colorado, Boulder, January 1994

[7] *Research Brief : Semiconductor Intellectual Property: The Key Ingredient for SLI/ SoC,* Bryan Lewis, March 23, 2001,Gartner Interactive, http://www.gartner.com

[8] Scalable processors in the billion-transistor era: IRAM, C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick IEEE Computer, vol. 30, no. 9, pp. 75--78, September 1997.

[9] *Resolution of dynamic memory allocation and pointers for the behavioral synthesis from C*. Luc Séméria, Koichi Sato, and Giovanni De Micheli,In Proceedings of Design, Automation, and Test in Europe, pages 312--319, Paris, France, March 2000

[10] *The Memory Fragmentation Problem: Solved?,* Mark S. Johnstone and Paul R. Wilson, ISMM'98 Proceedings of the ACM SIGPLAN International Symposium on Memory Management, Pages 26-36

[11] *Programming languages – C*, ISO/IEC 9899:1999

[12] *A Dynamic Memory Management Unit for embedded real-time System-on-a-Chip*, Mohamed Shalan, Vincent J Mooney III, Georgia Institute of technology.

[13] *A High-Performance Memory Allocator for Object-Oriented Systems*, J. Morris Chang, Edward F. Gehringer, IEEE Transactions on Computers, March 1996 (Vol. 45, No. 3), pp. 357-366.

[14] *Dynamic storage allocation: A survey and critical review*, P. R Wilson, M. S. Johnstone, M. Neely, and D. Boles. In Proceedings of the 1995 International Workshop on Memory Management, volume 986 of Lecture Notes in Computer Science, Kinross, United Kingdom, Sept. 1995. Springer-Verlag.

[15] *Empirical Measurements of Six Allocation-intensive C Programs,* Benjamin Zorn, A Dirk Grunwald, July 1992 , Department of Computer Science, University of Colorado, Boulder, Colorado, Technical Report CU-CS-604-92, Available by anonymous FTP and e-mail from ftp.cs.colorado.edu in the file pub/cs/techreports/zorn/CU-CS-604-92.ps.Z

[16] *Architectural support for dynamic memory management,* J. Morris Chang, Srisa-an, Chia-Tien Dan Lo, Proceedings of IEEE International Conference on Computer Design, Austin, Texas, Sep. 17-20. 2000, pp.99-104.

[17] *Boundary Analysis for Buddy Systems*, C. D. Lo, W. Srisa-an and J. M. Chang, Proceedings of 1998 International Computer Symposium, (Computer Architecture Track), Tainan, Taiwan, Dec. 17-19, 1998. pp. 96-103

[18] *Memory management overview ,* Karl Ingström, Anders Daleby. Department of Computer Engineering, University of Mälardalen, Sweden, 2001.

[19] *A New Implementation Technique for Memory Management*, Mehran Rezaei and Krishna M. Kavi, The University of Alabama in Huntsville] Preprint from the Proceedings of the 2000 SoutheastCon, Nashville, TN, April, 2000

[20] *The MacTutor History of Mathematics archive : François Edouard Anatole Lucas*, J J O'Connor and E F Robertson, School of Mathematics and Statistics University of St Andrews Scotland, ,http://www-history.mcs.st-andrews.ac.uk/history/Mathematicians/Lucas.html

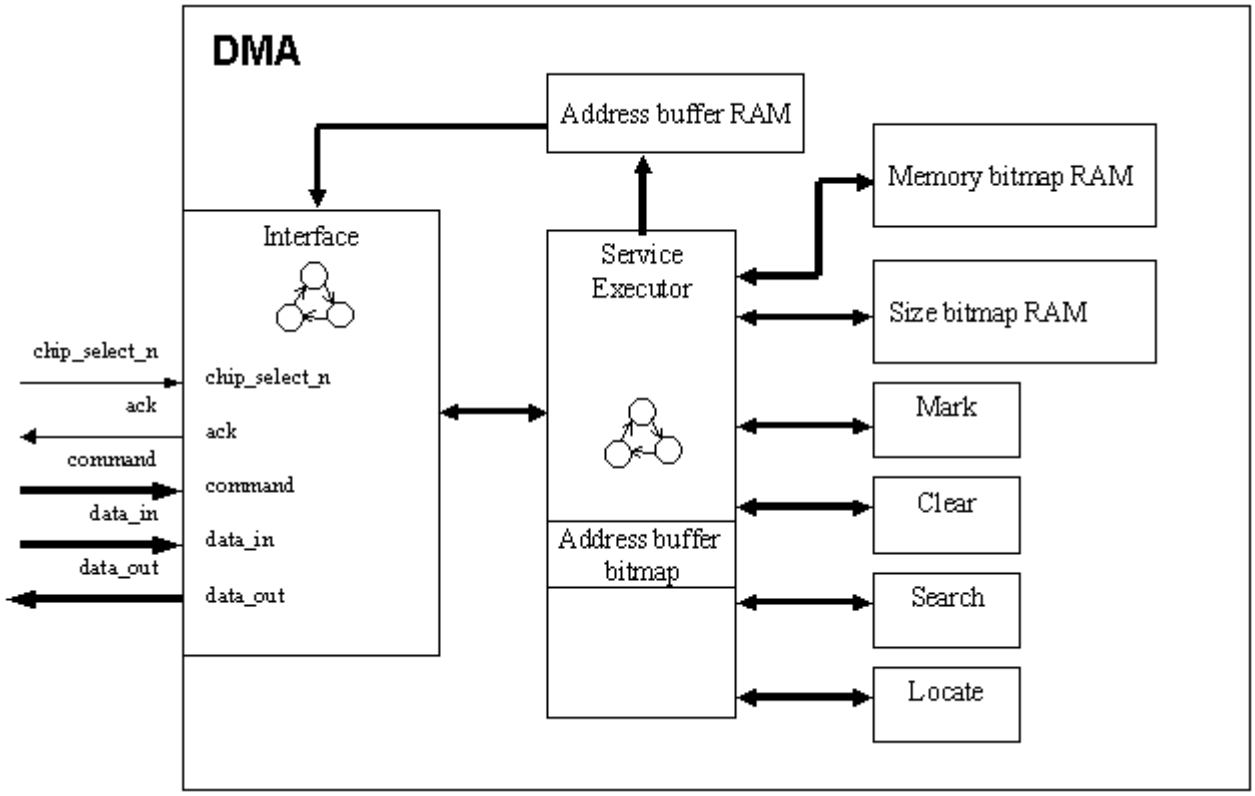
[21] *The Tower of Hanoi*, Philip Spencer, Department of Mathematics, University of Toronto, , http://www.math.toronto.edu/mathnet/games/towers.html

[22] *Design of a Reusable Memory Management System*, K. Agun and J. M. Chang, Proceedings of 14th IEEE International ASIC/SOC Conference, Washington, D.C., Sep. 12-15, 2001.

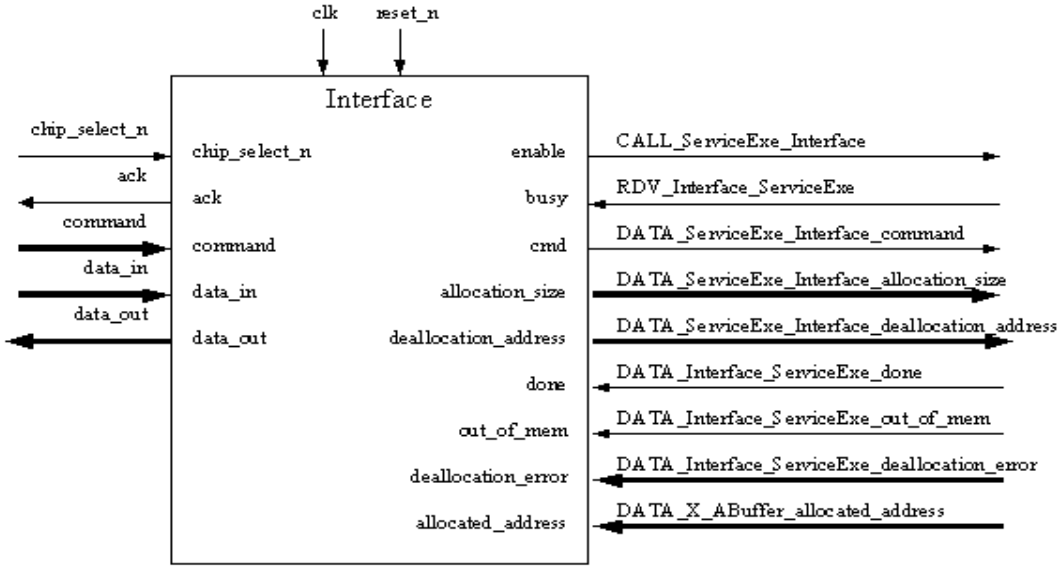
Appendix

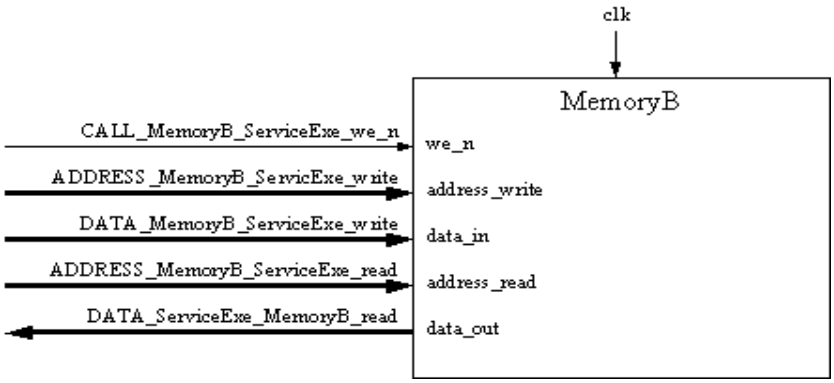
I. Mapping

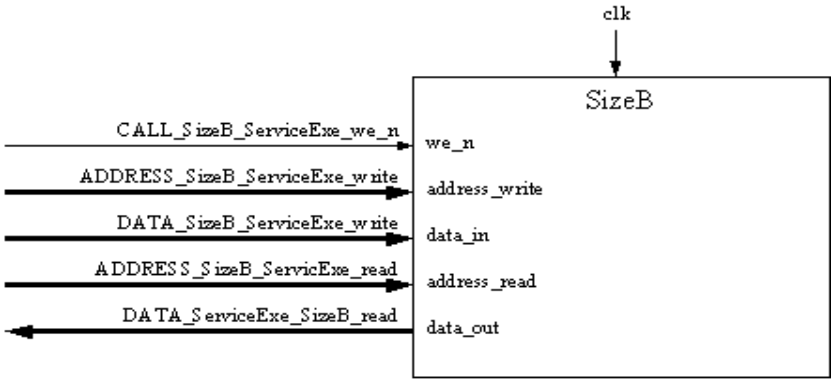
Top level

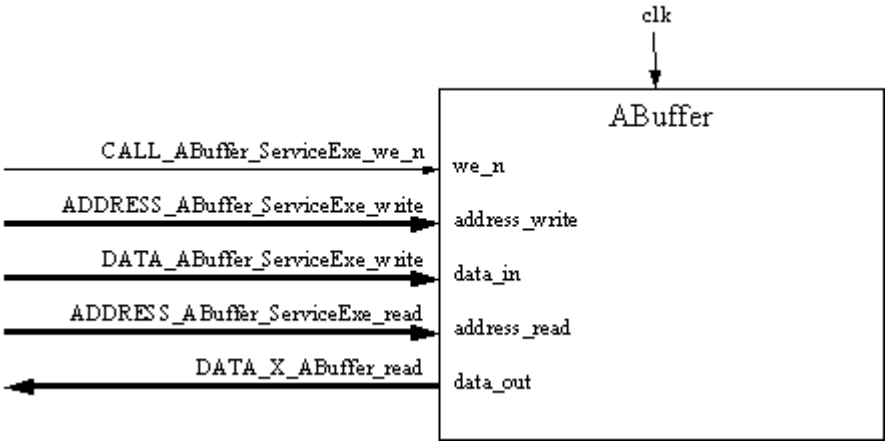


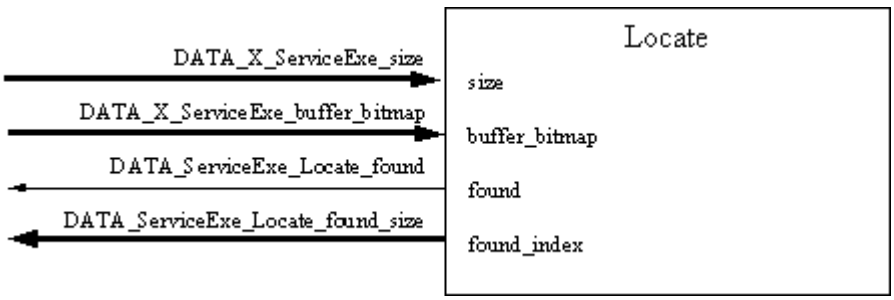
Component mapping





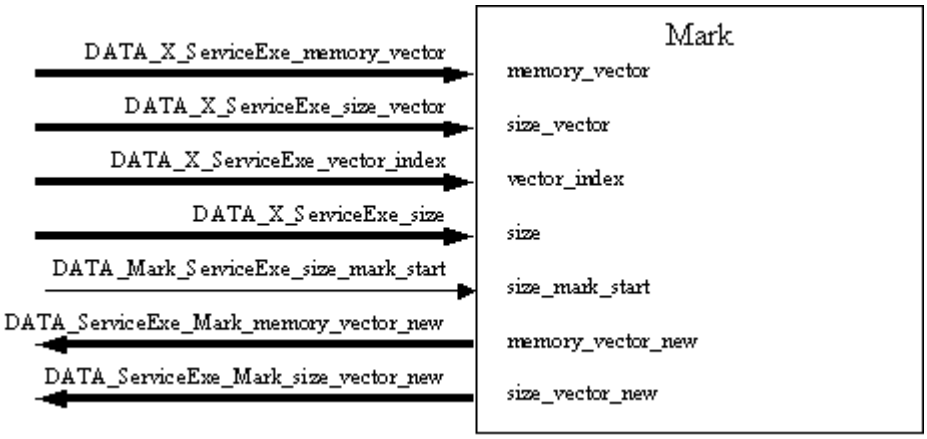


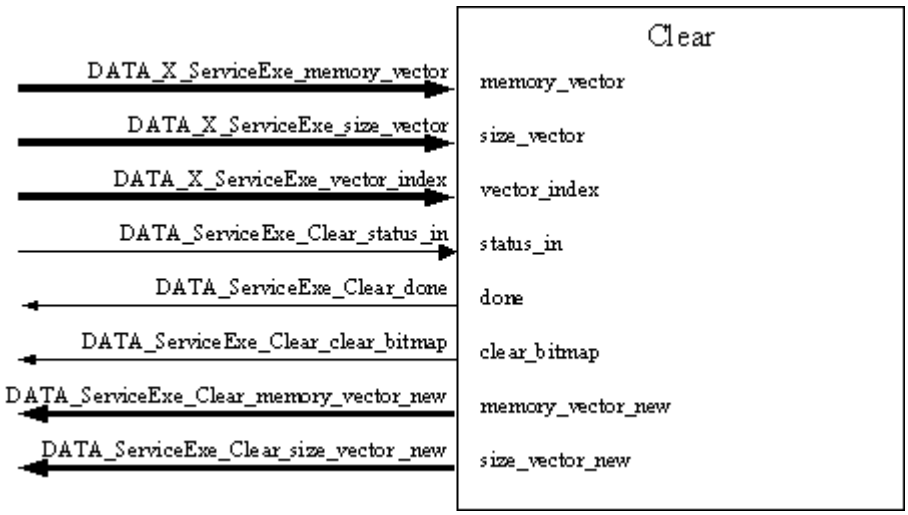


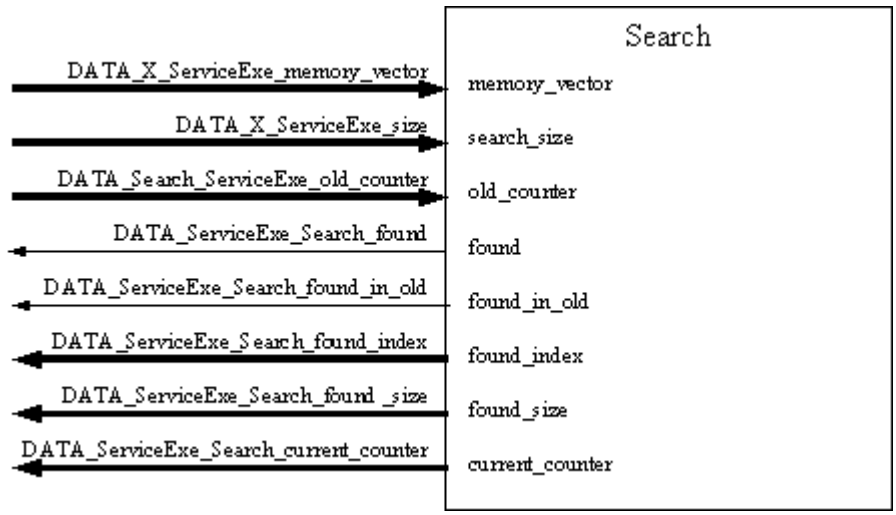


Karl Ingström
kim99001@student.mdh.se

Anders Daleby
ady99002@student.mdh.se

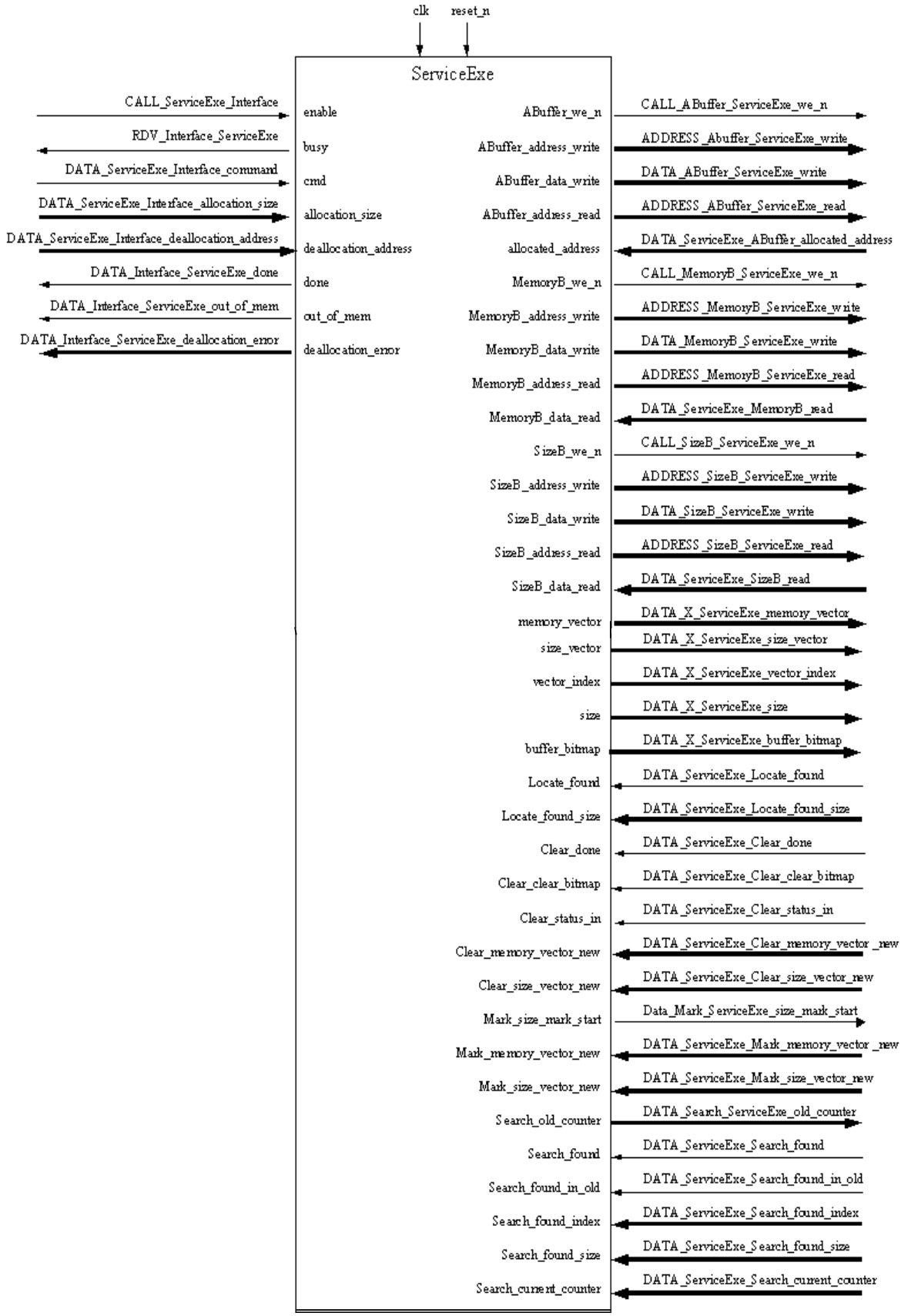






Karl Ingström
kim99001@student.mdh.se

Anders Daleby
ady99002@student.mdh.se



Karl Ingström
kim99001@student.mdh.se

Anders Daleby
ady99002@student.mdh.se

II. Time measurements

Tower height	3
Number of Allocations	17
Number of Deallocations	17
Total amount of allocated bytes	192

	Software	percent of program	Hardware	percent of program	Time reduction ((S-H) / S)	Speedup (S / H)
Total Run time	4,787	100,00%	2,421	100,00%	49,43%	1,979817
Total allocation time	2,1204	44,29%	0,3706	15,31%	82,52%	5,72
Maximum allocation Time	0,2524	5,27%	0,0218	0,90%	91,36%	11,58
Total deallocation time	0,9848	20,57%	0,3655	15,10%	62,89%	2,69
Maximum deallocation time	0,0665	1,39%	0,0215	0,89%	67,67%	3,09

Tower height	4
Number of Allocations	34
Number of Deallocations	34
Total amount of allocated bytes	392

	Software	percent of program	Hardware	percent of program	Time reduction ((S-H) / S)	Speedup (S / H)
Total Run time	11,1133	100,00%	4,7678	100,00%	57,10%	2,332424
Total allocation time	5,8481	52,62%	0,7412	15,55%	87,33%	7,89
Maximum allocation Time	0,4258	3,83%	0,0218	0,46%	94,88%	19,53
Total deallocation time	1,9727	17,75%	0,731	15,33%	62,94%	2,70
Maximum deallocation time	0,0665	0,60%	0,0215	0,45%	67,67%	3,09

Tower height	5
Number of Allocations	67
Number of Deallocations	67
Total amount of allocated bytes	784

	Software	percent of program	Hardware	percent of program	Time reduction ((S-H) / S)	Speedup (S / H)
Total Run time	27,7656	100,00%	9,3506	100,00%	66,32%	2,971363
Total allocation time	17,4124	62,71%	1,4606	15,62%	91,61%	11,92
Maximum allocation Time	0,7624	2,75%	0,0218	0,23%	97,14%	34,97
Total deallocation time	3,9099	14,08%	1,4405	15,41%	63,16%	2,71
Maximum deallocation time	0,0665	0,24%	0,0215	0,23%	67,67%	3,09

Karl Ingström
kim99001@student.mdh.se

Anders Daleby
ady99002@student.mdh.se

Tower height	6
Number of Allocations	132
Number of Deallocations	132
Total amount of allocated bytes	1560

	Software	percent of program	Hardware	percent of program	Time reduction ((S-H) / S)	Speedup (S / H)
Total Run time	77,1995	100,00%	18,4054	100,00%	76,16%	4,185571
Total allocation time	56,7981	73,57%	2,8776	15,63%	94,93%	19,74
Maximum allocation Time	1,4254	1,85%	0,0218	0,12%	98,47%	65,39
Total deallocation time	7,7178	10,00%	2,838	15,66%	62,64%	2,68
Maximum deallocation time	0,0665	0,09%	0,0215	0,12%	67,67%	3,09

Tower height	7
Number of Allocations	261
Number of Deallocations	261
Total amount of allocated bytes	3104

	Software	percent of program	Hardware	percent of program	Time reduction ((S-H) / S)	Speedup (S / H)
Total Run time	239,4922	100,00%	36,4042	100,00%	84,80%	6,580378
Total allocation time	199,1036	83,14%	5,6898	15,63%	97,14%	34,99
Maximum allocation Time	2,7412	1,14%	0,0218	0,06%	99,20%	125,74
Total deallocation time	15,295	6,39%	5,6115	15,41%	63,31%	2,73
Maximum deallocation time	0,0665	0,03%	0,0215	0,06%	67,67%	3,09

Tower height	8
Number of Allocations	518
Number of Deallocations	518
Total amount of allocated bytes	6184

	Software	percent of program	Hardware	percent of program	speedup (S-H) / S	Speedup (S / H)
Total Run time	817,4181	100,00%	72,291	100,00%	91,16%	11,30878
Total allocation time	737,1829	90,18%	11,2924	15,62%	98,47%	65,28
Maximum allocation Time	5,3626	0,66%	0,0218	0,03%	99,59%	245,99
Total deallocation time	30,3829	3,72%	11,137	15,41%	63,34%	2,73
Maximum deallocation time	0,0665	0,01%	0,0215	0,03%	67,67%	3,09

Karl Ingström
kim99001@student.mdh.se

Anders Daleby
ady99002@student.mdh.se

III. Gate Count measurements

All measurement uses a block size setting of 8 byte per block.

Memory Size	Maxobject size	Search width	Number of gates	Allocation Timing restriction
256	8	2	2953	25
256	8	4	3687	17
256	8	8	5071	13
256	8	16	4610	11
256	16	2	3063	26
256	16	4	3723	18
256	16	8	5365	14
256	16	16	5909	12
256	32	2	3880	27
256	32	4	4642	18
256	32	8	6698	14
256	32	16	6954	12
256	64	2	4375	29
256	64	4	5404	19
256	64	8	8060	14
256	64	16	8132	12
512	8	2	3745	41
512	8	4	3829	25
512	8	8	5426	17
512	8	16	8457	13
512	16	2	3776	42
512	16	4	3930	26
512	16	8	5666	18
512	16	16	10128	14
512	32	2	4708	43
512	32	4	5149	26
512	32	8	6821	18
512	32	16	11041	14
512	64	2	5242	45
512	64	4	5724	27
512	64	8	7712	18
512	64	16	13128	14
1024	8	2	5025	73
1024	8	4	5042	41
1024	8	8	5469	25
1024	8	16	8473	17
1024	16	2	5172	74
1024	16	4	5168	42
1024	16	8	5768	26
1024	16	16	9599	18
1024	32	2	6245	75
1024	32	4	6562	42
1024	32	8	7349	26
1024	32	16	11520	18

Karl Ingström
kim99001@student.mdh.se

Anders Daleby
ady99002@student.mdh.se

Memory Size	Maxobject size	Search width	Number of gates	Allocation Timing restriction
1024	64	2	6839	77
1024	64	4	7094	43
1024	64	8	8312	26
1024	64	16	13168	18
2048	8	2	7448	137
2048	8	4	7484	73
2048	8	8	7832	41
2048	8	16	8684	25
2048	16	2	7643	138
2048	16	4	7622	74
2048	16	8	8069	42
2048	16	16	10653	26
2048	32	2	8814	139
2048	32	4	8871	74
2048	32	8	9705	42
2048	32	16	11344	26
2048	64	2	9275	141
2048	64	4	9696	75
2048	64	8	10888	42
2048	64	16	13707	26
4096	8	2	11950	265
4096	8	4	12012	137
4096	8	8	12424	73
4096	8	16	13169	41
4096	16	2	12082	266
4096	16	4	12187	138
4096	16	8	12709	74
4096	16	16	14389	42
4096	32	2	13378	267
4096	32	4	13540	138
4096	32	8	14254	74
4096	32	16	15422	42
4096	64	2	13936	269
4096	64	4	14373	139
4096	64	8	15481	74
4096	64	16	16903	42
16777216	262144	8	9881	42
16777216	262144	16	12005	26

Karl Ingström
kim99001@student.mdh.se

Anders Daleby
ady99002@student.mdh.se

IV. Testbench output

Here a testbench output file are given for a command file. The settings used where those of the table below.

Name	Value
Memory_space	512
Address_space_Bits	16
Block_size	4
Max_allocation_size	32
Search_width	8
Base address to DMA	7F0040

Command file used :

```
o 4096
f 4
a 8
f 2
a 32
d -2
f 8
a 4
d 96
f 4
a 32
f 4
d -1
f 3
d -1
f 2
a 24
d 32
d 64
f 5
a 24
f 2
d -2
a 4
d 32
d 56
d 156
d 180
d 204
p
```

Testbench output:

```
Starting Test
------------------------------------------------
command #1
Allocation of :8 bytes
  - Allocation service performed OK
    Address: 4096
------------------------------------------------
command #2
Allocation of :8 bytes
  - Allocation service performed OK
    Address: 4104
------------------------------------------------
command #3
Allocation of :8 bytes
  - Allocation service performed OK
    Address: 4112
------------------------------------------------
command #4
Allocation of :8 bytes
  - Allocation service performed OK
    Address: 4120
------------------------------------------------
command #5
Allocation of :32 bytes
  - Allocation service performed OK
    Address: 4128
------------------------------------------------
command #6
Allocation of :32 bytes
  - Allocation service performed OK
    Address: 4160
------------------------------------------------
command #7
Deallocation of address:4096
  - Deallocation service performed OK
------------------------------------------------
command #8
Allocation of :4 bytes
  - Allocation service performed OK
```

Karl Ingström
kim99001@student.mdh.se

Anders Daleby
ady99002@student.mdh.se

```
    Address: 4096
-------------------------------------------------
command #9
Allocation of :4 bytes
  - Allocation service performed OK
    Address: 4100
-------------------------------------------------
command #10
Allocation of :4 bytes
  - Allocation service performed OK
    Address: 4192
-------------------------------------------------
command #11
Allocation of :4 bytes
  - Allocation service performed OK
    Address: 4196
-------------------------------------------------
command #12
Allocation of :4 bytes
  - Allocation service performed OK
    Address: 4200
-------------------------------------------------
command #13
Allocation of :4 bytes
  - Allocation service performed OK
    Address: 4204
-------------------------------------------------
command #14
Allocation of :4 bytes
  - Allocation service performed OK
    Address: 4208
-------------------------------------------------
command #15
Allocation of :4 bytes
  - Allocation service performed OK
    Address: 4212
-------------------------------------------------
command #16
Deallocation of address:4192
  - Deallocation service performed OK
-------------------------------------------------
command #17
Allocation of :32 bytes
  - Allocation service performed OK
    Address: 4216
-------------------------------------------------
command #18
Allocation of :32 bytes
  - Allocation service performed OK
    Address: 4248
-------------------------------------------------
command #19
Allocation of :32 bytes
  - Allocation service performed OK
    Address: 4280
-------------------------------------------------
command #20
Allocation of :32 bytes
  - Allocation service performed OK
    Address: 4312
-------------------------------------------------
command #21
Deallocation of address:4312
  - Deallocation service performed OK
-------------------------------------------------
command #22
Deallocation of address:4280
  - Deallocation service performed OK
-------------------------------------------------
command #23
Deallocation of address:4248
  - Deallocation service performed OK
-------------------------------------------------
command #24
Deallocation of address:4216
  - Deallocation service performed OK
-------------------------------------------------
command #25
Deallocation of address:4212
  - Deallocation service performed OK
-------------------------------------------------
command #26
Deallocation of address:4208
  - Deallocation service performed OK
-------------------------------------------------
command #27
Deallocation of address:4204
  - Deallocation service performed OK
-------------------------------------------------
command #28
Allocation of :24 bytes
  - Allocation service performed OK
    Address: 4204
-------------------------------------------------
command #29
Allocation of :24 bytes
  - Allocation service performed OK
    Address: 4228
-------------------------------------------------
command #30
Deallocation of address:4128
  - Deallocation service performed OK
-------------------------------------------------
command #31
Deallocation of address:4160
  - Deallocation service performed OK
-------------------------------------------------
command #32
Allocation of :24 bytes
  - Allocation service performed OK
    Address: 4128
-------------------------------------------------
command #33
Allocation of :24 bytes
  - Allocation service performed OK
    Address: 4152
-------------------------------------------------
command #34
```

Karl Ingström
kim99001@student.mdh.se

Anders Daleby
ady99002@student.mdh.se

Allocation of :24 bytes
- Allocation service performed OK
Address: 4252
---
command #35
Allocation of :24 bytes
- Allocation service performed OK
Address: 4276
---
command #36
Allocation of :24 bytes
- Allocation service performed OK
Address: 4300
---
command #37
Deallocation of address:4104
- Deallocation service performed OK
---
command #38
Deallocation of address:4112
- Deallocation service performed OK
---
command #39
Allocation of :4 bytes
- Allocation service performed OK
Address: 4104
---
command #40
Deallocation of address:4128
- Deallocation service performed OK
---
command #41
Deallocation of address:4152
- Deallocation service performed OK
---
command #42
Deallocation of address:4252
- Deallocation service performed OK
---
command #43
Deallocation of address:4276
- Deallocation service performed OK
---
command #44
Deallocation of address:4300
- Deallocation service performed OK

Printing allocated addresses
Begining with last addresses allocated
4104
4228
4204
4200
4196
4100
4096
4120

Karl Ingström
kim99001@student.mdh.se

Anders Daleby
ady99002@student.mdh.se