A DYNAMIC MEMORY MANAGER FOR FPGA APPLICATIONS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

CENK ÖZER

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

JUNE 2014

Approval of the thesis:

A DYNAMIC MEMORY MANAGER FOR FPGA APPLICATIONS

submitted by **CENK ÖZER** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** __________

Prof. Dr. Gönül Turhan Sayan
Head of Department, **Electrical and Electronics Engineering** __________

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı
Supervisor, **Electrical and Electronics Eng. Dept., METU** __________

Examining Committee Members:

Prof. Dr. Semih Bilgen
Electrical and Electronics Engineering Dept., METU ____________________

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı
Electrical and Electronics Engineering Dept., METU ____________________

Prof. Dr. Gözde Bozdağı Akar
Electrical and Electronics Engineering Dept., METU ____________________

Assoc. Prof. Dr. Ece Güran Schmidt
Electrical and Electronics Engineering Dept., METU ____________________

Dr. Fatih Say
ASELSAN Inc. ____________________

Date: 20/06/2014

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Cenk ÖZER

Signature :

ABSTRACT

A DYNAMIC MEMORY MANAGER FOR FPGA APPLICATIONS

Özer, Cenk

M.S., Department of Electrical and Electronics Engineering

Supervisor: Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı

June 2014, 49 pages

Recently, FPGAs are shipped with a large amount of internal memory (block RAM) sufficient to perform many complex computations without a need for off-chip memory. However, block RAMs (BRAMs) of FPGAs should be used efficiently especially for computations that need dynamic management of the memory. Thus, within the scope of this thesis work, a dynamic memory manager (DMM) unit is designed with an objective of meeting memory requests with a low fragmentation at runtime for FPGA applications. The unit is designed to have a bounded response time for dynamic memory requests to be suitable for real time applications. It can be interfaced with FPGA applications quite easily similar to interfacing an arbitrary IP core block. The proposed real-time DMM differs from other conventional memory allocators in a way that it allows for memory allocations composed of differing size blocks that are not necessarily contiguous. The address translator block in design provides to access separate non-contiguous blocks as a whole contiguous chunk of memory. Implementation and verification of the developed DMM on an FPGA demo board is also presented using synthetic memory request streams.

Keywords: Dynamic Memory Allocation, Hardware Allocator, Dynamic Memory Management Unit, Field Programmable Gate Array

ÖZ

APKD UYGULAMALARI İÇİN DİNAMİK BELLEK YÖNETİCİSİ

Özer, Cenk
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Cüneyt F. Bazlamaçcı

Haziran 2014, 49 sayfa

Son yıllarda APKD'ler, birçok karmaşık işlemin harici bellek ihtiyacı olmadan da yapılabilmesini sağlayacak oranda yüksek miktarlarda dahili bellek ile pazara sunulmaktadır. Ancak, özellikle dinamik bellek yönetimi gereken işlemlerde APKD'lerin blok bellekleri verimli kullanılmalıdır. Bu amaçla, bu tez çalışması kapsamında APKD uygulamalarının dinamik bellek isteklerini en düşük parçalanma ile karşılama gereksinimi sağlamayı ön planda tutan bir dinamik bellek yöneticisi (DBY) tasarlanmıştır. Bu birim, gerçek zamanlı uygulamalara da uygun olması için dinamik bellek isteklerini sınırlı zamanda karşılayabilecek biçimde tasarlanmıştır. APKD uygulamaları ile arayüzü, herhangi bir IP bloğuna benzer şekilde yapılabilmektedir. Önerilen gerçek zamanlı DBY, geleneksel dinamik bellek tahsis edicilerden bir yönüyle ayrılmakta ve bellek tahsis isteklerini bitişik tek blok halinde değil, birbiriyle ardışık olması gerekmeyen çeşitli büyüklüklerdeki bloklar halinde karşılayabilmektedir. Bu ayrık bloklara bitişik bir bellek alanıymış gibi erişilebilmesini adres çevirici sağlamaktadır. Geliştirilen DBY, bir APKD gösterim kartı üzerinde yapay bellek istek dizileri yaratılarak denenmiş ve doğrulanmıştır.

Anahtar Kelimeler: Dinamik Bellek Tahsisi, Donanımsal Bellek Tahsisi, Dinamik Bellek Yöneticisi, Alanda Programlanabilir Kapı Dizileri

To my family

ACKNOWLEDGEMENTS

I sincerely thank to my supervisor Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı for all his guidance and support throughout my study.

I would like to thank to my employer, ASELSAN.

I also thank Turkish Scientific and Technological Research Council (TÜBİTAK) for their financial support during my study.

I am grateful to my family for their love, trust and support throughout my life.

Last but not least, I would like to thank to my indispensable part of my life, Gözde Özer, for her endless love, support, encouragement and patience throughout my study.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

APKD	Alanda Programlanabilir Kapı Dizileri
BRAM	Block Random Access Memory
CPU	Central Processing Unit
DBY	Dinamik Bellek Yöneticisi
DMAC	Dynamic Memory Allocation Core
DMM	Dynamic Memory Manager
DMMX	Dynamic Memory Management Extension
EMA	Efficient Memory Allocation
FIFO	First In First Out
FLM	Free List Manager
FPGA	Field Programmable Gate Array
HS	Heap Size
IP	Intellectual Property
KB	Kilobyte
LUT	Look Up Table
MO	Memory Overhead
RAM	Random Access Memory
TLSF	Two Level Segregated Fits
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus
VHDL	Very High Speed Integrated Circuit Hardware Description Language

CHAPTER 1

INTRODUCTION

1.1 Dynamic Memory Allocation

Dynamic memory management has been an attractive topic in computer science since 1960s. Many researchers have been interested in managing memory requests of applications at runtime. Since the same memory area can be used again and again it is found much superior compared to static memory usage. However, handling memory requests in a dynamic fashion requires a special unit called the dynamic memory management unit. Part of the main memory, which is reserved to be used for such dynamic requests is named as heap memory. Since the requests cannot be identified at compile time, a unit, called as dynamic storage allocator or dynamic memory allocator, is used to control this heap memory. Basically, what dynamic memory manager does is keeping track of free and occupied parts of the managed memory. While doing this, it benefits from an internal data structure and a policy that governs the heap memory. Simply, two operations are done. The first one is allocating a free place and the latter one is freeing an occupied place. In order to allocate memory, only the requested size must be known. The response will then be the starting address of the reserved memory block. In case of deallocation, starting address of the block, which is required to be released should be given as input. Size information is not necessary since it is kept by the memory manager. The state of the memory blocks, free or not, is also recorded in the data structure, which may be in the form of a linked list, bitmap, tree, etc. [1]. Ideally, a dynamic memory manager should perform all these operations in a very short amount of time and minimize wasted space in the heap memory.

1.2 Performance Challenges

In recent five decades, many different types of dynamic memory management units are designed for many different platforms with various approaches. According to Wilson *et al.* [1], there may always be some applications that can beat an allocator policy and severely decrease its performance. Thus, it is very hard to design a management unit that performs well for all dynamic memory intensive applications. To evaluate the performance of an allocator, there are some absolute metrics, which will be briefly described below.

1.2.1 Execution Time

The time spent for allocation and deallocation of memory blocks is an important metric for performance. This may be quite significant while a dynamic memory intensive application is running. For software based allocators, execution times can reach up to 38% of the total runtime for some allocation intensive object oriented applications [2] [3]. However, execution times have been gradually decreased since the middle of 90's. Hardware based allocators decrease the time spent in return for extra hardware and complexity. Another important virtue is to have a bounded response time. In this way a memory management unit can be used for real time applications also.

1.2.2 Fragmentation

Fragmentation is simply defined as inability to use free heap memory [1]. It is classified as internal and external fragmentation. Internal fragmentation occurs when a larger block is reserved for a smaller size. For example a 16B block can be reserved for 10B and the 6B becomes useless, if there is no mechanism to split this large block. However, splitting comes with a time burden in a dynamic memory manager. External fragmentation, on the other hand, occurs when there are many small size blocks but there is no one block with a size larger than or equal to the requested size.

Different approaches are expressed as fragmentation measure in [20]. Within the scope of this thesis work, the ratio of the total wasted memory size to the whole heap memory size at the point of operation where dynamic memory manager cannot respond to memory allocation requests is used as the fragmentation measure. Since BRAMs of FPGAs are of limited size, their efficient use gains more importance and attracts more attention providing motivation for the DMM design in this work.

1.2.3 Memory Overhead

Dynamic memory managers use various data structures, which also need additional memory to keep track of the heap memory. Those structures that are based on software mostly use object headers to keep information such as size, link, etc. Hardware counterparts, on the other hand, mostly use bitmaps for storing such information. Bitmaps that show size and availability of a memory block are also stored in BRAMs in FPGAs. Thus, memory overhead should be kept as low as possible to decrease memory cost of an allocator. However, this may lead to inefficiencies in terms of execution time and fragmentation.

1.2.4 Scalability

Scalability is another issue that designers should be concerned about. How design complexity, execution time and memory overhead are affected when heap memory grows should be taken into account.

1.3 Motivation

For decades many different dynamic memory allocation techniques have been used for processor based object oriented systems, which tend to allocate and deallocate memory blocks frequently [4]. In 90s, object oriented programming languages, such as C and C++, were started to be used also for hardware synthesis. Generally, they have been used in hybrid CPU-FPGA based systems for acceleration purposes. Hardware synthesizable parts of codes are now implemented on FPGAs to benefit from parallelism. However, implementing dynamic structures such as dynamic memory allocators, pointers, etc. have not been a straightforward task. Semeria *et al*. [5] presented research results that allows synthesizing C code with dynamic memory allocation efficiently for hardware by accessing a primitive, which performs the allocation and deallocation tasks. Some other similar works have appeared in [4] [6]. In the present thesis work, a dynamic memory manager is designed as a hardware core block similar to those previously mentioned studies.

In [22], a reconfigurable platform is developed, which includes a hardware and software operating system for handling the context switching of hardware tasks. The system area circuitry in [22] is responsible for handling the memory request. The DMM developed in the present work may be considered as a candidate primitive core block targeting the mentioned system area circuitry.

The main concern however, is to keep fragmentation at very low levels while managing the BRAMs, which are limited and very valuable resources of FPGAs. Besides fragmentation, we aim to have a fast DMM with a bounded response time. In other words, it should be suitable to be used in systems that have real time constraints. Although BRAMS in FPGAs are targeted and are the main source of motivation, it should be also applicable for any type of memory.

1.4 Contributions

This thesis work focuses on managing BRAMs of FPGA for dynamic memory requests. In order to achieve this goal, a DMM is designed, which is suitable for real time FPGA applications that need dynamic memory usage. Our proposed DMM can be interfaced with FPGA applications very easily similar to interfacing an arbitrary IP core block. It has bounded response time for allocation (21 clock cycles), deallocation (10 clock cycles) and address translation (2 clock cycles) processes. The fragmentation depends on the average size and distribution of the memory request stream. However, it can be optimized by adjusting the block sizes to keep fragmentation at low levels.

1.5 Thesis Organization

The remainder of the thesis is structured as follows. Background information about the thesis subject and prior work about dynamic memory management are given in Chapter 2. In Chapter 3, a detailed description of the DMM design is presented. Chapter 4 includes the implementation details of the proposed design. Experimental setup and evaluations of the design appears in Chapter 5. Finally, Chapter 6 concludes the thesis, also suggesting some future directions.

CHAPTER 2

BACKGROUND

2.1 Dynamic Memory Allocation Concepts

In this chapter, some background information and definitions are given related to dynamic memory management. As was mentioned earlier, the purpose of dynamic memory manager is to track the availability of the memory area, which is reserved for memory requests in runtime. While performing this task, it aims to minimize the wasted space and time spent. Furthermore, when the application doesn't need a reserved place anymore, deallocation is done by the DMM. Related commands and their arguments should be delivered to DMM to perform the necessary allocation and deallocation tasks. For example, in object oriented language C++, *new* and *delete* represent allocation and deallocation commands, respectively. Allocation command takes memory size as an argument and returns the starting address of the reserved block for that request. On the other hand, deallocation takes the starting address of the block to be freed as its argument. The size is kept in the data structure of the memory manager, thus it is not a required argument for deallocation.

Policy and *mechanism* are two issues to be addressed within the context of memory allocation. *Policy* is a design procedure that is implementable for the placement of the requested memory. Next fit and best fit policies are examples of placement policy. Algorithms and data structures that are used to implement a policy is called a *mechanism* [1]. For instance, linked list that keeps free blocks as a list connected to each other is a *mechanism* example.

Fragmentation is the inability to use free memory due to allocation policy and mechanism [1]. It is classified as *internal* and *external* fragmentation [16]. *External fragmentation* occurs when there is free memory for allocation, but there is no available block which can meet the requested size. For example, there can be a lot of non-contiguous blocks that have 10B size, but it may not be possible to provide a place for a 20B request. The other case, *internal fragmentation* occurs when a larger memory is reserved for a small size request. For a 10B request for example, a DMM may allocate a 16B block and a 6B internal fragmentation occurs, if there is no *splitting* policy.

Splitting, as the name implies, divides large blocks into smaller ones to prevent internal fragmentation. In the previous example, if splitting were used, 6B would be added to the free list and marked as a free block after splitting. However, such a policy may generate many small blocks in the memory thus causing external fragmentation after a while [1]. In order to prevent this, another policy called as *coalescing,* is used. Coalescing merges adjacent free blocks in order to form larger blocks. It is worth noting that each such policy brings extra burden in terms of execution time.

2.2 Classification of Allocators

One type of classification can be done based on whether the dynamic memory allocator is software or hardware based. As a quick comparison, we can say that hardware allocators are considerably faster, more expensive and more complex compared to software allocators. A better classification is done according to mechanism and policy [20].

Sequential Fits, usually uses the linked list structure for keeping the free blocks. These blocks are maintained in FIFOs or LIFOs, which are searched according to the allocation policy. In this technique, search time may be considerably long when the number of free blocks increases. *Best fit, first fit* and *next fit* are the best known sequential fit policies. In the *best fit*, the smallest size block, which is enough to meet the request, is searched. Obviously, it gives better results compared to the other two policies in terms of fragmentation. However, it may suffer from long search times. Hence, it does not suit well to large heaps. *First fit* searches the list starting at the beginning of the list with every incoming request. The first block found to be larger than or equal to the requested size is reserved. If the reserved block is larger, it is split and the remainder part is added to the free list again. In this technique, large blocks at the beginning of the list will be divided first. Number of small blocks increase as time goes by and external fragmentation occurs as a result. Also, search time may considerably increase for larger blocks following the formation of many small blocks. *Next fit* can be seen as an optimization to first fit. Searching process begins at the point where it was left last. This approach improves search time, however it causes more fragmentation compared to best fit and first fit.

Segregated Free Lists is an array of free lists, which keeps the free blocks separately according to particular sizes. Since the size range of free lists are known, it is quite a fast technique. Known implementations can be classified as the *simple segregated storage* and *segregated fits.* In *simple segregated storage,* splitting is not applied. Thus, if one of the size classes is demanded a lot, it causes severe problems. On the other hand, *segregated fits* enables splitting if requested size class is empty. It reserves larger block than requested, splits and adds the remainder to related size class. There are three schemes according to lists and size classes namely, *exact lists, strict size classes with rounding* and *size classes with range list.* There are different free lists for every possible block size in exact lists. This may lead to a large number of free lists. In the second scheme, there exists defined sizes (e.g. powers of two) and requests are rounded to the minimum class size that is available in the size list. This approach reduces the number of free lists belonging to different sizes, however rounding up cause internal fragmentation to a certain extent. The last approach has free lists with a range of size. Since there are different sized blocks in the list, a sequential search (next fit, best fit, first fit) is generally carried out in the list.

Buddy System is a specialized case of segregated fits mechanism. It uses size classes with rounding and restricts splitting and coalescing according to some predefined rules. *Binary, fibonacci, double, weighted buddies* are examples of buddy systems. In all of these schemes, newly deallocated block is coalesced with its buddy if the buddy is free also. Only the size of buddies vary in these buddy systems. For example, heap area will be divided two equal parts in binary buddy system. These parts are also divided equal parts to handle sufficiently small area for memory requests. On the other hand, in Fibonacci buddy system, divisions are arranged to form a Fibonacci series.

Bitmapped Fits is the policy that uses a bit vector to represent free or used areas in the heap memory. For each block in the memory, a flag shows whether it is free or not. This may be regarded as a slow mechanism in software implementations, however it may be implemented quite fast in hardware [4].

2.3 Literature Overview

There have been many research works conducted about dynamic memory management since 1960s. The state of the art until 1995 is well summarized in [1]. It is a good reference, which includes general concepts about dynamic storage, fundamental techniques, and classification of memory allocation algorithms before summarizing the articles that have been published until that time. From then on, there appeared many other articles about dynamic memory management techniques, among which hardware based techniques also took place.

Chang and Gehringer's modified buddy system [4] is one hardware implementation of a buddy system with the bitmap approach. It uses pure combinational logic for allocation and freeing operations so that they are performed fast and in constant time. Although it provides a considerable speed-up, in some situations it cannot allocate free blocks due to the limitation of its AND-OR tree structure. Cam *et al.* [7] proposed an efficient memory allocation system, which eliminates fragmentation and limitation in [4]. However, it uses more logic components compared to its predecessor.

In [9], an active memory module, which is connected to the same bus with a traditional RAM but used only for dynamic allocations, is proposed. It is a DRAM with low density but including an active memory processor in addition. The processor is used to keep the heap status and make garbage collection. The method used for dynamic allocation is based on the mechanism in [4]. The realization of the AND-OR tree with hardware description language is explained in detail in [8]. One step ahead Chang *et al.* [10] came up with a hardware memory allocator, which can be easily integrated to CPUs. It works in conjunction with an application specific instruction set extension.

[11] targets those systems, which have FPGA as a computational resource only. Some peripheral devices and memory is connected to FPGA and the management of memory is performed by FPGA. Free parts of the memory are kept in a stack as pages. When a request arrives, the page pointed by the stack pointer is allocated. However, there exist no results in terms of dynamic memory management metrics.

As a continuation of [7], VHDL synthesis of work is presented with minor improvements in [13]. The proposed OR-gate prefix circuit has more gates than AND-OR structure mentioned in [4]. The reason why it consumes more resource is due to the requirement to find any free block existing in the bitmap. However, Chang's AND-OR tree [4] can detect a free block of size j under the circumstance that the free part's starting address should be a factor of j or k x j, where $k \geq 0$ and j is a power of 2 [13]. The proposed scheme has been implemented on an FPGA and some performance results and comparisons with [9] were presented.

Another alternative for dynamic memory allocation in FPGAs is proposed in [14]. Dynamic memory allocation controller (DMAC) core has been developed to manage output buffers of communication nodes in a high performance FPGA cluster. Free and occupied blocks are placed on a binary tree and this structure is kept in a BRAM in FPGA. Adding and deleting nodes from the tree is achieved done via the DMAC core. When tree gets larger, search time for add and delete operations inevitably increase.

CHAPTER 3

DESIGN OF DYNAMIC MEMORY MANAGER

3.1 Design Approach

Recently, block RAMs of FPGAs have become sufficient to carry out many complex computations without going out of the chip [21]. However, block RAMs (BRAMs) of FPGAs should be used efficiently especially for computations that need dynamic management of the memory. From this point of view, our primary design goal is to implement a dynamic memory manager in FPGA, which has fast and bounded response with very low fragmentation. In order to achieve this goal, primarily the combination of two techniques, namely, segregated free lists and bitmapped fits are employed simultaneously. Free blocks grouped according to their sizes are stored in free lists using the segregated free lists approach. On the other hand, availability information about a block, i.e., whether the block is free or not, is represented as a flag in a bitmap vector. In order to keep memory overhead low bitmap vector is also implemented in the BRAM of FPGA. Different from other conventional memory allocators, the allocated memory to a request is not necessarily a contiguous block in our design. It can be dispersed on different non-contiguous blocks. However, such blocks are connected with link vectors that include encoded information about the allocated blocks. For allocation, a modified version of AND-OR tree in [4] is used. Our dynamic memory manager (DMM) consists of the following three main parts:

i) allocator,

ii) deallocator and

iii) address translator

The following sections present the conceptual design and its related components which is realized with VHDL synthesis.

3.2 Memory Representation

The memory area which is to be used as heap is partitioned into blocks and sub-blocks having strict boundaries (Figure-3.1). Heap area is first divided into main blocks of size $32x2^n$. *n* can be used as a parameter for arranging the block sizes in the main block. In this scheme, one can allocate blocks ranging from 1 byte up to $32x2^n$ bytes with a resolution of 2^n. Therefore, as will be detailed in Section-5.3, more than one DMMs having different *n* values can be combined to increase the range of sizes. These main blocks are partitioned logically further into sub-blocks as shown in Figure-3.1. Allocations will be done as a combination of different sized sub-blocks from the heap memory. For example for a request of $11x2^n$ bytes $(8+2+1) \times 2^n$ or $(4+4+1+1+1) \times 2^n$, sub-blocks can be provided.

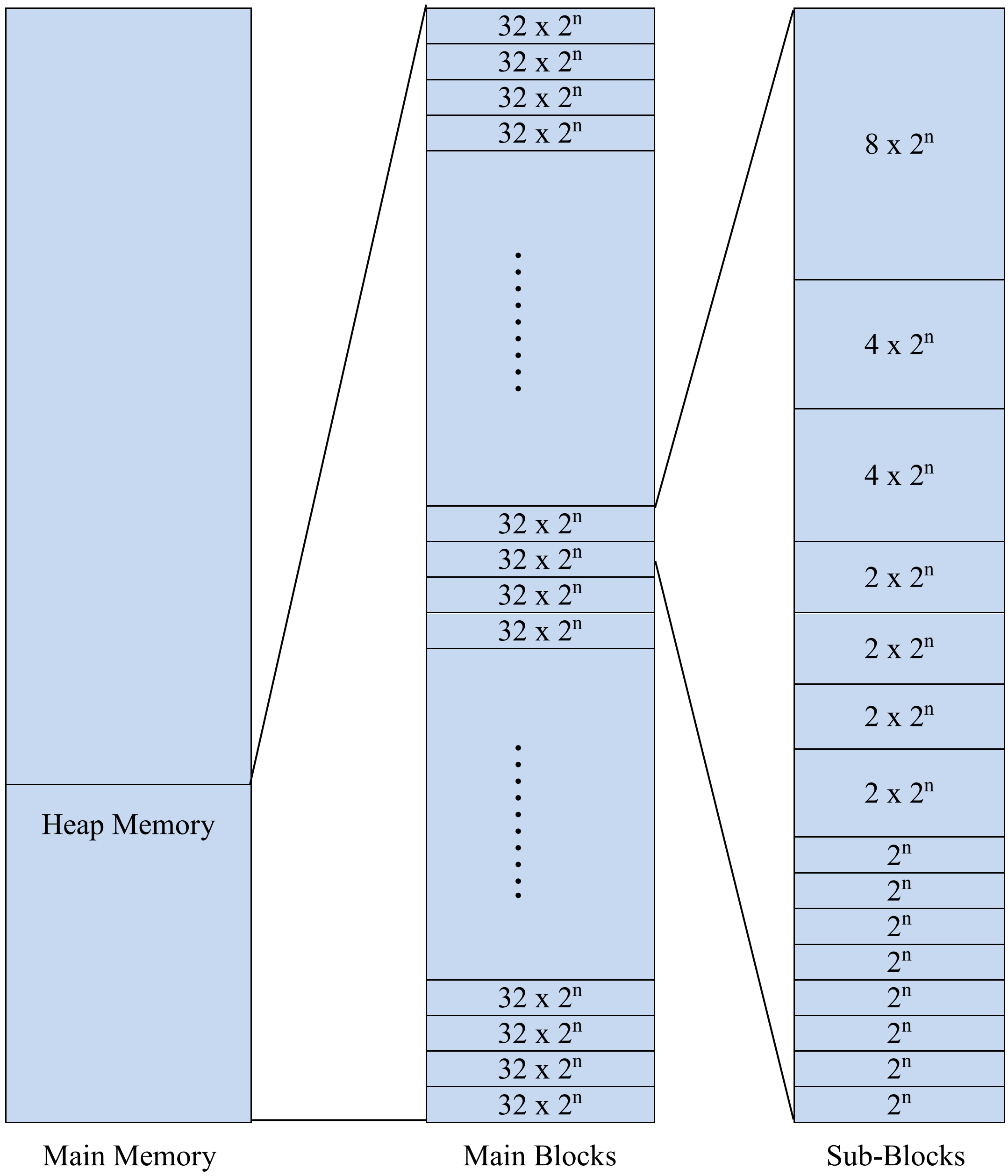


Figure 3.1: Heap Memory Partitioning

The motivation behind the choice of the sub-block distribution in Figure-3.1 is to provide an infrastructure, which is capable of allocating memory sizes ranging from 2^n to $32x2^n$ with 2^n increments.

3.3 Bitmap and Link Vector

As was mentioned earlier, whether a block is free or not is decided by checking this state information in the bitmap vector. Each bitmap and its associated link vector corresponds to only one main block of size $32x2^n$. Since every sub-block is represented as a flag, 15 bits are used for each main block. The vector also includes maximum number of available contiguous sub-blocks with the same size. Figure-3.2 illustrates the fields of the bit vector. This example bit vector of “1-01-1100-00101000-01-010-0011” is interpreted as follows: there exists one free block with size $4x2^n$, two free blocks with size $2x2^n$ (also contiguous) and six free blocks of size 2^n (however, 3 of them are contiguous).

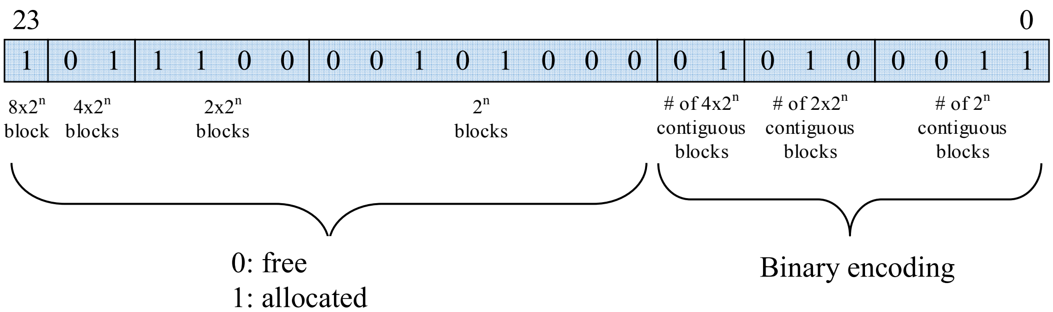


Figure 3.2: Bitmap Vector

Our second data structure is the link vector, which shows sub-block connections of an allocation (Figure-3.3). Allocation details are encoded in link vector. It includes all combinations that can be chosen using $8x2^n$, $4x2^n$, $2x2^n$ and 2^n sized blocks. There are 3 different fields in the link vector, namely *next block information*, *number of blocks with same size* and *next block starting address*. *Next block information* is encoded in 2 bits for $8x2^n$ and $4x2^n$ blocks and in one bit for $2x2^n$

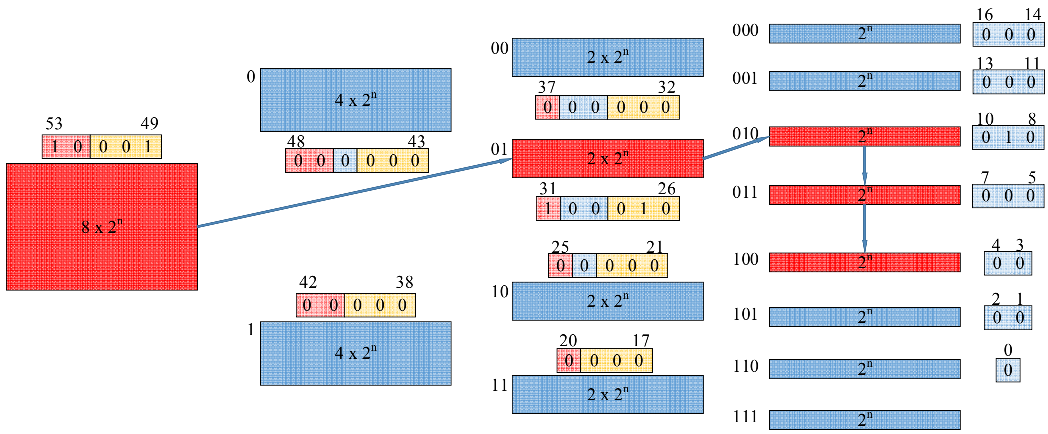
blocks because the next block of $8x2^n$ can be $4x2^n$, $2x2^n$, 2^n sized blocks or none. Thus, there are 4 possibilities for $8x2^n$ block, 3 possibilities for $4x2^n$ blocks and 2 possibilities for $2x2^n$ blocks. There is no next block for 2^n sized blocks.

Table 3-1: Next Block Encoding in Link Vector

Next / Current	$4x2^n$	$2x2^n$	2^n	No next block
$8x2^n$	"01"	"10"	"11"	"00"
$4x2^n$	-	"01"	"10"	"00"
$2x2^n$	-	-	"1"	"0"

The *number of blocks with same size* shows the number of used blocks that are contiguous in the same sub-block. For instance, there are eight *n* sized blocks. If four of the *n* sized blocks are used, there should be "011" in the corresponding field in the link vector. This indicates that three contiguous blocks following the starting block, i.e. four in total are allocated

Next block starting address is a 3-bit field for all the blocks because next block can be one of the eight 2^n sized blocks and they are addressed using 3 bits only. If the next block is a $4x2^n$ block, one least significant bit is sufficient to determine which one is the next block. Similarly when the next block is a $2x2^n$ sized block, two least significant bits are sufficient to determine the address of the block. Figure-3.3 illustrates an example memory request of size $13x2^n$ where the bitmap vector is as "0-11-1000-11000011-00-011-0100". The request is met by allocating the first available blocks of $8x2^n$, $2x2^n$, and three 2^n. Then the link vector becomes "10-001-XXXXXX-XXXXX-XXXXXX-1-00-010-XXXXX-XXXX-XXX-XXX-010-XXX-XX-XX-X" (where only the affected bits are shown). Link vector is then updated as a result of the allocation request and accessed for deallocation and address translation requests later.



=> Next block information

=> Number of blocks with same size

=> Next block starting address

Figure 3.3: Link Vector

3.4 Free Lists and BRAM Structure

As was previously mentioned, we use the segregated free lists approach in order to find the block with the requested size quickly. In this scheme, there are lists of blocks organized according to the available size in corresponding blocks. Free lists in this design are similar to that used in software based TLSF allocator in [12]. As seen in Figure-3.4, there are 32 free list FIFOs that keeps BRAM addresses. The content of an addressed BRAM is the bitmap and link vectors of the corresponding main block. Free lists are arranged according to maximum available free block size that can be allocated in return to a request. Their size range from 2^n to $32x2^n$.

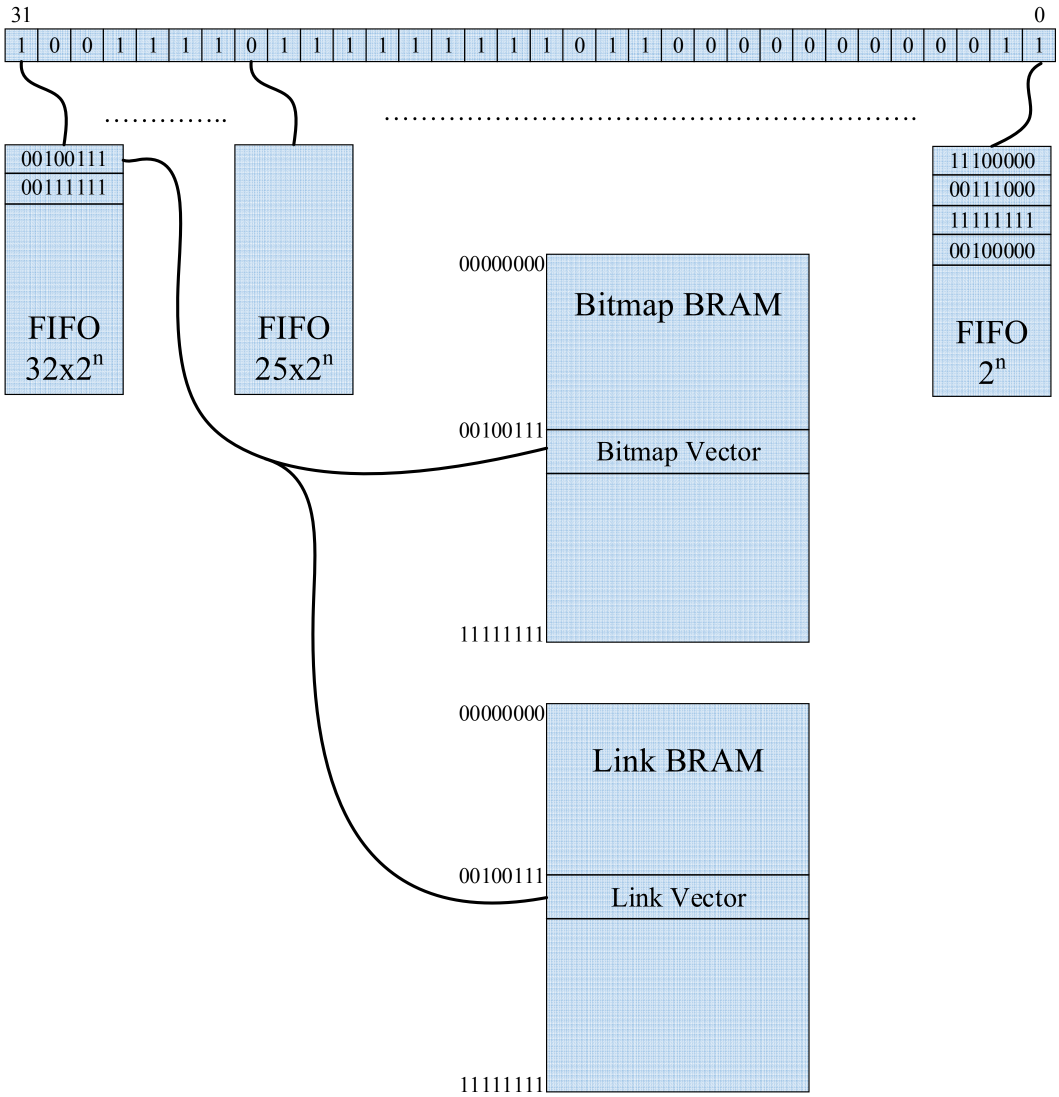


Figure 3.4: Free Lists and BRAM Structure

Besides the free list structure, there are BRAMs that store bitmap and link vectors. For a main block with size $32x2^n$, one word is reserved in link BRAM and bitmap BRAM each. The address of BRAM, which stores these vectors, is related to the starting address of the main block in the heap memory.

3.5 Hashing

In order to be able to use BRAM structures mentioned above efficiently, a simple hashing is applied to the actual address of the heap area in the main memory. Starting address of the heap memory should be given as a generic input to the DMM. Starting address of the heap partition is then regarded as an offset. The difference between the starting and ending address of the heap partition defines BRAM size for that partition. Since every word in the BRAM corresponds to a main block with $32x2^n$ (2^{n+5}) size, link and bitmap BRAMs have word length that is equal to heap memory size divided by the main block size. In other words, if the heap size is 2^k words, BRAMs will have $2^{k-(n+5)}$ words. The process is summarized in Figure-3.5.

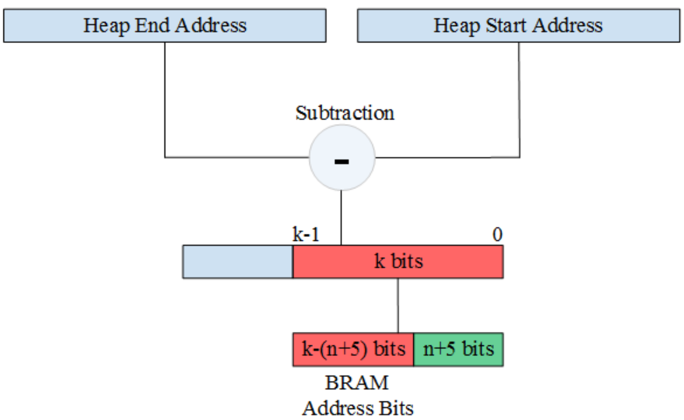


Figure 3.5: Address Hashing

3.6 Description of Subcomponents

3.6.1 Free List Manager

There are 32 free lists each of which keeps BRAM addresses of the bitmap and link vectors grouped according to corresponding free block sizes. These are FIFO structures with a word size of 16, which is the minimum number that can be created.

Obviously, free sub-blocks in a main block will be updated following an allocation or deallocation task. Therefore, free list FIFOs should be re-arranged according to the new condition. This task is realized by *free list manager* (FLM) in our DMM.

Managing free lists in the allocation task is relatively easy. When an allocation request arrives, the top element of the corresponding size FIFO is popped out. If it is empty, one greater size FIFO is used concurrently. Following the completion of the allocation process, BRAM address of the related block is pushed into the FIFO corresponding to new free size in that block.

On the other hand, when deallocation takes place it is not known whether the deallocated block's BRAM address is in the free list FIFO or not. Hence the corresponding size FIFO is emptied via a reset input.

The other task that free list manager performs is filling up the FIFOs. When there is no allocation or deallocation, FLM deals with pushing BRAM addresses to FIFOs. It starts with the first address, reads BRAM content and sends it to the related free list, which is empty, by checking the free size in that block. If there is an element in the corresponding FIFO, FLM does not push the new BRAM address. Instead, it goes to the next address and this process continues in the same fashion.

3.6.2 Allocator

DMM basically performs the task of reserving memory of the requested amount. The only parameter that should be provided to the allocator is the size of the requested memory. While performing this task, DMM benefits from hardware structures such as AND-OR tree similar to the one in [4] and some combinational blocks namely, the bit flipper, address conversion and size conversion blocks (Figure-3.6).

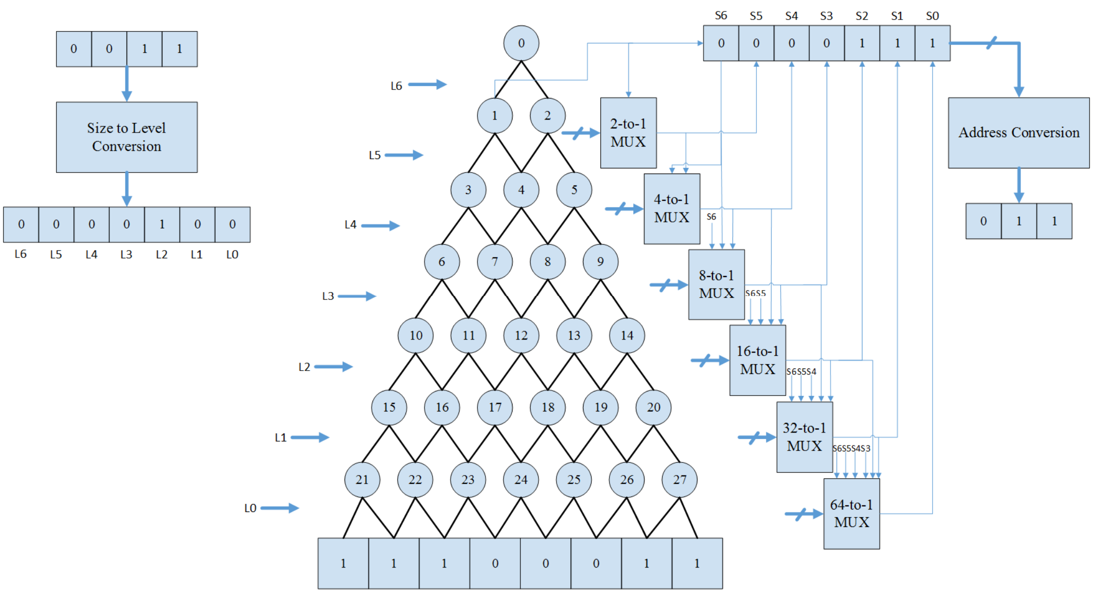


Figure 3.6: Modified AND-OR tree

Each node in Figure-3.6 is composed of and gates, or gates, multiplexer and D-type flip-flop. Similar to the node structure in [8] is used in the modified AND-OR tree nodes with a slight change. The node have been changed to eliminate combination gate delays. A flip-flop is added to output of the nodes. Thus, it provides the increase in the overall circuit operating frequency. In Figure-3.7, the inner structure of node number 15 is given as an example. Furthermore, tree structure is modified in this work to eliminate the shortcomings of Chang's AND-OR tree in [4]. Chang's AND-OR tree can detect a free block of size j under the condition that the free part's starting address is a factor of j or k x j, where $k \geq 0$ and j is a power of 2. However in the modified AND-OR tree, a requested blocks can be found anywhere in the given bitmap.

Modified AND-OR tree is used to determine the starting address of the sub-block (Figure-3.6). For every sub-block group, there are gate trees. Thus, the design has a total of three gate trees for different widths corresponding to 2, 4, and 8 bit width bitmap vectors (not needed for 1 bit). For example, there are four $2x2^n$ sub-block in a main block, therefore the corresponding 4 bit part of the bitmap vector is provided as input to the 4-bit width gate tree. The output will be the starting address of the sub-block to be allocated. Since the memory is partitioned according to a predefined rule, finding the actual starting address of the given memory block is straightforward. Then this is given as an input parameter to bit flipper with the allocated size. The corresponding bits are flipped to indicate that these blocks are not free anymore. Finally, FLM places the block to a new free list FIFO according to the maximum available size of blocks.

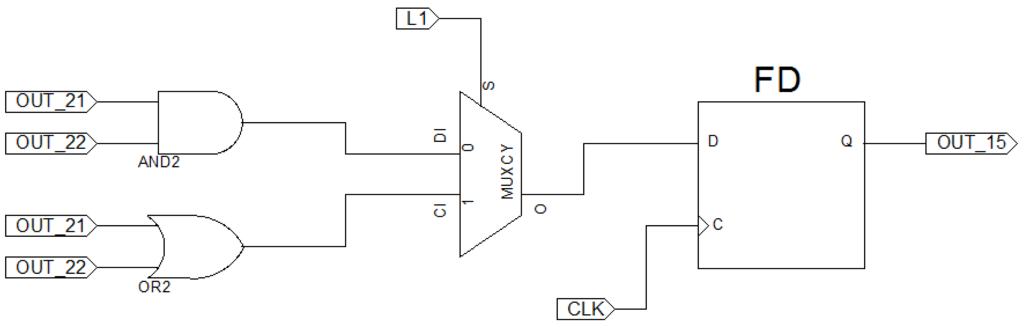


Figure 3.7: AND-OR structure

3.6.2.1 Allocation Process

The following tasks are performed within the context of the allocation process:

- Pick the non-empty FIFO with the requested size or more,
- Read the top element, which is the BRAM address of both bitmap and link vector

- Read the content of the BRAMs, send the bitmap vector to modified AND-OR trees
- After obtaining the starting addresses of sub-blocks, update the link and bitmap vectors
- Write the vectors to BRAMs
- Place the BRAM address according to free contiguous size to the corresponding free list FIFO

3.6.3 Deallocator

When the application does not need the allocated memory anymore, it releases the previously occupied part. Starting address of the memory block that will be freed is sufficient to perform this task. The size and link information of other blocks can be extracted from the link vector of the corresponding block. Then link and bitmap vectors are updated. Finally, similar to FLM section, it places the block to a new free list FIFO by checking the maximum available size of the block.

3.6.3.1 Deallocation Process

The following tasks are performed within the context of the deallocation process:

- Find the BRAM address from the provided starting address by hashing
- Read the BRAM content
- Find the link between blocks using the link vector
- Update the link and bitmap vectors
- Reset the corresponding size FIFO (size before the deallocation)
- Place the BRAM address according to new free size to corresponding free list FIFO

3.6.4 Address Translator

The obvious difference of our DMM from conventional memory allocators is the fact that the requested memory is not provided as a whole contiguous chunk. Instead, it may be provided as a combination of different blocks with varying sizes. Therefore, an extra task in DMM, i.e. *address translation* is required. In return to an allocation request, DMM sends smallest of the starting addresses of the allocated blocks. An application should only know the starting address of the object, the rest, i.e. link between blocks and total size, is in the DMM data structure. Thus, when an application demands to access the heap memory, a simple offset calculation won't be sufficient due to the non-contiguous nature of the system. Instead actual address should be calculated using the starting address and the offset in the DMM with a two cycle delay. At first, block combination is extracted from the link vector and then the actual address is returned using this and the offset value.

CHAPTER 4

IMPLEMENTATION OF DYNAMIC MEMORY MANAGER

In this chapter, implementation details of DMM are presented. It is implemented in VHDL. Xilinx ISE 14.6 [17] tool is used as the development environment. Specifications of the units that are developed in this design are described in the following sections.

4.1 Top Level

In Table-4.1, top level ports of the DMM are explained. Besides the ports, some generic values are shown in Table-4.2.

Table 4-1: Port definitions of DMM

Port Name	*Direction*	*Explanation*
Clk	IN	System clock
rst	IN	System reset
allocate	IN	When asserted with *alloc_size(23:0)*, allocator starts to search for available memory place as requested size
alloc_size(15:0)	IN	Size of requested memory
deallocate	IN	When asserted with *dealloc_addr(31:0)*, allocator frees the memory allocated before using starting *dealloc_addr(31:0)*

Table 4.1: Port definitions of DMM (Continued)

Port Name	*Direction*	*Explanation*
dealloc_addr(31:0)	IN	Starting address of the memory to be freed
find_address	IN	To find the address that the application wants to access
mem_addr_start(31:0)	IN	The starting address of the block that has the desired data in it. It uses *addr_offset(23:0)* to reach the desired address in the memory
addr_offset(23:0)	IN	The offset value that is used with *mem_addr_start(31:0)* to access desired data
alloc_done	OUT	Indicates that allocation is done successfully
mem_addr_return(31:0)	OUT	The starting address of allocated block. It is ready when the *alloc_done* signal is high
dealloc_done	OUT	Indicates that deallocation is done successfully
mem_addr_actual(31:0)	OUT	It is the actual address that the application wants to access. It is found using *mem_addr_start(31:0)* and *addr_offset(23:0)*
error	OUT	Indicates that an error has occurred
error_reg(7:0)	OUT	Type of the error

Table 4-2: Generic Values of Top Level

Generic Value	*Type*	*Explanation*
min_block_size	Integer	n value in minimum block of size 2^n
heap_start_address(31:0)	std_logic_vector	Starting address of the managed heap
heap_end_address(31:0)	std_logic_vector	Ending address of the managed heap

There are three main functionalities of the DMM. One of them is to reserve memory as the requested size. In order to start the process, *allocate* signal and *alloc_size(15:0)* should be applied. One of the free lists that is greater than or equal to the desired size is chosen. Top element of chosen free list FIFO, i.e. BRAM address, is popped out. Content of the bitmap and link BRAMs are read and sent for doing the necessary arrangements on the bitmap and link vectors. When the process is done, the smallest of the starting addresses of the allocated sub-blocks is returned as *mem_addr_return(31:0)*.

Second task is tp free memory area, which is not necessary any more. For this purpose, applying *deallocate* signal and *dealloc_addr(31:0)* is required. Obviously, dealloc_addr(31:0) is the address that has been sent by DMM when the allocation has been done. BRAM address that holds the corresponding bitmap and link vectors is found using hashing. From this point onwards, starting block and how they are linked are known. Thus, necessary bits on the vectors are flipped. Deallocation process is finally done after resetting the corresponding free list FIFO. This will be detailed more in the implementation of the free list manager.

The final task is about address translation in DMM. Since the provided area is non-contiguous, the application cannot access data using only starting address of the object. Linking of the blocks must be known and the offset calculation must be done accordingly. So, when the desired address in heap (*mem_addr_start(31:0)* +

addr_offset(15:0)) and *find_address* signals are provided to DMM, link vector should be found first as in the deallocation process. Afterwards, the actual address, i.e. *mem_addr_actual(31:0)*, is calculated and returned. This is completely independent from allocation and deallocation processes since it uses another other port of the link vector BRAM. Top level state flow is shown in Figure-4.1.

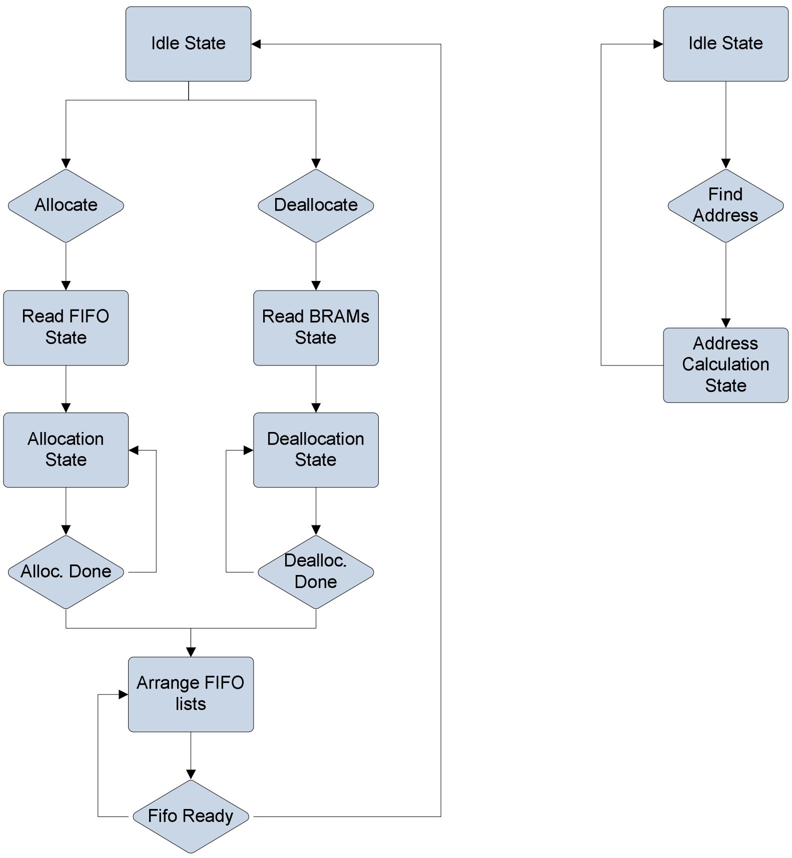


Figure 4.1: Top Level Flowchart

4.2 Free List Manager

4.2 Free List Manager (FLM) makes the necessary arrangements about free list FIFOs. It simply reads from or writes to the free list FIFOs in response to requests arriving from the top level. Besides these tasks, another important task is to reset the FIFO. When deallocation occurs, corresponding BRAM address could be in the free list FIFO. Following the deallocation, free size can be changed, so it should not stay in the previous free list FIFO.

Table 4-3: Port Definitions of FLM

Port Name	*Direction*	*Explanation*
clk	IN	System clock
rst	IN	System reset
wr_fifo	IN	*bitmap_bram_addr_in (15:0)* is written to the stated FIFO having the number *wr_fifo_number (4:0)*
rd_fifo	IN	*bitmap_bram_addr_out (15:0)* is read from the stated FIFO having the number *rd_fifo_number (4:0)*
find_in_fifo	IN	After deallocation it empties (resets) FIFO having the number *rd_fifo_number (4:0)*
wr_fifo_number(4:0)	IN	Shows which FIFO will be written
rd_fifo_number (4:0)	IN	States which FIFO will be read
bitmap_bram_addr_in(15:0)	IN	Bitmap and link vector BRAM address that will be written to FIFO
fifo_ready	OUT	States that the process is completed
fifo_full(31:0)	OUT	FIFO full signal, every bit corresponds to one FIFO
fifo_empty(31:0)	OUT	FIFO empty signal, every bit corresponds to one FIFO
bitmap_bram_addr_out(15:0)	OUT	Bitmap and link vector BRAM address that will be read from FIFO

FLM continuously loads the FIFOs if they are empty. It reads the bitmap BRAM consecutively and sends the address by checking the free contiguous size in it. If corresponding FIFO is not empty, it passes the next BRAM address. The interface of FLM is given in Table-4.3.

4.3 AND-OR Trees

AND-OR trees are used to determine free spaces of the heap memory using the bitmap vector that corresponds to a section of the heap memory. It is the modified version of the gate tree used in [4]. In [4], it provides a considerable speed-up, however it is not guaranteed to allocate free blocks in all cases due to the limitation of the used gate tree structure. This disadvantage is eliminated by using more resources in the present work. The ports of the AND-OR tree is shown in Table-4.4.

Table 4-4: AND-OR tree ports

Port Name	*Direction*	*Explanation*
clk	IN	System clock
rst	IN	System reset
bitmap_in(n:0)	IN	n+1 bits bitmap vector
level(n:0)	IN	Defined according to the requested size
free_address(n-1:0)	OUT	The number of '1's in this vector gives the starting address of the reserved area.

There are three 'n' values 1, 3 and 7 in the present design. These AND-OR gates are used for 2, 4 and 8 bits bitmap vectors. Although gate tree can be used as a combinational block, it is implemented as a clocked circuitry to prevent large gate delay. In order not to decrease the frequency of the overall design, the operation in the 8 bit gate tree is made to last in 7 clock cycles.

4.4 Address Conversion

Address conversion block is a simple one that converts AND-OR gates' free address output to an actual address. It simply checks the number of '1's in the input vector. For example, a free address output of 8-bit AND-OR tree "1100110" is converted to "100", which is a one clock cycle operation.

Table 4-5: Ports of Address Conversion Block

Port Name	*Direction*	*Explanation*
Clk	IN	System clock
rst	IN	System reset
free_address_8bit(7:0)	IN	Free address output of 8 bits tree
free_address_4bit(3:0)	IN	Free address output of 4 bits tree
address8(2:0)	OUT	Actual address of 2^n sized blocks
address4(1:0)	OUT	Actual address of 2 x 2^n sized blocks

4.5 Size to Level Conversion

Size to level conversion block converts binary size data to level information that can be understood by the AND-OR tree. For example, for the requested size of "0010" and "0011" from 2^n blocks, level information will be "00000010" and "00000100" respectively. Similar to address conversion, this is also a one clock cycle operation.

Table 4-6: Ports of Size to Level Conversion Block

Port Name	*Direction*	*Explanation*
Clk	IN	System clock
Rst	IN	System reset
size8(3:0)	IN	Requested size from 2^n sized blocks
size4(2:0)	IN	Requested size from 2 x 2^n sized blocks
level_8bit(7:0)	OUT	Level of 8 bits gate tree
level_4bit(3:0)	OUT	Level of 4 bits gate tree

CHAPTER 5

EXPERIMENTAL SETUP AND EVALUATION

5.1 Setup

Following the implementation of DMM, experiments are conducted to reveal the characteristics and performance of DMM. As the setup, a test computer and Xilinx KC705 demo board [18] shown in Figure-5.1 have been used. We prepared a simulator program in C# to communicate with the demo board via UART. Since the board has a USB to UART bridge, it can also be connected to the computer's USB port via its mini USB port. The demo board has Kintex-7 FPGA (XC7K325T) [19] which includes a large amount of logic resources.

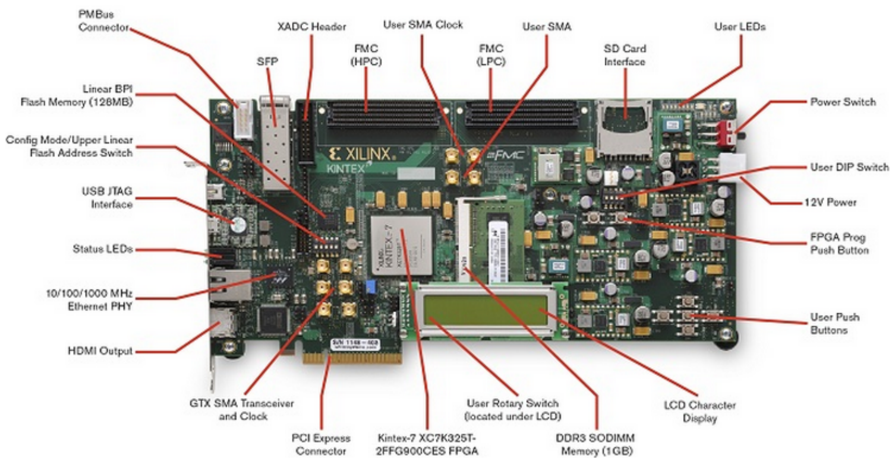


Figure 5.1: KC705 Demo Board

5.2 Characteristics of DMM

5.2.1 Logic Resources and Operating Frequency

Table-5.1 shows FPGA resource usage and the maximum operating frequency. In XC7K325T FPGA, LUTs and slices consumed for the DMM implementation correspond to 2% and 4% respectively while the minimum clock period is equal to 6ns corresponding to 166 MHz operating frequency.

Table 5-1: Resource Usage

LUTs	Slice	Max. Frequency
6075	2295	175,26 MHz

5.2.2 Memory Overhead

As was mentioned in the previous chapter, a data structure is used to keep track of the heap memory. For every main block there are two vectors that are bitmap and link vectors. Bitmap vectors are of 24 bits in length while link vectors are 54 bits in length for a main block of size $32x2^n$. These are kept in BRAMs of the FGPA. Therefore, heap size (HS) affects the overhead directly. Besides these, memory resources of the FPGA are also used for keeping the free list FIFOs, each of which occupies 16 BRAM addresses. However, we implemented these in the form of distributed RAM storage by using the FPGA's register sources instead of BRAM blocks. Therefore, memory overhead (MO) became

$$MO = (24 + 54)\ x\ \frac{HS}{32x2^n}\ bits$$

In synthesizing the FPGA 18Kbit memory blocks are employed as bitmap and link vector BRAMs. Thus, memory overhead for our DMM design in FPGA is given in the formula

$$MO = \left(\left\lceil \frac{HS}{32x2^n} \, x \frac{24}{18K} \right\rceil + \left\lceil \frac{HS}{32x2^n} \, x \frac{54}{18K} \right\rceil \right) x \; 18 \; Kbits.$$

For example, for n=3 and 512KB heap size, link and bitmap BRAMs have 2048 (512K/256) words. Therefore, first part of the formula becomes 3 and the second part becomes 6. The resulting memory overhead will then be 20.25 KB (9x18Kbits) for managing a 512 KB area.

5.2.3 Allocation Time

Allocation time is the time spent from incoming allocation request to the completion of the allocation process. As was previously mentioned in Chapter 3, popping out a BRAM address from a free list FIFO, reading the content of that address, processing the bit vectors and writing again to BRAM are the main tasks that are performed for the allocation. But it lasts no more than 21 clock cycles (Figure-5.2).

5.2.4 Deallocation Time

Deallocation time is the time spent for reading the content of the BRAM address to be freed, processing bit vectors and writing to BRAMs again. Deallocation time is bounded and lasts no more than 10 clock cycles (Figure-5.3).

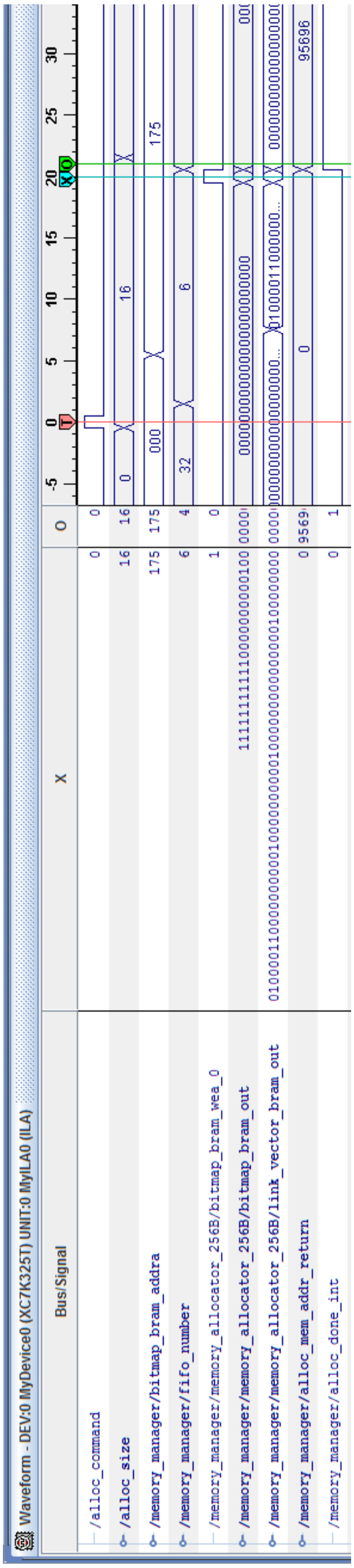


Figure 5.2: Allocation Process

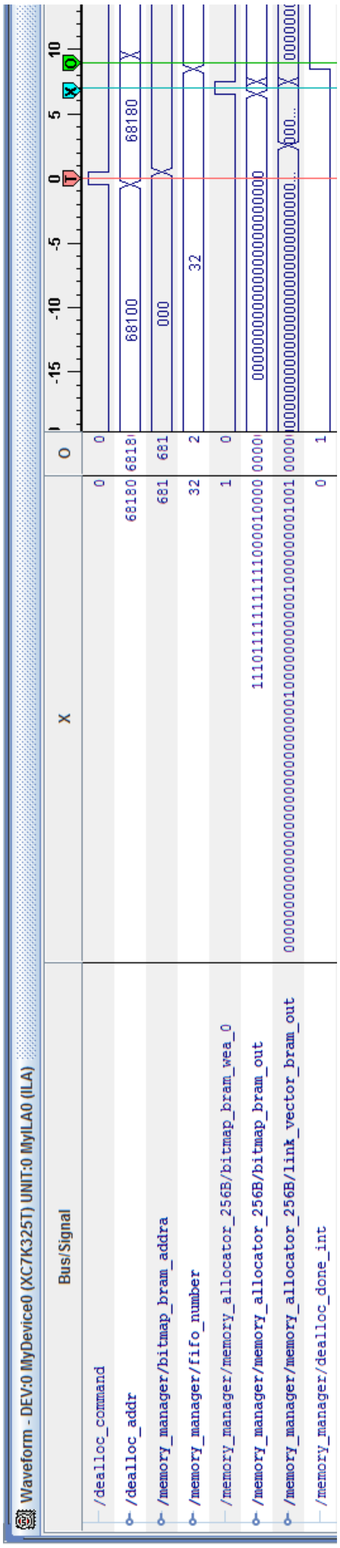


Figure 5.3: Deallocation Process

5.2.5 Experiments with Synthetic Trace

The configuration file, which is formed as a result of code implementation is loaded to KC705 demo board. It is connected via the USB port to the test computer. Computer and the board communicate via UART thanks to USB to UART bridge on the demo board. Then, a simple C# code is written to send commands and to gather replies from the DMM. The program sends 3 bytes (command code and size) as the allocation command, 5 bytes (command code and deallocation address) as the deallocation command (Figure-5.4).

Allocation

Command (0xAA)	Size (15:8)	Size (7:0)

Deallocation

Command (0xDD)	Address (31:24)	Address (23:16)	Address (15:8)	Address (7:0)

Figure 5.4: Serial Channel Commands

To record the effect of average block size to fragmentation, a set of synthetic allocation commands are sent to the DMM, consecutively. The sum of the allocated object sizes are recorded until the DMM cannot return an affirmative respond to an allocation request. The ratio of the total allocated places to the whole heap size shows to the unused memory area due to fragmentation. To create object sizes randomly, *Random()* function of the C# is used, which creates random numbers with a uniform distribution in a given interval. To provide various traces with different average allocation sizes three *Random()* functions are used. First one creates a uniformly distributed number between 0.0 and 1.0. Then, a second random function generates allocation sizes between 1 (minimum size) to average size and a third random function generates allocation sizes (average size + 1) to $32x2^n$ (maximum size). In this way, uniformly distributed traces having different average sizes are handled.

In the experiment, 512 KB of heap memory is managed using 2048 bitmap and link vectors, i.e. $n = 3$ and average sizes are selected in the range from 8 to 248 with increments of 8. Every random trace are sent to DMM for one hundred times. Memory usage at the point where DMM becomes irresponsive is recorded. Figure-5.5 presents the average percentage of unused memory to whole heap size due to fragmentation is shown.

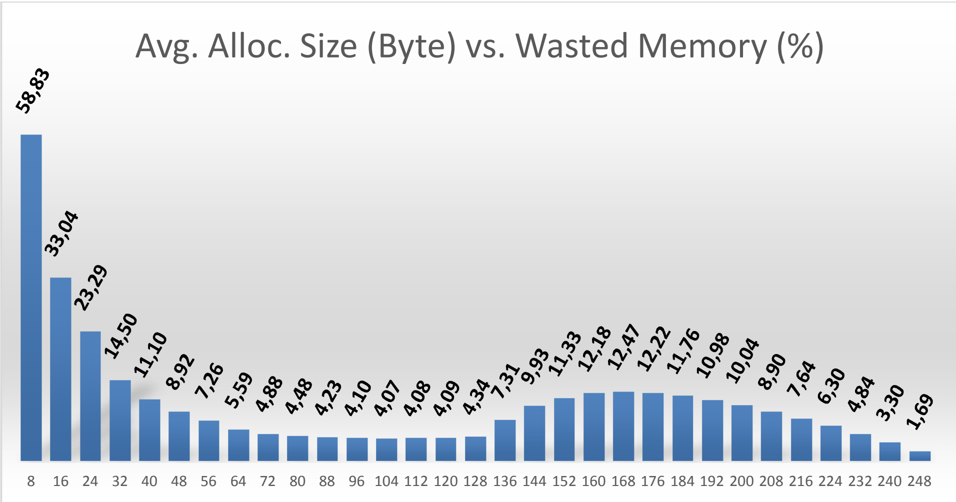


Figure 5.5: Percentage of wasted memory vs. average allocation sizes

It is worth emphasizing that this is an example used to demonstrate how fragmentation changes under a uniformly distributed memory request pattern. The values in Figure-5.5 may be different for other distributions. Similar graphs can be handled using different memory request distributions and accordingly the parameter n can be arranged to keep fragmentation low. For example, from the graph in Figure-5.5, we can choose $n=4$ (min. block size = 16) for a uniformly distributed trace with an average size of 176 bytes. For $n=3$ wasted memory percentage would be around 12.2% at 176 bytes. However, for $n=4$ it is around 4.2%, which is the fragmentation for $n=3$ case with 88 bytes.

5.3 Evaluation of the Design

5.3.1 Limitations

The proposed DMM in this work can allocate memory of size at most $32x2^n$. This value is flexible and depends on *n*, however it cannot exceed $32x2^n$. The partitioning of the heap memory is strict. Therefore, heap can be enlarged either by enlarging the bitmap and link vector BRAMs or by increasing the number *n*.

5.3.2 Scalability

Since the heap memory partitioning is not flexible, DMM can be scaled only by sacrificing some FPGA resources or fragmentation performance. One option to deal with a bigger heap memory is simply by increasing *n*. However, increasing *n* can cause more wasted memory. On the other hand, it may also cause a decrease in memory overhead percentage.

Another issue in scalability is the increase in the range of allocated sizes. For example, with $n=3$, allocation of 1 byte to 256 byte is possible. If another DMM is used with $n=8$, two of them can be linked in a pipelined manner. Hence, allocation size range can be extended to $32x2^8$, i.e., 8KB. In return, LUT usage in FPGA will be doubled and execution time will increase. Since two DMMs can work concurrently, execution time will not be doubled, but increase a few clock cycles only. On the other hand, memory overhead will drop as a percentage of the managed heap size. Lastly, the address translation mechanism is also affected. It would need an extra clock cycle for the translation of addresses to be accessed.

5.3.3 Memory Access Delay

Due to the non-contiguous reservation of the heap memory, our design includes an *address translator*, which computes the desired address. When the starting address and the offset are provided to access the desired data, firstly link vector BRAM is read using the starting address info. The result is the corresponding link vector, which keeps the information about block combinations. Then, the offset is added and the actual address is handled in the second cycle.

5.3.4 Comparison with Other Works

An alternative way of dynamic memory allocation in FPGA is proposed in [14]. Dynamic memory allocation controller (DMAC) core has been developed to manage the output buffers of the communication nodes in a high performance FPGA cluster. Free and occupied blocks are placed on a binary tree and this structure is kept in BRAMs of the FPGA. Adding and deleting nodes from the tree are done via DMAC core. When the tree gets larger, search time for add and delete operations increases inevitably. Compared to our DMM, execution time is much longer in DMAC. But it can allocate a broader range of object sizes.

Another work in [13] allocates a free block in 6 clock cycles. It uses approximately 12,600 LUTs for the allocator with a bit vector length of 512 bytes. Each bit of the vector represents one block and it can allocate a maximum of 64 blocks for an allocation request. It uses more FPGA resources than our DMM. In order to enlarge the managed heap area, two options arise: bit vector length and the block size. If the bit vector length increases to 1024 bytes, LUT usage will also be doubled. Instead of increasing the bit vector length, block size can be arranged to increase the managed heap memory size. However, it causes more fragmentation due to reduced granularity.

In DMMX [10], a scalable dynamic memory manager is proposed primarily for CPUs. It has a worst case allocation time of 96 clock cycles and uses a cache like architecture to keep bit vectors. It is claimed that bit vector of size 500 bits is sufficient to handle a cache hit ratio of 97%. Maximum allocation size in one request is bounded with the bit vector length times the block size. The only disadvantage of

DMMX when compared to DMM is the worst case allocation time for the requested object.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

Dynamic memory usage for high level synthesis tools still remains a hard problem to be implement well [15]. This thesis proposes a memory manager for dynamic memory allocations as a hardware IP core. By merging the bitmapped fits and the segregated free lists approach, the DMM tracks free and occupied parts of the managed memory. In achieving this, it aims to minimize the wasted space in the heap memory. Unlike conventional memory allocators, it satisfies memory requests by providing a combination of non-contiguous blocks to keep fragmentation at low levels.

The proposed DMM can be integrated to applications quite easily. Time spent for allocation and deallocation processes are 21 clock cycles and 10 clock cycles, respectively. Due to the non-contiguous nature of the reserved spaces, it adds a delay of 2 clock cycles when an application accesses to memory which is previously reserved by the DMM.

Bounded response time and low fragmentation are two major advantages of the proposed DMM while slightly increased access delay is the drawback. In terms of scalability, two cases should be regarded. One is increasing the heap size without changing the object size range by increasing the bitmap and link vector BRAM sizes. Performance metrics will not be influenced in this situation but memory overhead will increase. The second case is enlarging the object size range by increasing the *n* parameter. In this case memory overhead will drop, but wasted memory figures may be affected adversely..

It is believed that working towards eliminating the limitations mentioned in the previous chapter and decreasing memory access delay are two possible future directions to follow in this line of research. In order to increase object size range without affecting fragmentation, more than one DMM block can be used in a pipelined manner with a slight modification in the link vector. A next address area should be added to the first stage DMM to link the second stage DMM. In order to handle different object size ranges, n parameter should be different. For example, with $n=3$, allocation of 1 byte to 256 byte is possible. If another DMM is used with $n=8,$ two of them can be linked. Hence, allocation size range can be extended up to $32x2^8$, i.e., 8KB. As a result, LUT usage in FPGA will be doubled and execution time will increase a few clock cycles only. However, memory overhead will drop.

Another future work is to minimize the effects of memory access delay. For this, one can benefit from the memory access distribution of the applications. Generally, applications tend to access allocated objects sequentially. So, the next access of the application can be guessed and this avoids the two clock cycles delay for every access. Cache may also be used to keep the most recently accessed objects and their possible next addresses.

REFERENCES

[1] Wilson, P. R., Johnstone, M. S., Neely, M., Boles, D., "Dynamic storage allocation: a survey and critical review", Proc. of the Int. Workshop on Memory Management, September, 1995.

[2] Detlefs, D. L., Dosser, A., Zorn, B., "Memory allocation costs in large C and C++ programs", Software Practice and Experience, vol. 24, no. 6, pp. 527-542, 1994.

[3] Zorn, B., "The measured cost of conservative garbage collection", Software Practice and Experience, vol. 23, no. 7, pp. 733-756, July 1993.

[4] Chang, J. M., Gehringer E. F., "A high performance memory allocator for object oriented systems", IEEE Trans. Computers, vol. 45, no. 3, pp. 357-366, March 1995.

[5] Semeria, L., Sato, K., De Micheli, G., "Synthesis of hardware models in C with pointers and complex data structures", IEEE Trans. Very Large Scale Integration Systems, vol. 9, no. 6, pp. 743–756, December 2001.

[6] Wuytack, S., Da Silva, J., Catthoor, F., De Jong, G., Ykman, C., "Memory management for embedded network applications", IEEE Trans. Computer Aided Design, vol. 18, pp. 533-544, May 1999.

[7] Cam, H., Abd-El-Barr, M., Sait, S. M., "A high performance hardware efficient memory allocation technique and design", Proc. of the Int. Conf. on Computer Design, pp. 274–276, October 1999.

[8] Agun, S. K., Chang, J.M., "Design of a reusable memory management system", Proc. of the Int. ASIC/SOC Conference, pp. 369-373, September 2001.

[9] Srisa-an, W., Lo, C. D., Chang, J. M., "A performance analysis of the active memory system", Proc. of the Int. Conf. on Computer Design, pp. 493–496, September 2001.

[10] Chang, J. M., Srisa-an, W., Lo, C. D., Gehringer E. F., "DMMX: dynamic memory management extensions", The Journal of Systems and Software, vol. 63, no. 3, pp. 187–199, September 2002.

[11] Danne, K., "Memory management to support multitasking on FPGA based systems", Proc. of the Int. Conf. on Reconfigurable Computing and FPGAs, September 2004.

[12] Masmano, M., Ripoll, I., Crespo, A., Real, J., "TLSF: a new dynamic memory allocator for real time systems", Proc. of Euromicro Conf. on Real Time Systems, pp. 79–88, June 2004.

[13] Karabiber, F., Sertbas, A., Ozdemir, S., Cam, H., "An efficient memory allocation algorithm and hardware design with VHDL synthesis", Int. Journal of Electronics, vol. 95, no. 2, pp. 125–138, February 2008.

[14] Rajasekhar, Y., Sass, R., "A first analysis of a dynamic memory allocation controller (DMAC) core", Symposium on Application Accelerators in High Performance Computing, pp. 64-67, July 2011.

[15] Winterstein, F., Bayliss, S., Constantinides, G.A., "High level synthesis of dynamic data structures: a case study using Vivado HLS", Int. Conf. on Field Programmable Technology, pp.362-365, December 2013.

[16] Randell, B., "A note on storage fragmentation and program segmentation", Communications of the ACM, vol. 12, no. 7, pp. 365-372, July 1969.

[17] Xilinx ISE Design Suite, http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm, last visited on May 2014.

[18] Xilinx KC705 Evaluation Kit, http://www.xilinx.com/products/boards-and-kits/EK-K7-KC705-G.htm, last visited on May 2014.

[19] Xilinx Kintex-7 FPGA Family, http://www.xilinx.com/products/silicon-devices/fpga/kintex-7/index.htm, last visited on May 2014.

[20] Johnstone, M. S., Wilson, P. R., "The memory fragmentation problem: solved?", Proc. of the Int. Symp. on Memory Management, pp. 26-36, October 1998.

[21] Bacon, D. F., Cheng, P., Shukla, S., "And then there were none: a stall-free real-time garbage collector for FPGAs", Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp. 23-34, June 2012.

[22] Say, F., Bazlamacci, C. F., “A reconfigurable computing platform for real time embedded applications”, Microprocessors and Microsystems, vol. 36, no. 1, pp. 13-32, September 2011.