Introduction

Dynamic memory allocation has been an interest in research for decades. Many researchers have been interested in managing memory requests of applications at runtime. Static allocation of RAM resources requires designers to analyze the memory needs of each component, which can lead to poor memory efficiency and is infeasible in a dynamically changing system. Dynamic memory allocation is one solution to this problem but is usually only supported in software [1]. Most languages have a dynamic memory allocation mechanism, often invoked with an allocation method such as malloc( ) in C or by using a special operator such as “new” to request that enough memory for a particular object is allocated. Using this approach, it is possible to request exactly the right amount of memory as and when needed. Generally, static memory reserves the maximum memory than needed by the application. Even in FPGA applications, BRAM is the resource that saturates faster that other resources like FFs, LUTs, DSPs. This dynamic memory allocation is mostly handled by the compiler which consumes 23-38% of the time in the allocation-intensive program [2]. The use of fast and efficient allocation schemes is necessary to improve the overall performance.

There are different types of existing systems for dynamic memory allocation in hardware. The most popular systems are Buddy allocator, bit Vector and trees of the and-or gate. Binary buddy system[3] is one of the common approaches. In binary buddy systems, an allocation request is processed by repetitively splitting the memory block in half in order to create a memory block just large enough for the request. Memory blocks always have a size that is a power of 2, and the resultant two blocks from a split are buddies. If both buddies are free, they will be merged to form a block double their size. The binary buddy system has the advantage of simple computation of memory blocks addresses because blocks are always splitting in half. It also has the drawback of high internal fragmentation caused by rounding up the request size to size in power of 2.

A bit-vector is used to track the memory usage where each bit represents the smallest size block, with empty blocks represented as value 0. To allocate memory, the allocator firstly checks for a free memory block at least as large as the requested size. If such a block is available, the allocator finds the address of the memory block. In the final step, the allocator flips corresponding bits in bit-vector to mark the allocation. The tree structure of the and-or gate is shown in figure 1. The tree structure has a large overhead while storing the data using a tree data structure.

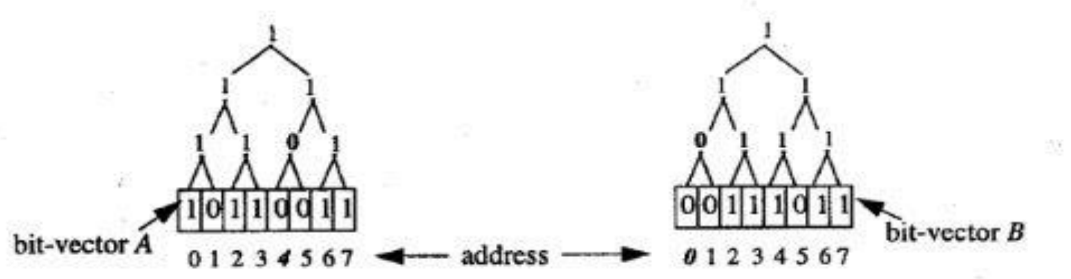

Figure 1. Tree of and-or gate

There are different policies used for dynamic memory allocation. One of the policy is the sequential fit. It is a linked list for keeping free blocks. Since it is a linked list, one free block points to another block forming a chain of sequence. The sequential fit is categorized into the best fit, first fit and next fit. Another fit is segregated fit where splitting is enabled when the requested size is not found. Bitmapped fits are also used in a similar fashion.

Methodology

We propose a solution for the Dynamic Memory Management (DMM) unit on an FPGA hardware to efficiently use BRAM resources using the block diagram shown in Figure 2.

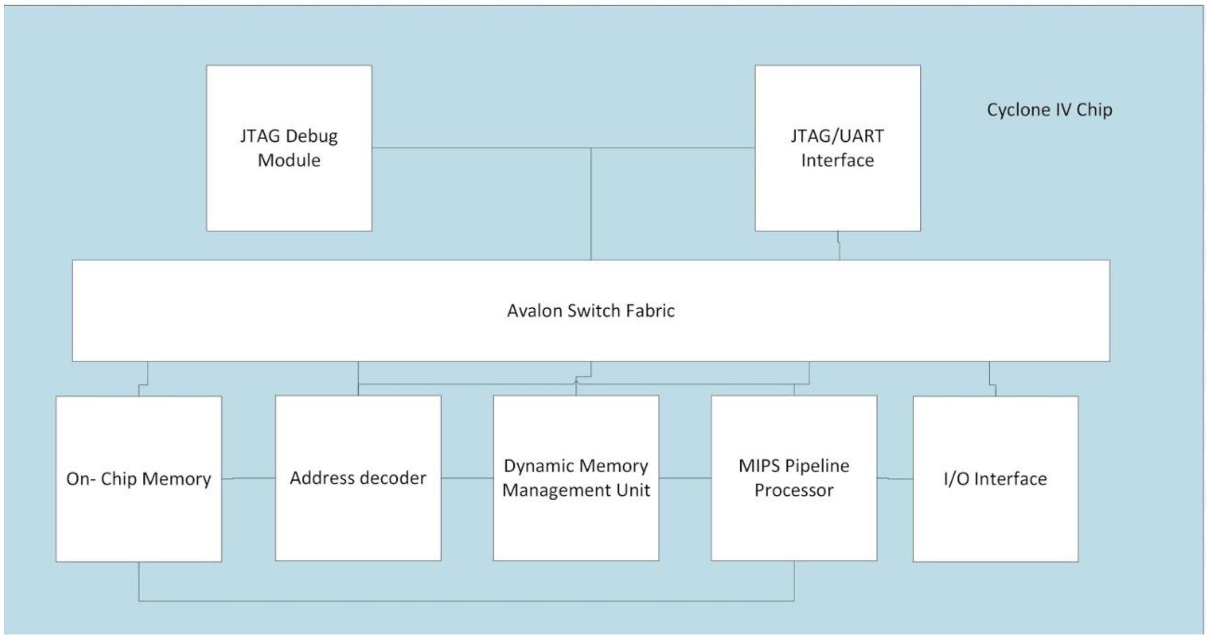

Figure 2. Dynamic Memory Management Unit for Cyclone IV FPGA platform

The DMM in figure 1 consists of 3 modules:

1. Allocator
2. Deallocator
3. Address decoder

The heap memory is logically divided into N blocks each of 2^k size is shown in Figure 3. Heap memory is divided into several blocks in order of power of 2 like 2, 4, 8, 16, 32 and 64. A table of allocated/deallocated bit vectors with depth*width N*N is maintained (Table 1). The starting address of each block is stored in a separate table. The bit is flipped from 1 to 0 if it is freed and 0 to 1 if it is occupied. Requested memory size is searched according to first fit policy. If a block with capacity 2^b can no longer accommodate 2^b, then bit corresponding to that block and that size is changed to 1 as shown in Table 1.

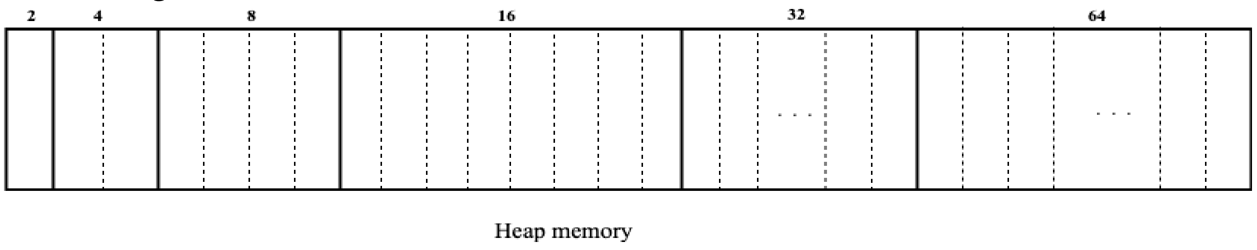

Figure 3: Heap memory

2	0	2	0	2	0	2	0
4	100000	4	100000	4	100000	4	100000
8	110000	8	11**1**000	8	11**1**000	8	11**11**00

16	111000	16	111000	16	111**1**00	16	111**1**00
32	111100	32	111100	32	111100	32	111100
64	111110	64	111110	64	111110	64	111110

Table 1: Allocated/Deallocated table for 3 consecutive request size of 7

The overall block diagram for memory allocation is shown in Figure 4. For this process, we have blocks like the address generator, one hot encoder, allocation status register, priority encoder and the registers to keep track of address that is used for allocation or de-allocation. The policy that we have used on our system is a parallel fit.

The address generator receives the 6-bit input from MIPS processor and generates the 3-bit output from the requested memory size. It is passed to the one hot encoder. The output from one hot encoder and allocation status register is ANDed together for the parallel search of free space. The free space of the smallest possible block is allocated for the requested size and is provided to the priority encoder. The priority encoder generates the 3-bit output which is the address of the selection line to find out the remaining size in the heap memory. We also have comparator to compare the bits and allocate the memory as requested. The two other registers are used to store the address that is used for allocation and de-allocation.

The deallocation stage is similar to the allocation stage. Whenever processor issues a free signal, a *d_en* signal is issued by the DMM controller. On receiving the *d_en* signal the deallocation block of DMM looks at the value in the register and the register itself so that it can go to the corresponding location in allocation start register and starting address register to change the status to free.

Since garbage collection and reference counting methods of memory allocation require us to keep track of every object and variable being referenced, the memory overhead for such techniques become very large and hence implementing them on memory constrained environment is difficult. So, we use explicit memory management technique where we explicitly need to free up allocated heap memory preventing potential memory leaks.

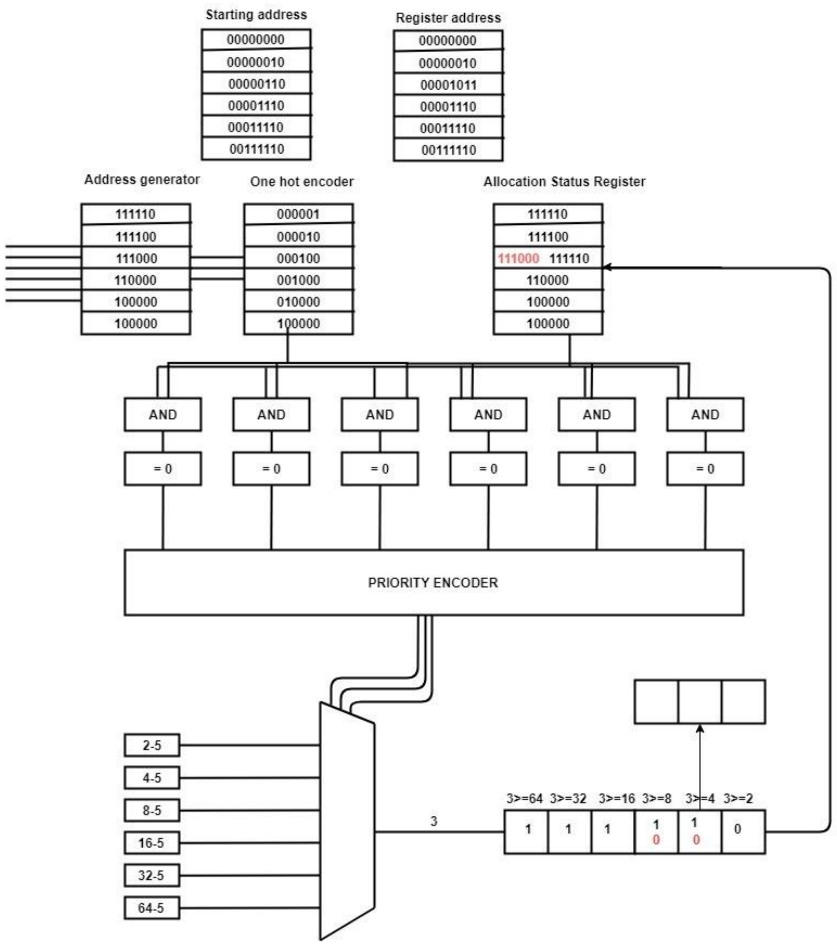

Figure 4. Block Diagram for memory allocation

The state diagram of our system is shown in Figure 4. Our system has three states s1, s2, and s3. At reset, we are at initial state s0. Malloc/Free signal from the MIPS process enables the controller and moves to next state ie.s1. When our updated allocation status register is ready then the up-ready signal is provided by the system, then the corresponding write signal to the register is given and moves to the next state ie. s2. Finally, memory allocation acknowledgment (mack) is sent back to the processor to tell the processor that the allocation is successful.

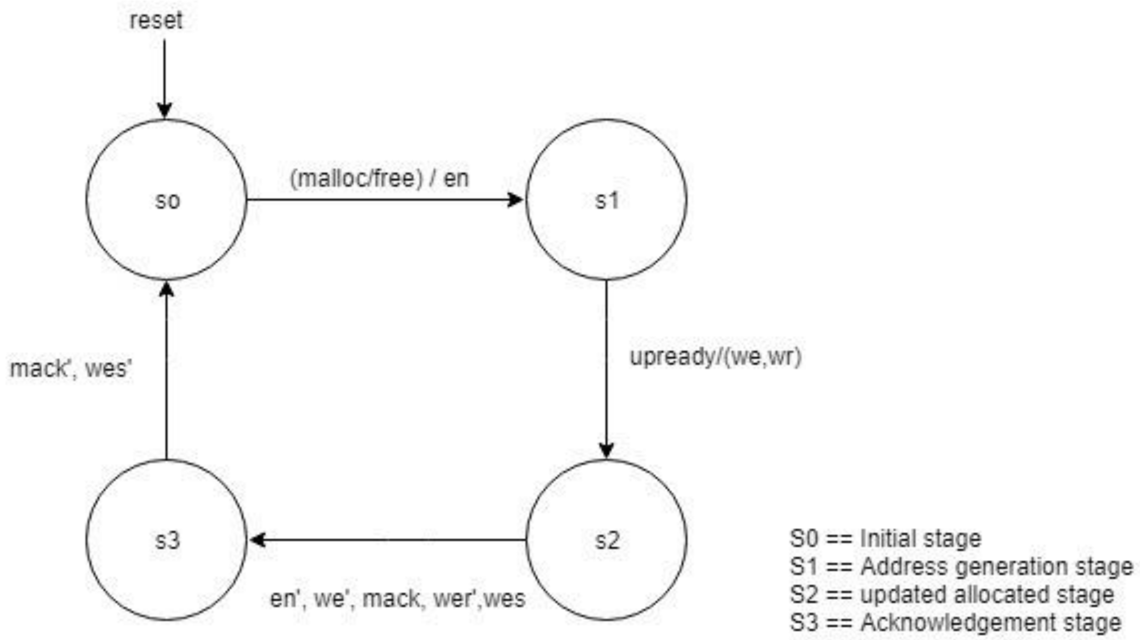

Figure 4. State Diagram of the system

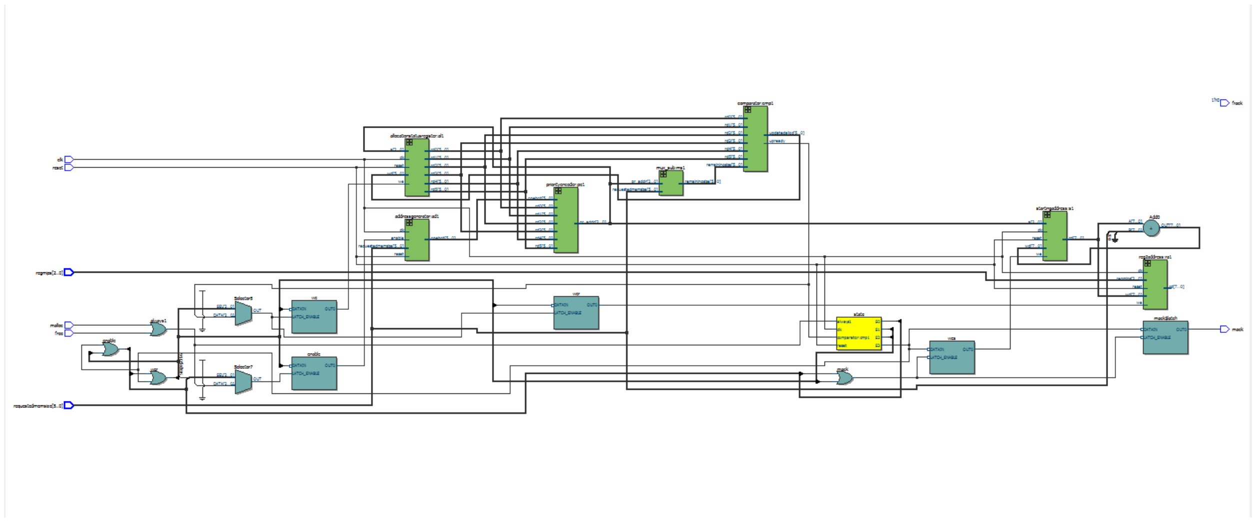

Figure 5. Quartus generated schematic for Dynamic memory allocation module

Simulation and results

A simple program to display ‘n’ Fibonacci numbers was used to test our application, where n is a number given by the user at run time.

Opcode generated for the malloc and free are as follows

```
6’b000111:controls <= 11’b00000000010;//malloc

6’b000110:controls <= 11’b00000000001;//free
```

We have tested the system by providing the memory request of 5 which is shown in Figure 6.

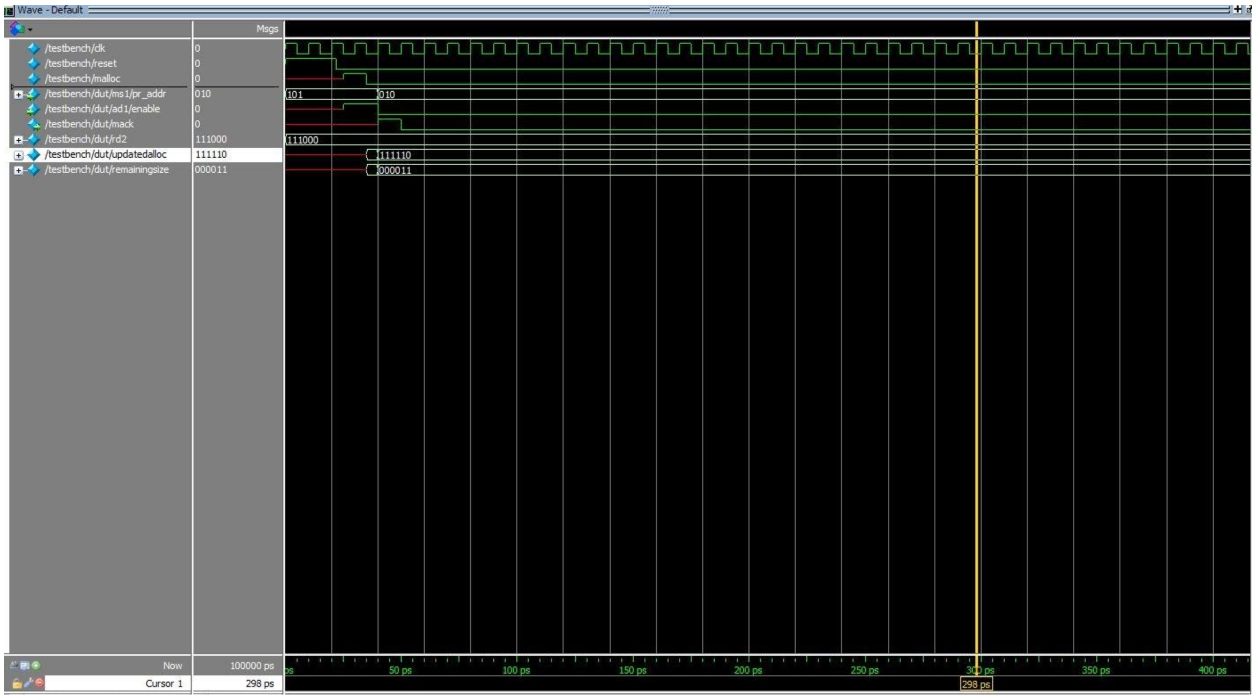

Figure 6: Simulation result for memory allocation request of size 5.

From Figure 6, we can see the request of 5 (101) is sent to the system and the address generator selects 111000 address and the updated allocation is 111110 which is expected and is exactly same to the block diagram example.

Conclusion

A hardware allocation and deallocation system are introduced with the following features.

- Fragmentation: Internal fragmentation rate is lower but external fragmentation increases if large memory size requested multiple times

- Memory overhead required by the system is less.
- Scalable: The address decoder, as well as the encoders used in the system, can be scaled up with increasing memory blocks.
- The system can be synthesized to be incorporated with high-level synthesis-IP cores.
- It can be interfaced with the SDRAM controller to access virtual memory.

Future Improvements

The current system can run only if there are a few instructions between malloc and actual memory allocation. We can find a way to stall the processor for executing writing to memory immediately for future implementation.

References

[1] Zeping Xue and D. B. Thomas, "SysAlloc: A hardware manager for dynamic memory allocation in heterogeneous systems," 2015 25th International Conference on Field Programmable Logic and Applications (FPL), London, 2015, pp. 1-7.

[2] Cam, H., Abd-El-Barr, M., & Sait, S. (1999). A high-performance hardware-efficient memory allocation technique and design. Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No.99CB37040), 274-276

[3] K. C. Knowlton, "A Fast Storage Allocator", Communications of the ACM, vol. 8, pp. 623-624, 1965.

Appendix

Test code in assembly

```
ldw $3,0($0) #input memory size at run time
malloc $3   #requested memory size of ($3) to heap
subi $10,$3,2  #counter for fibonacci
addi $11,$0,0  #first element of fibonacci
addi $12,$0,1  #second element of fibonacci
addi $13,$0,1
lw $15,0x7fff($0) #starting address of heap returned by dmm register
```

```
sw $11,0($15)  #store 1st element in starting address of heap

addi $15,$15,4  #increase heap memory by 4

sw $12,0($15)  #store 2nd element in starting address of heap

addi $15,$15,4  #increase heap memory by 4


loop:

beq $10,$0,exit #exit when request size is done

sub $10,$10,1  #decrease counter

add $14,$11,$12 #add previous 2 elements

sw $14,0($15)  #store new element of series into heap

addi $15,$15,4 #increase heap by 4

add $11,$12,$0  #update previous elements

add $12,$14,$0

j loop

exit:

free $3  #explicitly free up the heap
```

corresponding machine code

```
8c030000

1C600000

00715022

200b0000

200c0001

200d0001

8c0f7fff

adeb0000

21ef0004

adec0000

21ef0004

11400007

014d5022

016c7020

adee0000
```

```
21ef0004

01805820

01c06020

0800000b

18600000
```