

A Dynamic Memory Management Unit for Embedded Real-Time System-on-a-Chip

Mohamed Shalan
Georgia Institute of Technology
School of Electrical and Computer Engineering
801 Atlantic Drive
Atlanta, GA 30332-0250
(770) 757-6772
shalan@ece.gatech.edu

Vincent J. Mooney III
Georgia Institute of Technology
School of Electrical and Computer Engineering
801 Atlantic Drive
Atlanta, GA 30332-0250
(404) 385-0437
mooney@ece.gatech.edu

ABSTRACT
Dealing with global on-chip memory allocation/de-allocation in a dynamic yet deterministic way is an important issue for upcoming billion transistor multiprocessor System-on-a-Chip (SoC) designs. To achieve this, we propose a new memory management hierarchy called Two-Level Memory Management. To implement this memory management scheme – which presents a paradigm shift in the way designers look at on-chip dynamic memory allocation – we present a System-on-a-Chip Dynamic Memory Management Unit (SoCDMMU) for allocation of the global on-chip memory, which we refer to as level two memory management (level one is the operating system management of memory allocated to a particular on-chip processor). In this way, heterogeneous processors in an SoC can request and be granted portions of the global memory in twenty clock cycles in the worst case for a four-processor SoC, which is at least an order of magnitude faster than software-based memory management. We present a sample implementation of the SoCDMMU and compare hardware and software implementations.

Keywords
System-on-a-Chip, dynamic memory management, real-time systems, embedded systems, SoCDMMU, two-level memory management.

1. INTRODUCTION
In the next five years it will be possible to fabricate integrated circuits with close to one billion transistors on a single chip [11]. Such chips will no longer be individual components of a system but "silicon boards." A typical System-on-a-Chip (SoC), as shown in Figure 1, will consist of multiple *Processing Elements* (PE's) of various types (i.e., general purpose processors, domain-specific CPU's such as DSP's, and custom hardware), large memory, analog components and digital interfaces [1][5]. Architecture such as this will be suitable for embedded real-time applications. Such applications – especially multimedia – require great processing power and large volume data management [6][12].

Management of the memory of a large SoC with heterogeneous processing elements and significant on-chip memory requires sophisticated analysis and optimization. One of the issues that the designer must take care of in an SoC is the allocation of the large global on-chip memory between the PE's. Will the allocation be static (i.e., determined at compile time), or dynamic (decided at run-time and capable of being changed from one moment to another during operation)? Most previous research in embedded systems has focused on static allocation and how to synthesize memory hierarchies for an SoC [12]. For applications whose memory requirements change significantly during run-time, static allocation of memory makes the on-chip memory utilization inefficient and makes system modification after implementation very difficult [2][10]. On the other hand, dealing with memory allocation between the PE's in a dynamic way can make the memory utilization more efficient. Also, the memory allocation will be programmable and can be changed at any moment depending on the system load. From the general-purpose end of the spectrum, there has been significant research in shared memory multiprocessing [8]. However, in shared memory multi-processing, dynamic memory allocation is not deterministic and typically requires hundreds or thousands of clock cycles in the worst case [7], which makes satisfaction of real-time constraints on such shared memory architectures difficult if not impossible.

Example 1 Consider a handheld device that can be used for communication as well as other personal applications (e.g., an Audio Player) like the example that is described in [9]. This device may be based on a multiprocessor SoC like that shown in Figure 1. Assume the device uses Orthogonal Frequency Division Multiplexing (OFDM) for communication. When used for communication, the OFDM processing is performed by the on-chip processors (e.g., DSP1 performs an FFT and DSP2 performs the rest of the OFDM processing). If the user switches from an audio player application (that does **not** use OFDM) to a communication application (that **does** use OFDM), then the switching time should be fast enough to receive the incoming data packets. Specifically, memory has to be rearranged (dynamically de-allocated and re-allocated) fast enough for the OFDM application to properly interpret the incoming OFDM symbols.□

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'00, November 17-19, 2000, San Jose, CA.
Copyright 2000 ACM 1-58113-338-3/00/0011…$5.00.

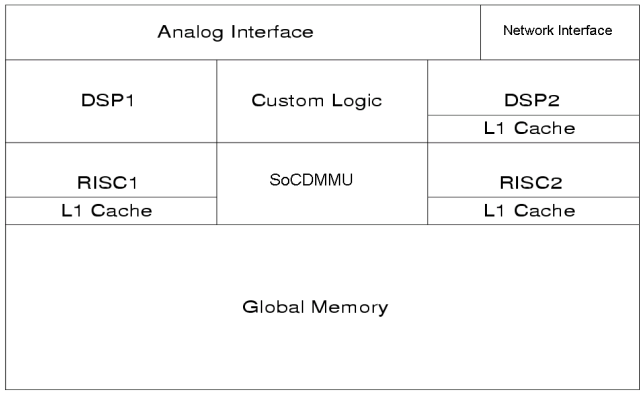


Figure 1. Example of A Billion-Transistor SoC

In this paper we describe a novel approach for memory allocation/de-allocation between PE's in an SoC that is suitable for real-time applications. Such systems require fast deterministic behavior that cannot be provided by general-purpose computer systems. The approach focuses on implementing a special hardware SoC Dynamic Memory Management Unit (SoCDMMU) to dynamically allocate the large global on-chip memory between the PE's. Note that after the SoCDMMU allocates a portion of the large global on-chip memory to a particular PE, the PE itself manages the use of this memory by its processes/threads. The SoCDMMU allows fast and deterministic dynamic memory allocation/de-allocation of the large global on-chip memory between the PE's.

The paper is organized as follows: Section 2 will describe the programming model. Section 3 describes the SoCDMMU. Section 4 discusses experimental results of our approach. Finally, the paper will be concluded in Section 5.

2. THE PROGRAMMING MODEL

We propose a programming model and memory management scheme, which we call *Two-Level Memory Management*. Two-Level Memory Management assumes that the SoCDMMU handles the allocation of the global on-chip memory while each PE handles the local dynamic memory allocation among the threads/processes running on the PE, for example with a Real-Time Operating System (RTOS) (a hardware accelerator can be used to accelerate the memory allocation/de-allocation at the threads/processes level [4]). The SoCDMMU manages distribution of memory between the PE's. On the other hand, each PE manages the usage of memory by the processes that run on that PE. So, typically, if a process requests a memory allocation it will request it from the RTOS. If the PE has currently allocated enough extra global memory to satisfy the request, the RTOS will simply allocate the memory right away; otherwise, the RTOS will request more memory from the SoCDMMU. Thus, there are two levels: the process/thread level managed by the RTOS (local allocation), and the PE level (global allocation) managed by the SoCDMMU. Before going further through the rest of the paper we will first state some assumptions on which we base our approach:

- The application running on the SoC fits (instruction and data) in the global on-chip memory.
- The global memory is divided into a fixed number of equally sized blocks (for example, 16KB or 64KB).
- Global memory allocation done by the SoCDMMU is referred to as *G_allocation*; global memory de-allocation done by the SoCDMMU is referred to as *G_deallocation.*
- A *page* consists of one or more blocks.
- Each memory block has one physical address and one or more virtual addresses. The block virtual address may differ from one PE to another.
- The block virtual address is referred to as PE-address.
- Each PE may request dynamic *G_allocation* of a page.
- Multiple PE's may issue *G_allocation* or *G_deallocation* commands simultaneously.
- The PE's can G_allocate three types of memory pages:
    - *Exclusive Memory*: Only the owner (the PE that G_allocates it) can access the G_allocated memory. No other PE can access the page. We will refer to the command that G_allocates this type of pages as *G_alloc_ex*.
    - *Read/Write*: The PE which G_allocates the page can read from or write to the memory. Another PE can read from the page if that PE G_allocates the page as read only. No other PE may G_allocate the same memory as *Read/Write* or as *Exclusive* Memory. This limitation significantly reduces cache coherency problems. We will refer to the command that G_allocates this type of pages as *G_alloc_rw*.
    - *Read Only*: The PE can read from (but not write to) the page. The page must be G_allocated as *read/write* by a different PE. We will refer to the command that G_allocates this type of pages as *G_alloc_ro.*

To handle the G_allocation and G_deallocation in programming languages (C/C++, etc.), the normal functions for G_allocation and deallocation (*malloc()* and *free()*) will be used without any change in the syntax. The only modification is to add a comment after the memory G_allocation function to distinguish between the different G_allocation commands discussed before. Figure 2 shows how that can be done in the C language. A preprocessor analyzes the source code and replaces the memory management functions with suitable code according to the comment inserted in the source code. The preprocessor also generates unique ID's to handle the *G_alloc_rw* and *G_alloc_ro* commands. These ID's are stored in the SoCDMMU and are used to resolve the addresses of the shared blocks at run-time.

```
p1=malloc(2*BLK_SZ);    /*DMMU G_alloc_ex(2)*/
p2=malloc(4*BLK_SZ);    /*DMMU G_alloc_rw(4,XBUF)*/
p3=malloc(4*BLK_SZ); /*DMMU G_alloc_ro(XBUF)*/
```

Figure 2. How to use the SoCDMMU in the C language

We deal with the cache coherency problem that may exist in the pages that are G_allocated as read/write using a simple cache invalidation scheme. Note that, since only one processor may allocate any memory as read/write (with other processors only allowed to allocate the memory as read only), the only time a cache line needs to be invalidated is when PE writes read/write memory which other PE's – which have allocated the memory as read only –

have cached. In this case, the write through from the PE with read/write will cause the cache lines in the read only PE's to be invalidated.

3. THE SoCDMMU

As Figure 1 showed, the System-on-Chip consists of multiple PE's connected to a large memory block. Figure 3 shows the memory configuration and connections in greater detail. Each PE can be a microprocessor, a micro-controller, a DSP, or application specific hardware. The large global memory is dynamically G_allocated between the PE's. While we assume the global memory is SRAM, the memory management scheme here is equally applicable to DRAM.

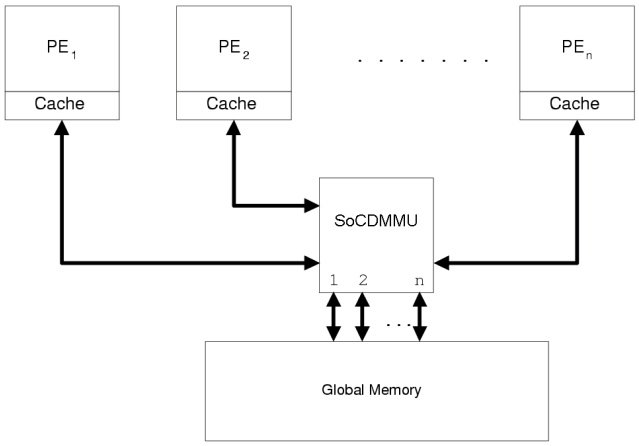


Figure 3. System with SoCDMMU

Notice that we have a separate bus for each PE. This is possible since all PE's are on the same piece of silicon. We hypothesize that this architecture will dramatically improve performance over an architecture with only one bus connecting all PE's to the SocDMMU.

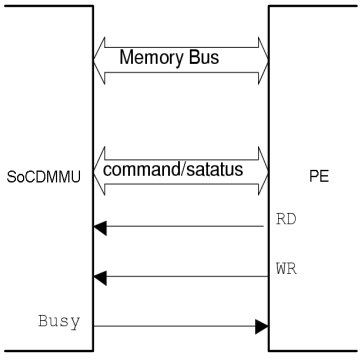


Figure 4. SoCDMMU-PE Interface

3.1 The PE-SoCDMMU Interface

As mentioned in the previous section, the SoCDMMU controls PE access to the global memory and executes the *G_allocate* and *G_deallocate* commands issued by the PE's. Figure 4 shows how a PE is connected to the SoCDMMU.

The PE's memory bus is connected to the SoCDMMU to allow the SoCDMMU to control all of the global memory access. This enables the SoCDMMU to convert the PE-address to physical address. The PE can map any allocated block to any memory location inside its address space (as shown in Figure 5). This feature allows the allocation of non-contiguous memory blocks, so there is no need for memory compaction.

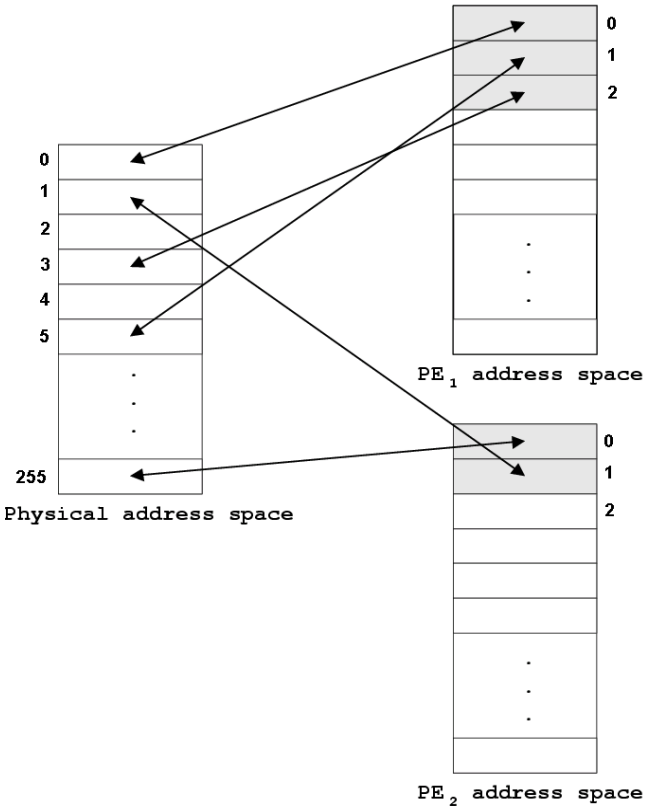


Figure 5. Mapping of physical memory blocks to PEs address spaces.

Example 2 Consider an SoC like that of Figure 1. Assume the total memory is 16MB, which is divided into 256 blocks of 64KB each. Figure 5 and Tables 1 and 2 show the mapping of the physical memory blocks into the address space of PE_1 and PE_2 respectively.□

Physical Memory Block No.	Virtual Block No.
0	0
5	1
3	2

Table 1. Physical to Virtual address mapping for PE_1 in Figure 5.

Table 2. Physical to Virtual address mapping for PE_2 in Figure 5.

Physical Memory Block No.	Virtual Block No.
255	0
1	1

At the same time the SoCDMMU is mapped into a location in the I/O space of the PE. This I/O port (or memory mapped location) to which the SoCDMMU is mapped is used to send commands to the SoCDMMU (write data to the mapped location) and to receive the status of the command execution (reading from the mapped location).

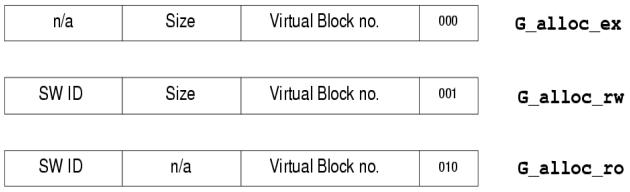


Figure 6. Format of G_allocate commands

There are three types of commands that the SoCDMMU can execute:

- G_allocate Commands:

  Figure 6 shows the command word format for the G_allocate commands. There are three types of G_allocate commands as mentioned earlier: *G_alloc_ex*, *G_alloc_rw*, and *G_alloc_ro*. The first part of the command word specifies the command type. The second part specifies the location at which the PE wants to map the requested blocks on. The other fields depend on the type of the G_allocate command:

  - *G_alloc_ex* needs the size of the memory to be allocated in blocks.
  - *G_alloc*_rw needs the size of memory and the software assigned ID of that page.
  - *G_alloc_ro* needs the software assigned ID of that page.

- G_deallocate Commands:
  Figure 7 shows the format of the G_deallocate command word. The G_deallocate command needs to know the Page ID (virtual block number of the first block in the page) to be de-allocated.



Figure 7. The G_deallocate command word format

- Move Command:
  Figure 8 shows the format of the move command. The move command is used to re-map allocated memory blocks to another location in the PE-address space. This is useful because it allows PE address space compaction. The command needs two parameters: the first parameter specifies the old PE address assigned to that block; the second parameter specifies the new PE address to be assigned to the block.



Figure 8. The Move command word format

The possible errors that can be detected at the SoCDMMU level during the G_allocation and G_deallocation operation are as follows:

- Not enough memory to G_allocate (G_allocation error).
- Trying to G_deallocate non-owned memory page (G_deallocation error).
- Trying to move a non-existing memory block (move error).
- Trying to G_deallocate a non-existing memory page (G_deallocation error).

Example 3 Consider an SoC like that of Figure 1. Assume the total memory is 16MB, which is divided into 256 blocks of 64KB each. Also, assume the virtual address space of each PE is 4GB. Table 3 shows the bit width of each field of the G_allocate commands (*G_alloc_ex*, *G_alloc_rw*, and *G_alloc_ro*). The command field occupies 3 bits. The virtual block number occupies $\log_2$ (4GB/64KB)=16 bits. The size field, which specifies how many blocks are being allocated requires $\log_2$ (256)=8. The Software ID field has 5 bits; this is what remains in a single word (32 – 3 –16 – 8 = 5). If more than 32 software ID's are required this field can be expanded at the cost of a second word for all SoCDMMU commands. Table 4 shows different commands issued by different PE's to the SoCDMMU.□

Table 3. G_allocate command field widths

Field	Width (bits)
Command	3
Virtual Block Number	16
Size	8
Software ID	5

Table 4. Different command words for Example 3

PE	Command	Size	Virtual Block Number	Command Word
1	*G_alloc_ex*	4	8000h	00240000h
2	*G_alloc_rw*	2	9000h	00141001h
3	*G_alloc_ro*	n/a	8000h	00040002h

3.2 The SoCDMMU Architecture

Figure 9 shows the structure of the SoCDMMU. The SoCDMMU can accept simultaneous requests and serializes them using the scheduler. The block labeled "BASIC SoCDMMU" can handle only one request at a time. Multiple requests are handled by having multiple commands and status registers. Each PE has its own command register and status register. Each PE writes its command into its associated command register. When multiple commands are received simultaneously, the scheduler determines which command

will be executed on the BASIC SoCDMMU according to a priority-scheduling algorithm, where priorities are dynamically assigned to ensure that the G_deallocate commands are always executed first.

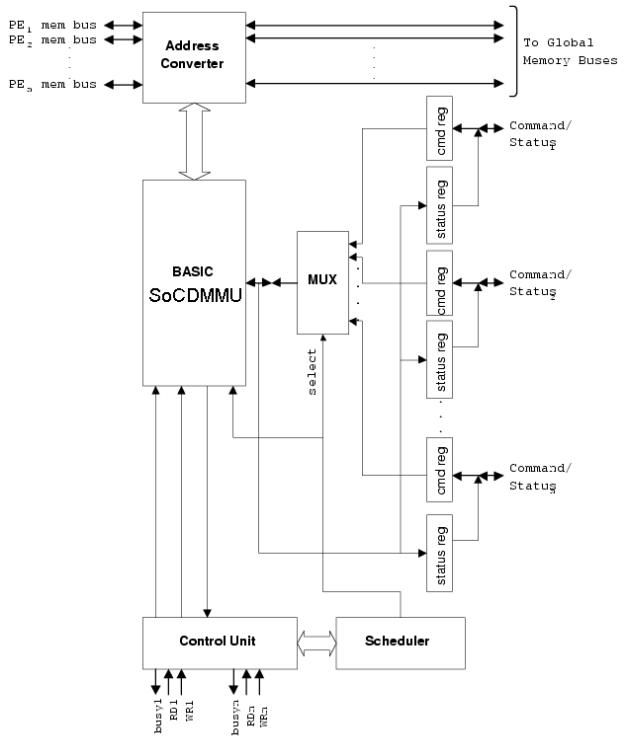


Figure 9. The SoCDMMU Architecture

3.2.1 The Basic SoCDMMU Architecture

Figure 11 shows the structure of the BASIC SoCDMMU. This unit performs G_allocation and G_deallocation commands; it also keeps track of block G_allocation status using the *Allocation Vector*. The *Allocation Vector* is a bit vector where the number of bits equals the total numbers of blocks in the global memory. A value of '1' in a particular bit of the *Allocation Vector* indicates that the corresponding block in global memory is G_allocated; a value of '0' indicates that the global memory block is a available for possible allocation.

Example 4 Consider an SoC like that of Example 2. Figure 10 shows the 256-bit allocation vector where blocks 0,1, 5, and 254 are allocated. □

1	1	0	0	0	1	0	.....	0	1	0

Figure 10. The *Allocation Vector for* Example 4

The BASIC SoCDMMU also stores information about the G_allocated blocks using the *Allocation Table*. The *Allocation Table* is a register file with number of words equal to the total number of blocks. Each word corresponds to a particular block. Figure 12 shows the *Allocation Table* word format. Figure 13 shows an example of the *Allocation Table* with some entries.

The G_allocation process is performed using the *Allocation Unit*. The *Allocation Unit* inputs are the page size in blocks and the information stored in the *Allocation Vector*. Using this information

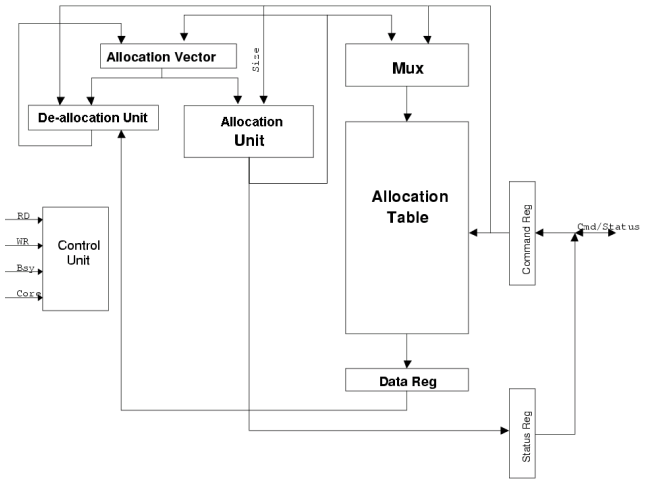


Figure 11. The Basic SoCDMMU Architecture

the *Allocation Unit* allocates the requested page using a first fit algorithm. The output from the Allocation Unit is used to update the Allocation Vector and insert a record in the *Allocation Table*. Also, the *Allocation Unit* updates the information stored in the address converter look up table. The *Allocation Unit* output is written into the status word. If there are any errors then the error code will be written to the status word and no update will be performed.

Size	PE	mode	S/W generated ID

Figure 12. The *Allocation Table* word format

To G_deallocate a page, first the page ID is used to read the page information from the *Allocation Table*. The page information along with the information stored in the *Allocation Vector* are fed to the de-allocation Unit. The de-allocation unit will then de-allocate the page by updating the allocation vector and deleting the page entry from the *Allocation Table*.

3.2.2 The Address Converter

This unit is used to convert the PE address to a physical memory address. The *Address Converter* stores the PE address and physical address for each memory block. This information is updated using the different commands of the SoCDMMU. Note that each PE has its own lookup table to avoid any bottleneck.

0	3	DSP1	ex	n/a
1	0	0	0	0
2	0	0	0	0
3	4	RISC1	rw	1
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0
7	5	RISC2	ex	n/a
	. . .			

Figure 13. Example of an *Allocation Table* with some entries

The PE address to physical address conversion is required to make the allocated non-contiguous memory blocks appear as contiguous

memory blocks to the PE address space. This prevents memory fragmentation; hence, no memory compaction is needed for the global on-chip memory blocks.

The conversion process is done as in Figure 14. First the converter use the virtual block number part of the block PE address to look up the physical block number. The physical block number is used along with the block-offset part of the PE address to construct the physical address.

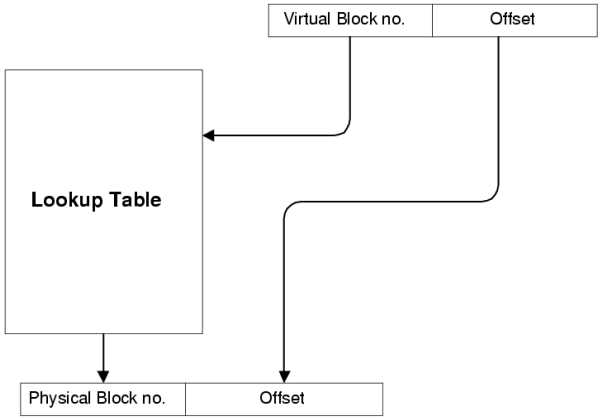


Figure 14. PE Address to Physical Address Conversion

4. EXPERIMENTS AND RESULTS

To test the effectiveness of our approach, we simulated a model for an SoC that utilizes an SoCDMMU using the Synopsys VCS™ verilog simulator. The simulated system looks like the system illustrated earlier in Figure 1. The system has four PE's (2 RISC processors – e.g., MIPS – and 2 DSPs – e.g., Motorola DSP56k –), 16MB SRAM and the SoCDMMU.

The memory is divided into 256 blocks; each block is 64KB. The SoCDMMU utilizes a 256-bit *Allocation Vector* and an *Allocation Table* with 256 entries. We conducted a number of experiments to test the quality of our approach. Figure 15 shows a screenshot of the simulation of the PE-SoCDMMU interface, where four PE's are connected to the SoCDMMU. The last four signals in the timing diagram show the commands that are explained earlier in Example 4 being issued by the different PE's.

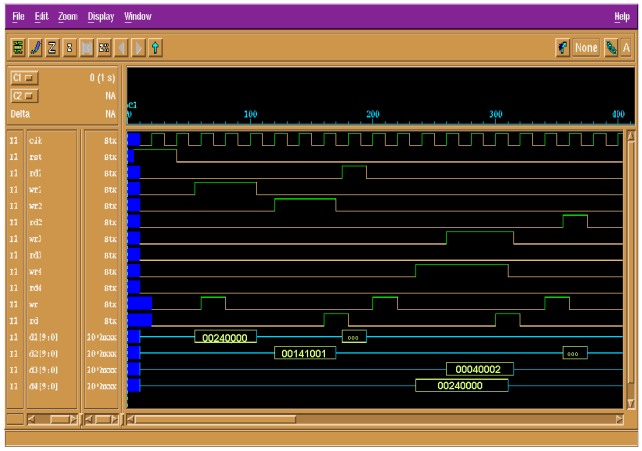


Figure 15. Screenshot of the Simulated System

4.1 SoCDMMU Command Execution Times

In this experiment we measured the worst-case execution time of the SoCDMMU commands. Table 5 summarizes the results. We found that the worst-case execution time occurs when all PE's issue G_deallocate commands.

In Table 6, we see the results for the synthesis of the SoCDMMU using the Synopsys Design Compiler™. The second column in Table 4 shows the number of gate equivalents of hardware required using the AMI 0.5-micron Logic library.

Table 5. Execution Times in Cycles

Command	Number of Cycles
G_alloc_ex	4
G_alloc_rw	4
G_alloc_ro	3
G_dealloc	5
Worst-Case Execution Time	20

Table 6. The SoCDMMU Synthesis Results

Lines (RTL Verilog)	Area
1,030	41,561.5

Example 5 Consider the handheld device in Example 1. Originally, the audio player application uses all available memory. When used for communication, OFDM processing requires memory allocation to be dynamically changed. Specifically, OFDM uses DSP1 and DSP2. DSP1 reads the incoming data from the FIFO buffer and performs a 1024-point FFT for each received symbol to find the original transmitted spectrum, and then DSP1 stores the results into a memory buffer that is shared with DSP2. The phase angle of each transmission carrier is then evaluated and converted back to data words by demodulating the received phase. The demodulation is performed by DSP2. The operation is outlined in Figure 16. DSP1 allocates the shared memory buffer as read/write and DSP1 allocates it as read only.

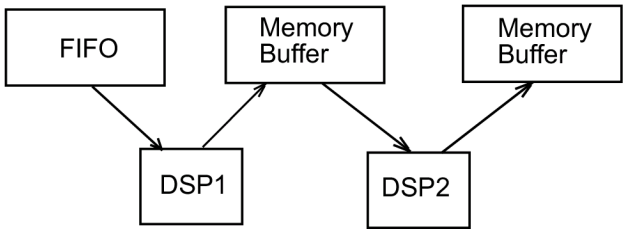


Figure 16. OFDM sub-System

Assume that the incoming data rate is 10 GBit/Sec and the input FIFO buffer is 1024 words. The FFT performed by DSP1 must start before the input FIFO buffer reaches 10% of its capacity (or else under some conditions the FIFO could overflow). Then, the time for the device to start processing the incoming packet must be less than 10% x 1024 x 32 / (10 GBit/S) i.e., less than 305ns. For the first time executed, part of this time is the time consumed to allocate the required memory buffers. This means that the memory allocation must be fast and its worst-case execution time must be predictable and low. If the system involves SoCDMMU to manage the global

on-chip memory then the allocation of the two memory buffers will be 2 x 20 = 40 cycles in the worst case. For a system clock of 400MHz (which can be consider low clock speed compared to the current clock speeds) the time is 100 ns, which can satisfy the real-time requirement. Note that a conventional memory allocation scheme would probably be at least a factor of ten slower in the worst case, and hence would not meet the real-time requirement.

4.2 Comparison with a Micro-controller Implementation

To demonstrate the importance of building the SoCDMMU in custom hardware, we compared the SoCDMMU performance with the performance of software running on RISC micro-controller (Microchip PIC™ micro-controller). This software performs the same function as the SoCDMMU. Table 7 compares the worst-case execution time of the hardware SoCDMMU with the best-case execution time for the micro-controller implementation of the SoCDMMU in software. The comparison is shown in clock cycles. We assume that the hardware SoCDMMU and the micro-controller both have the same clock rate (e.g., 400MHz).

Table 7. A comparison between the SoCDMMU and the Micro-controller Execution Times

SoCDMMU Worst-Case Execution Time	20 Cycles
Micro-controller Best-Case Execution Time	221 Cycles

5. CONCLUSION

In this paper, we described an approach to handle on-chip memory allocation between PE's in an SoC. Our approach is based on a hardware SoCDMMU that allows a dynamic, fast way to allocate/de-allocate the on-chip memory. Moreover, the SoCDMMU allocation/de-allocation of the memory blocks is completely deterministic, which makes it suitable for real-time applications. Thus, this approach fits in the gap between general-purpose fully shared memory multiprocessor SoCs and application specific SoC designs with custom memory configurations.

Currently, different types of RTOS's are being modified to extend their memory management schemes to support the hardware SoCDMMU. We also plan to extend the SoCDMMU to support *G_alloc_rw* of the same block by multiple PE's, thus providing fully shared memory blocks. Finally, for future work we plan to carry out a study comparing our multiprocessor SoC to a SoCDMMU with fully shared memory multiprocessor SoC like Hydra [5].

6. Yes! Open Hardware Description Language

The verilog code for the SoCDMMU is available at *www.yohdl.org under* the YOHDL open source license for downloading

7. ACKNOWLEDGMENTS

The authors would like to thank Kyeong Keol Ryu for his help in understanding the OFDM system under development at GaTech Broadband Institute [3].

This research is funded by a fellowship from the Egyptian government and by NSF under grant numbers INT-9973120 and CCR-9984808.

8. REFERENCES

[1] C.E. Kozyrakis et al, "Scalable Processors in the Billion-Transistor Era: IRAM," *IEEE Computer*, 75-78 (September, 1997).

[2] D. Verkest et al., "CoWare-A Design Environment for Heterogeneous Hardware/Software Systems," *Design Automation for Embedded Systems*, 2(4), 357-386 (October 1996).

[3] Georgia Institute of Technology, Broadband Institute, *www.broadband.gatech.edu*

[4] J. M. Chang, W. Srisa-an, and C.D. Lo, "Introduction to DMMX (Dynamic Memory Management Extension)", *ICCD Workshop on Hardware Support for Objects and Microarchitectures for Java*, Austin, TX. (October 10, 1999).

[5] K. Olukotum et al., "The Case for a Single-Chip Multiprocessor," *Proceedings of the Seventh International Symp. Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, Cambridge, MA, (October, 1996).

[6] P.R. Panda,N.D. Dutt, and A. Nicolau "Mmeory Data Organization for Improved Cache Performance In Embedded Processor Applications," *ACM Transaction on Design Automation of Electronic Systems*, 2(4) (October, 1997).

[7] P.R. Wilson et al., "Dynamic Storage Allocation: A Survey and Critical Review," *Proceedings 1995 Int'l workshop on Memory Management*, Scotland (September, 1995).

[8] *Proceedings of IEEE*, Special Issue on Distributed Shared Memory Systems, 87(3), 397-532, (March, 1999).

[9] S. Morgan, "Jini to the rescue," *IEEE Spectrum*, 37(4), 44-49 (April, 2000).

[10] S. Wuytack et al., "Memory Management for Embedded Network Applications," *IEEE Transaction On Computer-Aided Design of Integrated Circuits and Systems*, 18(5), (May, 1999).

[11] The International Technology Roadmap for Semiconductors, *edited by SIA Semiconductor Industry Association*, 1999.

[12] Y. Li, and W.H. Wolf, "Hardware/Software Co-Synthesis with Memory Hierarchies," *IEEE Transaction Computer-Aided Design of Integrated Circuits and Systems*, 18(10), 1405-1417, (October 1999).