Coded data storage for dense memory architecture

Laboratory of Informatics, Networks and Communications (LINC)   
The University of Texas at Austin

Table of Contents

[1. Introduction 3](#_Toc448820044)

[2. Problem Definition and Evaluation Criteria 3](#_Toc448820045)

[3. Introduction to HBM & HMC 4](#_Toc448820046)

[4. Code Designs for Dense Memories 6](#_Toc448820047)

[5. HBM systemC Model 9](#_Toc448820048)

[6. Results 11](#_Toc448820049)

[7. Conclusion 12](#_Toc448820050)

[8. References 12](#_Toc448820051)

List of Acronyms

BIST – Built in Self Test

CRC – Cyclic Redundancy Check

DDR – Double Data Rate

DRAM – Dynamic Random Access Memory

ECC – Error Correcting Code

FIFO – First In, First Out

GPU – Graphics Processing Unit

HBM – High Bandwidth Memory

HDFS – Hadoop Distributed File System

HMC – Hybrid Memory Cube

MDS – Maximum Distance Separable

RS – Reed Solomon

RAID – Redundant Array of Independent Disks

SoC – System on Chip

TSV – Through Silicon Via

# Introduction

In the era of disruptive development in the field of data science and machine learning, computing systems with high processing capacity are in huge demand. According to Moore’s law, computing systems double their overall computation capability every 18 months. The processing power of a system is mainly determined by its compute capability and memory operations. Recent trends in computer architecture systems have shown that memory access speed is a major bottleneck in increasing the processing capability. This is due to the large amount of time required to transfer data between the processor and the memory. The research and industrial communities have improved data access capacity and speed through various innovations in integrated circuit design. However there still remains work to be done for further improvement in memory access efficiency. In this project, we propose a new, innovative solution to improve the access efficiency of next-generation memory systems such as high bandwidth memory (HBM) and hybrid memory cube (HMC). We propose a novel architecture that uses redundancy in data storage to provide parallel memory access with low latency.

# Problem Definition and Evaluation Criteria

In this section, we define the problem for memory data storage and develop key parameters that will be used to evaluate our proposed solutions.

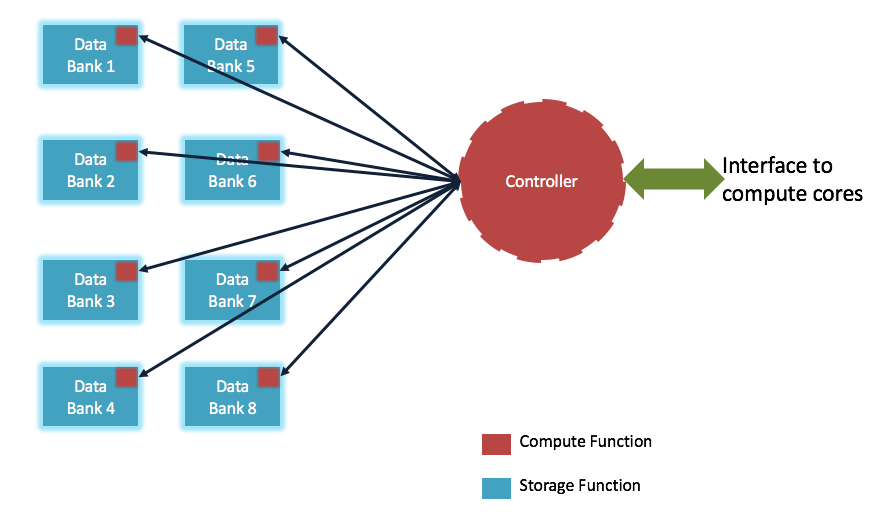
In our previous efforts from 2014-2015, we developed coded data storage architectures which applied mainly to integrated memories in system-on-chip (SoC) subsystems. The proprietary memory design on SoC provides a lot of flexibility to design and implement a complicated coding scheme with less overhead. However, such architectures are limited to a certain subset of products and do not scale to ubiquitous DRAM memories deployed in most computing systems. The more prevalent DRAM memories are manufactured by multiple vendors and are governed with set industry standards and protocols. Interoperability with commercial devices (and conformity to standards) requires a new design for coded data storage, one which can be implemented by enhancing the memory controller and providing the promised benefits of the coded architecture.

Figure New memory architecture for dense memories where each data bank has a logic unit to perform simple arithmetic operations.

Another key motivation for this project is to explore the new key features envisioned in two dense memory architectures, HBM and HMC. One feature is the ability to integrate a small logic function with each bank, which interfaces with the memory controller and performs certain preset arithmetic functions. As shown in Figure 1, each data bank is now capable of computing codes on the fly by accessing elements from its memory and constructing arithmetic combinations with its locally available logical block. This capability helps the code designers to structure codes which can be constructed dynamically to improve the system’s overall efficiency. Local arithmetic also allows designers to draw from similar results and techniques from the field of distributed memory systems for large data servers.

We believe that there are three principal criteria to evaluate memory systems: latency, throughput, and complexity of the memory controller. We plan to characterize the first two parameters via their probability density functions and cumulative distribution functions. Such an analysis will help us understand the performance of a given solution and provide intuition during the design process. Memory controller complexity is driven largely by the complexity of the coding scheme and the number of writes required to update data. Secondary criteria include cost analysis, computation overhead, storage overhead, and the support of specific read/write request patterns. Specifically, read/write support can be evaluated via the maximum number of parallel reads (both overall and restricted to a single bank), as well as the number of disjoint banks accessible during a coded read.

# Introduction to HBM & HMC

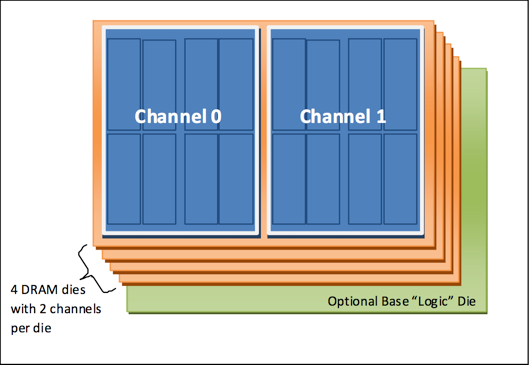
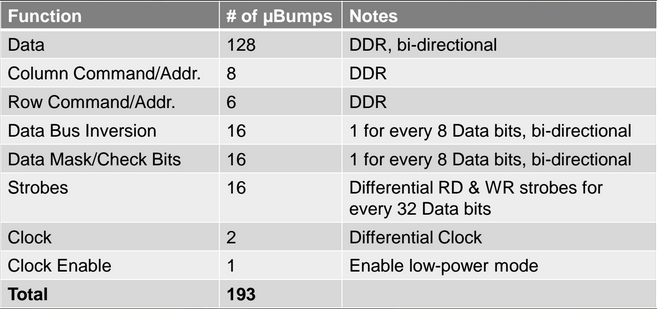
    HBM stands for High Bandwidth Memory which enables systems with extremely high bandwidth requirements like future high-performance GPUs. The HBM standard defines a stack of memories as shown in the Figure 2 where each stack contains 2 channels with 4 DRAM dies. This memory design enables a 3D scaling of memory, which packs more data in the same space as 2D scaling and reduces access latency. The architecture also helps the computing cores to parallelize their access channels, meeting the high data access requirements while consuming less power.

Figure 2 High density memory architecture of HBM. The 3D stacking of memory channels provides an orthogonal plane to pack a large amount of storage in a smaller footprint while maintaining the electromagnetic characteristics of the system.

The HBM standard defines a stack with features such as footprint, signaling interface, commands, protocols and ECC support. However, it does not specify the stack’s internal architecture or the precise DRAM timing parameters. Such a flexible architecture enables system designers to come up with various designs for memories and storage patterns which enable higher access rates.

Each HBM channel is expected to provide a 128-bit data interface that supports up to 1-2 Gbps of data rate per signal equivalent to 500-1000MHz of DDR memory. This is equivalent to 16-32 GB/sec of bandwidth per channel. 8 channels stacked together as in Figure 2, which provides 128-256 GB/sec of bandwidth per stack. The following table summarizes key parameters of HBM memory:



Key features of HBM organization:

* Completely independent channels
  + Independent clocks & timing
  + Independent commands
  + Independent memory arrays
  + In short, nothing one channel does affects another channel
* Up to 8 channels per stack and 8 or 16 banks per channel; varies by device density/channel
* 16-32 GB/sec of bandwidth per channel, 128-256 GB/sec of bandwidth per stack
* Self Refresh Modes
  + Bank refresh command to single bank
* Bank Grouping supported
* Differential clock inputs (CK\_t/CK\_c)
* Optional ECC pin support
* Semi-independent Row & Column Command Interfaces allowing Activates/Precharges to be issued in parallel with Read/Writes
* Channel density of 1 Gb to 32 Gb
* 2K or 4K Bytes per page; varies by device density/channel

HMC memory stands for hybrid memory cube. A hybrid memory cube (HMC) is a single package containing either four or eight DRAM die and one logic die (Figure 3), all stacked together using through-silicon via (TSV) technology. Within each cube, memory is organized vertically; portions of each memory die are combined with the corresponding portions of the other memory die in the stack. Each grouping of memory partitions is combined with a corresponding controller within the logic die, forming what is referred to as a vault.

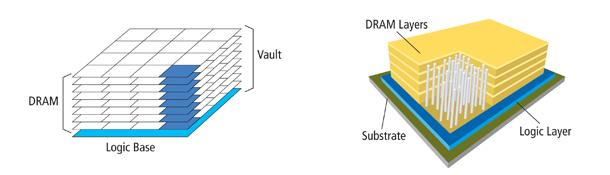


Figure 3 Example memory organization in HMC.

Key features of HMC memory organization:

* Closed-bank memory architecture
* Built-in memory controller for each vault
  + Automatic refresh control over all temperatures
* Internal ECC data correction
* Advanced RAS features including data scrubbing
* Post-assembly repair capability
* In-field repair for ultimate reliability
* 12.5 Gb/s, 15 Gb/s, 25 Gb/s, 28 Gb/s, or 30 Gb/s SerDes I/O interface
* Up to four 16-lane, full-duplex serialized links
  + Half-width link (8-lane) and quarter-width link (4-lane) configurations also supported
  + Up to 320 GB/s effective bandwidth
* Packet-based data/command interface
* Supports 16, 32, 48, 64, 80, 96, 112, 128, and 256 byte references per request
* Error detection (cyclic redundancy check [CRC]) for packets with automatic retry
* Power management supported per link
* Through-silicon via (TSV) technology
* Built-in self-test (BIST)
* JTAG interface (IEEE 1149.1-2001, 1149.6)
* I2C interface up to 1 MHz
* SPI master interface

# Code Designs for Dense Memories

We propose an implementation on 16 data storage units, which could be either data banks or channels. The main component is a (14,10) Reed Solomon code over a Galois Field size of 28, denoted GF(28). This code’s systematic implementation can be used to convert 10 messages in to a codeword of length 14 with the ability to recover the original 10 messages using any 10 symbols of the code word. This (14,10) code is a maximum-distance separable (MDS) code that reduces the overall memory transaction to reconstruct the code. In the coding theory literature, the MDS codes are defined as codes that meet the Singleton bound:

Here, d is the distance, i.e. the minimum number of positions in which any 2 code words differ, n is the block length, and k is the dimension. For our RS code =5, =14, and =10. One can correct up to errors in an MDS code, which corresponds to at most 2 errors in our implementation. MDS codes provide maximum distance and thus are preferable in any such designs.

    This code was chosen for several reasons. First, codes with the MDS property are preferred to ones with distance strictly less than the Singleton bound. When available, Reed Solomon codes provide an explicit MDS construction for the pair of parameters (). Often this comes at the expense of a large field size, but later in this section we describe a method to map the arithmetic to binary operations on the individual bits of each byte vector [6]. Reduced logical complexity means that the vector RS code is amenable to pipelining and may be used in both HBM and HMC systems. Throughout the design process, adjusting the RS parameters will allow us to achieve other points on the overhead-performance tradeoff. In contrast to other coding schemes for memory systems like SEC-DED [3], BAMBOO [2], and ChipKill [4], the proposed coding scheme allows for more intelligent memory controller design by improving data access as well as error correction capability.

Although some randomized codes are known to satisfy the MDS bound, our hardware implementation benefits from having deterministic encoding and decoding functions. Then the operations can be “hardcoded” into the memory controller to improve its performance at runtime. Efficient encoding and decoding implementations have been studied for decades. Moreover, RS codes have proven to be useful in previous work on distributed storage, both in theory and in practice [1,5-6]. In fact, the (14,10)-RS code is in production at Facebook in their HDFS-RAID system [5]. This makes it a natural starting point for our dense memory storage architecture.

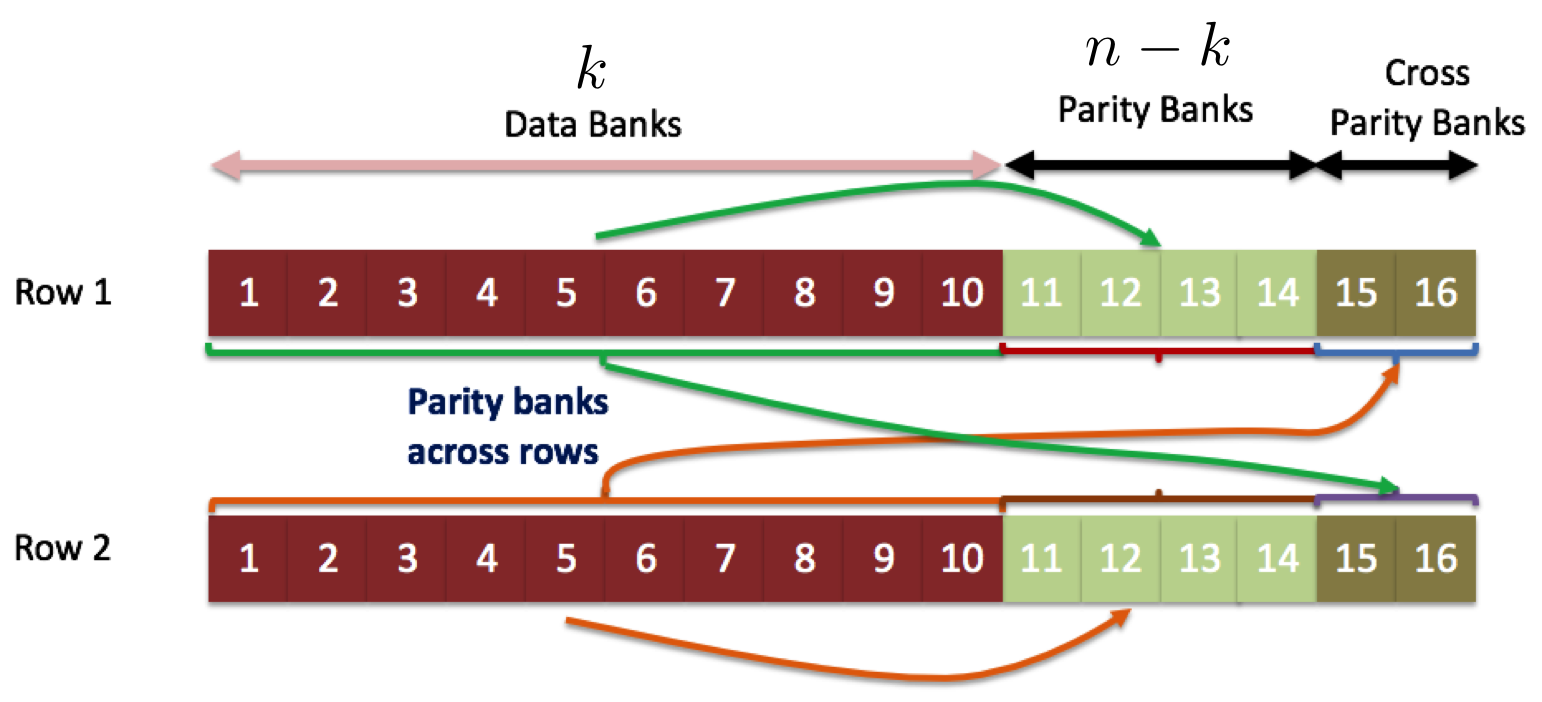
Given the base RS encoding at each bank, our next proposed step is to augment each bank with two cross bank parity symbols. These cross parities allow for recovery of original data by accessing the same number of banks spread across multiple rows. In Figure 4, for example, the data in Row 1 may be recovered by accessing 10 banks via Row 1 or by accessing Banks 1-6 and 11-12 in Row 1 and 15-16 in Row 2. Moreover, due to the MDS property of RS codes, we can correct up to 2 errors on these 10 bank accesses. This will be analyzed and evaluated in more detail as future work.

Figure 4 The proposed new architecture allows for storage of cross parity data in a row. The cross parities can be used to improve access capability by decoding the original message using a partially recovered code word.

The following subsections describe details of the base RS base code and how it is implemented.

**Reed Solomon (RS) Codes:** Reed Solomon codes are a family of error correcting codes that enjoy the advantages of linearity and ease of encoding while providing MDS property in some constructions. RS codes are an example of algebraic codes in which the weights of a polynomial are used to construct a code word. Here follows a formal definition:    
Let be a prime number and let . The Reed-Solomon code over the field with message symbols and n code symbols is defined as follows. Given a message vector , let be the polynomial

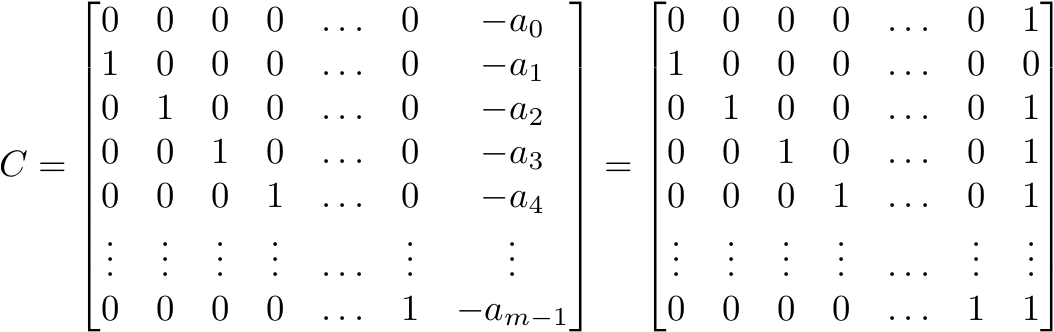
with coefficients given by the message symbols. Thus is a polynomial of degree atmost in one variable , with coefficients in . Then the code vector y for this message vector is the list of n evaluations of the polynomial ):

,

which is evaluated using modular arithmetic in .

For RS (14,10) the evaluations are powers of the primitive element of a generator polynomial that generates GF() from GF(2),  given by .

**Vector-Matrix Implementation:** The companion matrix of the primitive polynomial is a matrix given by



The companion matrix may be used to relate scalar multiplication in the extension field GF() to matrix-vector multiplication in the base field GF(2). Concretely, any can be

interpreted as a vector that belongs to a vector space of dimension  over GF() with the following vector representation:

These can be interpreted as coefficients of a polynomial in the primitive element, or as an isomorphic mapping to these coefficients. Similarly, any nonzero field element in GF()  can be written as , 0 ≤ n ≤ -2 . The mapping g() = is an isomorphism between GF ()  and the set of matrices {}  over GF()  that preserves the field multiplication and addition in terms of matrix multiplication and addition over the space of matrices (GF(p)). This means that we can map c = a\*b in GF() to **c** = A\***b** in GF() by applying the transformations , , and

Figure 5 represents the equivalence of the two representations for encoding a systematic Reed Solomon code. On the right hand side, arithmetic is performed as scalar multiplication in GF(28), while the equivalent operation on the left hand side is a matrix-vector multiply in GF(2).

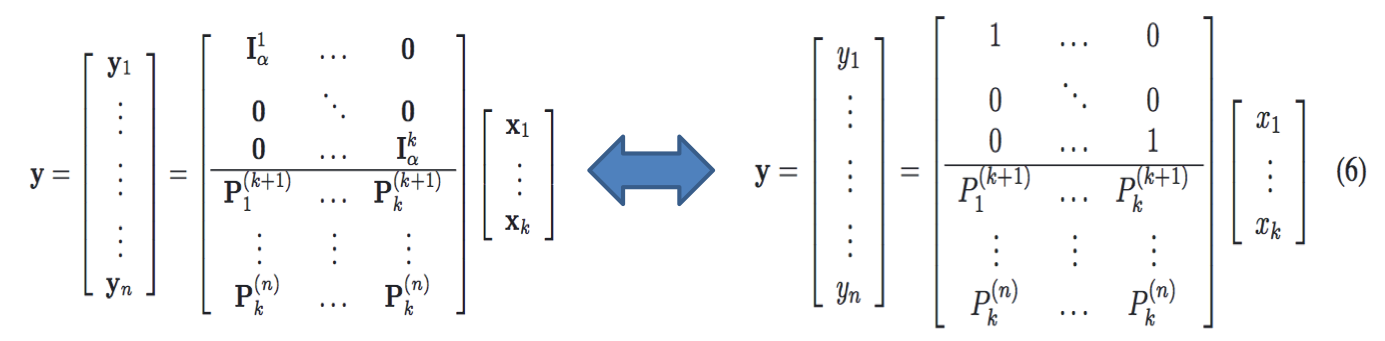


Figure 5 Correspondence between matrix-vector multiplication and scalar multiplication.

# HBM systemC Model

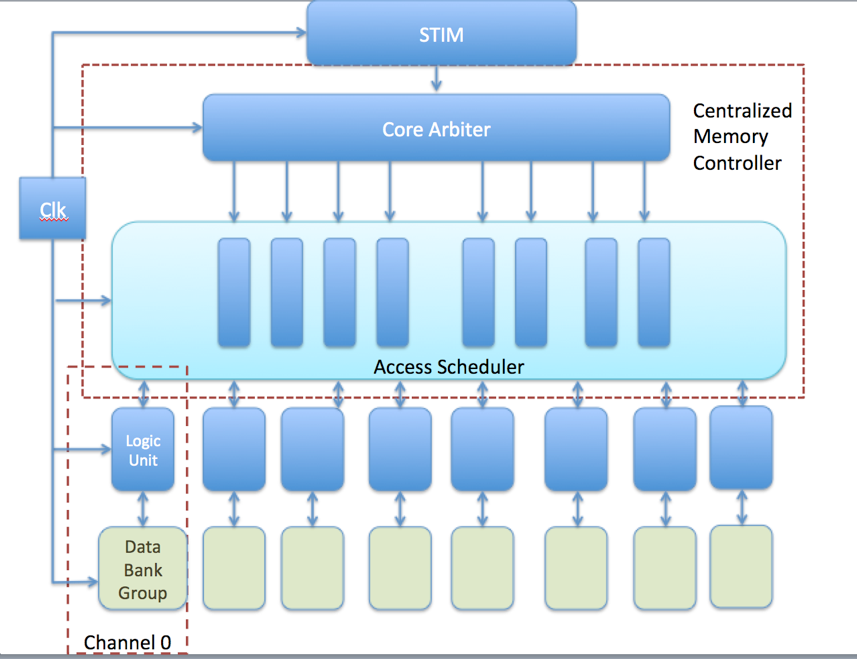
To evaluate the performance of a coded architecture for HBM, we have developed a systemC based simulator which models the HBM memory controller. This section describes the implementation and architectural details of our simulator. Since the data access protocol for a channel is similar to the existing double data rate (DDR) protocol, we first implement the architecture for one channel and extend it to 8 channels.

Figure 5

Figure 6 Architectural block diagram of memory controller for HBM.

The memory controller simulation model consists of three main processing blocks (Figure 6): The first block is a STIM module, which simulates the functionality of master cores. The second module is a core arbiter, which handles and prioritizes requests from cores. The third block is an access scheduler, which forms access patterns to the memory array.

**STIM Module:**This module simulates the behavior of memory access request generation from compute cores. In the current implementation, this block reads and plays the trace files that contain the access parameters with their timestamps. It simulates six master cores that issue AXI burst requests to the memory.

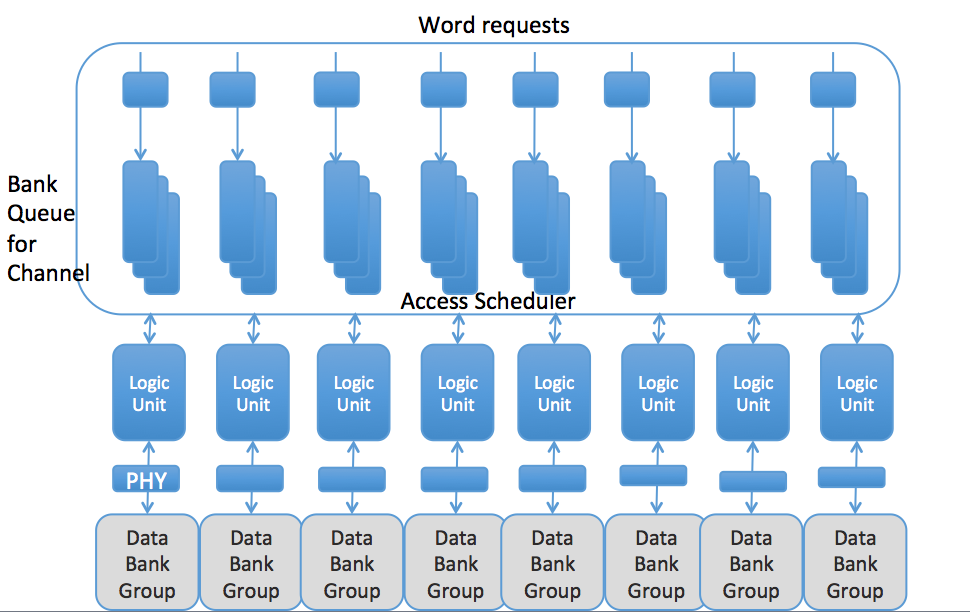
**Core Arbiter:**  The core arbiter is responsible for handling access requests from memory cores. It handles requests from all cores at every clock cycle and maintains a depth 8 first in first out (FIFO) queue for each core. When the queue fills up for any core, it generates a busy signal to the respective core triggering a stall until the queue becomes available again. The current implementation of the core arbiter checks and assigns resources to the queues in a round robin fashion. It also parses the burst request commands into word requests and forwards each request to the appropriate read/write channel/bank queue.

Figure 6 Block diagram of bank access scheduler for HBM

Figure 7 Block diagram of bank access scheduler for HBM.

**Access Scheduler:** This module forms access patterns and interacts with the DRAM memory banks. Each channel has its own independent bank queues, logic units, and data storage arrays. At every cycle, the scheduler looks at the size of each channel’s bank queues and tries to form an access pattern. A typical access pattern is formed by first determining the request type for each channel by looking at the relative sizes of the read and write queues. Then, the scheduler reads the queue and forms an optimized access pattern that maximizes the amount of data stored/retrieved. Figure 7 presents a detailed architectural block diagram of access scheduler and its interface with the memory banks. Meanwhile, the scheduler will send bank refresh command to arbitrary single bank in a relatively fixed frequency, which can save potential bandwidth compared with general DRAM controller.

The DRAM access interface is implemented based on state diagram in Figure 8(A). It describes a fixed procedure for accessing a word in a bank. First, the current row is closed, *i.e.* the data from the row buffer is charged back to the memory row. Then the row to be accessed is fetched into the row buffer through column access. Finally, the data is retrieved or written into the row buffer to complete the access.

Figure 8 DRAM memory read and write command sequence.

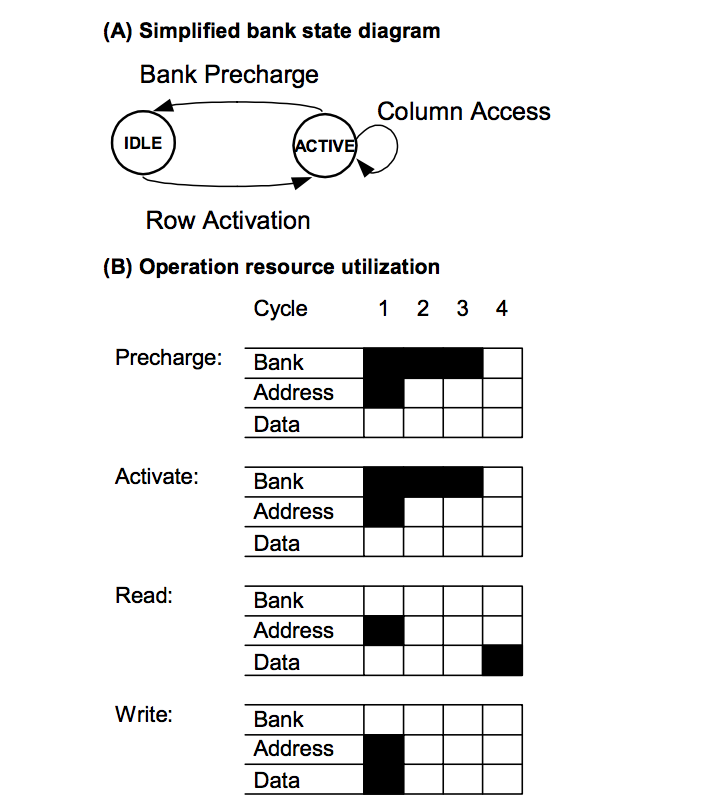
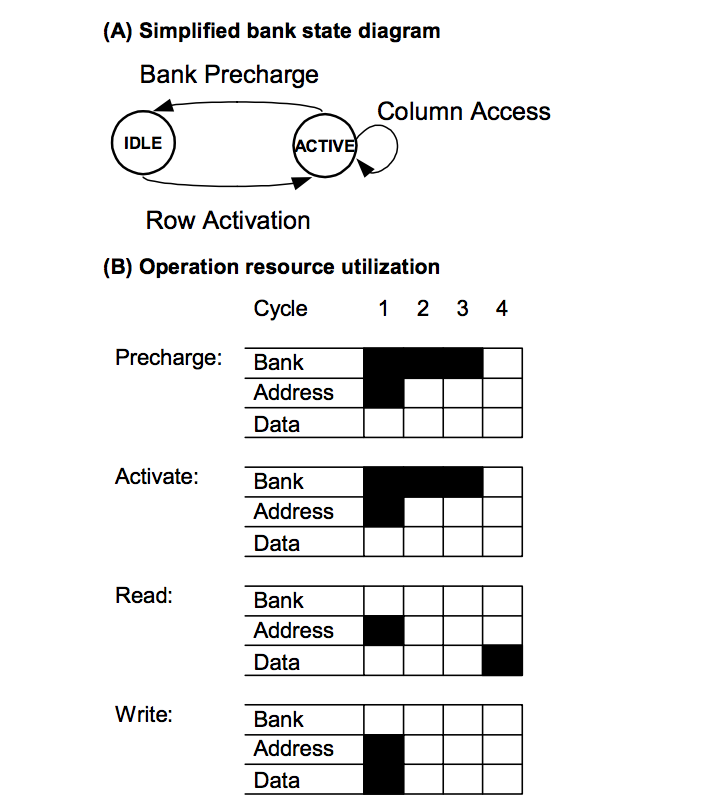


Figure 8(B) describes the number of cycles taken by each DRAM command. Clearly, pre-charge and activate take up many more cycles than the other commands. Row activation requires 3 cycles during which no other operations may be issued to that bank. This is indicated by the utilization of the bank resource for the duration of the operation. During that time, however, operations may be issued to other banks of the DRAM. Once the DRAM’s row activation latency has passed, the bank enters the ACTIVE state, during which the contents of the selected row are held in the bank’s row buffer. Any number of pipelined column accesses may be performed while the bank is in the ACTIVE state. To issue either a read or write column access, the address lines are required to indicate the bank and the column of the active row in that bank. A write column access requires the data to be transferred to the DRAM at the time of issue, whereas a read column access returns the requested data three cycles later. Additional timing constraints not shown in Figure 8, such as a required cycle of high impedance between reads and writes, may further restrict the use of the data pins.

# Results

The assumptions we have:

1. Computations in logic block, like address calculation and data encoding, do not take significant time compared to the data transmission and DRAM operations.

2. DRAM operations are simplified as a 2-state FSM shown in Figure 8(A) along with the resource utilization.

3. In the current implementation, single bank fresh is implemented where all the banks refresh at the same time as shown in Figure 9.

4. Read activity has priority than write so when scheduling access to memory array, we prefer to schedule read requests.

The following table shows the read and write latency performance in the baseline implementation of HBM memory controller in systemC.

Data rate = word length (256 bit) /average read or write latency

The parameters for the baseline model is:

Number of banks 16

Number of channels 8

The latency is caused by the time requests stay in the queue and the DRAM operations, we assume pre-charge takes 3 cycle, row activation takes 3 cycles, read and write takes 1 cycle.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Average Read latency (ns)** | **Average Write latency (ns)** | **Worst read latency (ns)** | **Worst write latency (ns)** |
| LTE | 25.99 | 31.8 | 328 | 494 |
| Trace 1 | 22.7 | 25.13 | 153 | 217 |
| Trace 2 | 23.21 | 25.35 | 129 | 210 |

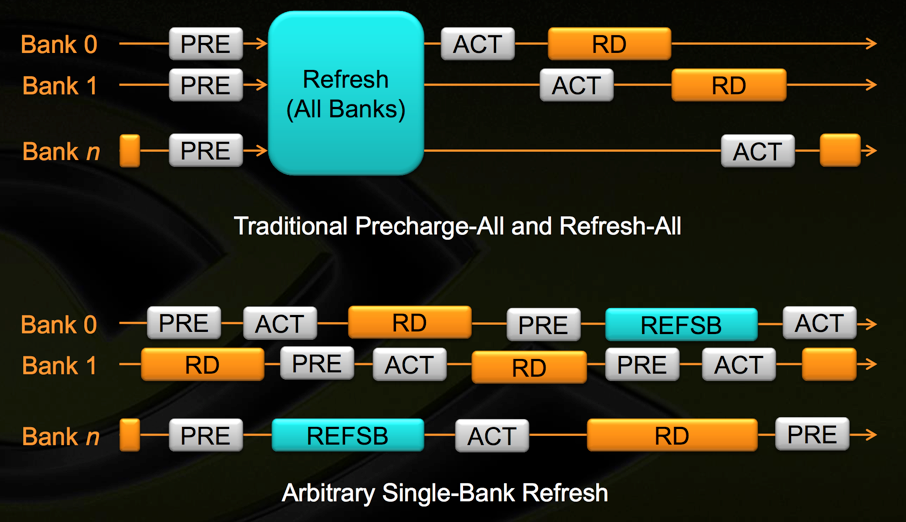


Figure 9 Single bank refresh vs. arbitrary bank refresh. HBM will employ arbitrary bank refresh where banks independently refresh allowing for more flexibility for command execution.

# Conclusion

Coding architecture for dense memories presents unique opportunities and challenges not seen in on-chip memory architectures. We propose to use ideas from distributed data storage literature as a starting point to design novel coded memory architectures with the goal of improving latency and error correction. In the next phase, we plan to thoroughly evaluate code designs for their benefits on the performance metrics developed in this phase. This design and evaluation cycle will be the main focus of the project going forward.

# References

[1] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, “Erasure Coding in Windows Azure Storage,” *Atc*, p. 12, 2012.

[2] J. Kim, M. Sullivan and M. Erez, "Bamboo ECC: Strong, safe, and flexible codes for reliable computer memory," *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Burlingame, CA, USA, 2015, pp. 101-112.

[3] Intel Corp., “Intel Xeon Processor E7 Family: Reliability, Availability, and Serviceability,” 2011.

[4] International Business Machines Corp. (IBM), “Chipkill Memory,” http://www-05.ibm.com/hu/termekismertetok/xseries/dn/chipkill.pdf, Tech. Rep., 2012

[5] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, “XORing Elephants: Novel Erasure Codes for Big Data,” *39th Int. Conf. Very Large Data Bases, VLDB*, vol. 6, no. 5, pp. 325–336, 2013.

[6] K. Shanmugam, D. S. Papailiopoulos, A. G. Dimakis, and G. Caire, “A repair framework for scalar MDS codes,” *IEEE J. Sel. Areas Commun.*, vol. 32, no. 5, pp. 998–1007, 2014.