AFRL-RI-RS-TR-2019-014

AGILE 3D MEMORY INTERFACES

NORTH CAROLINA STATE UNIVERSITY

JANUARY 2019

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2019-014 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
CHRISTOPHER FLYNN
Work Unit Manager

/ S /
STEVEN JOHNS
Chief, Trusted Systems Branch
Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JANUARY 2019	FINAL TECHNICAL REPORT	AUG 2015 – AUG 2018

4. TITLE AND SUBTITLE

AGILE 3D MEMORY INTERFACES

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA8750-15-1-0280

5c. PROGRAM ELEMENT NUMBER
62788F

6. AUTHOR(S)

Paul Franzon, Lee Baker, Theodoros Nigussie

5d. PROJECT NUMBER
95SB

5e. TASK NUMBER
TM

5f. WORK UNIT NUMBER
EM

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

North Carolina State University
P.O. Box 7911
Raleigh, NC 27695

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2019-014

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The DiRAM4 memory permits fast random access of DRAM via 64 bidirectional memory ports. A memory controller was designed and verified that is suited for interfacing a multicore CPU to this unique DRAM.

15. SUBJECT TERMS
Dynamic Memory, Memory Controller, Secure Processor

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**CHRISTOPHER FLYNN**
U	U	U	UU	27	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

Table of Contents

Section **Page**

1. Summary 1
2. Introduction 1
3. Methods Assumptions and Procedures 1
4. Results 1
5. Deliverables 4
6. Publications 4
7. Technology Transfer 4
8. Conclusions 4
9. Appendix Detailed RTL Description 5
10. List of Acronyms 22

1. Summary

The DiRAM4 memory permits fast random access of DRAM via 64 bidirectional memory ports. A memory controller was designed and verified that is suited for interfacing a multicore CPU to this unique DRAM.

2. Introduction

The original objectives were as follows:

1. Simultaneously, do the following:
    a. Design of a test package and testing of a DiRAM4 DRAM. This enables us to understand the interface.
    b. Design of baseline DMA/Cache refill controller and PHY interface, capable of interfacing to this DRAM. This will be the first general purpose memory controller and first DiRAM cache controller.

3. Methods, Assumptions and Procedures

A DiRAM4 specficiation was obtained from TEzzaron, including a Verilog model. A memory controller was designed and verified against that model. It was designed using the synthesizable subset for Verilog 2001, and verified using SystemVerilog.

4. Results

We achieved the following:

a. We designed and verified a DiRAM memory controller.
b. We designed a test package for the DiRAM memory controller and explored interposer design tradeoffs.

We did not complete testing of a DiRAM4 DRAM as a working DRAM was never provided to us by Tezzaron.

4.1 DiRAM Memory Controller

A block diagram of the memory controller is shown in Figure 1. It has five major blocks with features as follows. All features can be configured and optionally disabled.

1) Decoder
    - Directs system transactions to addressed bank
    - Converts the single system transaction into two internal transactions
    - Manages tag generation and read data tag ordering
    - Merges tags of potentially out-of-order data

2) Bank Controller
    - Keeps track of currently open page
    - Can reorder transaction based on open page (during congestion)
    - If reordering is enabled, provides age timers to ensure transactions get processed
    - Converts a system transaction into an command packet which is sent to the scheduler
        - E.g. A single command encodes POPCCW, POCR, CR etc.
    - Optionally closes page if current transaction is last cache line in page
3) Scheduler
    - Takes command packets from each bank controller and generates individual DiRAM4 commands which are sent to the DFI interface
    - Ensures DiRAM4 cycle-to-cycle requirements are met
    - Takes command packets from the refresh controller and merges individual DiRAM4 commands into the command path
4) Refresh Controller
    - Periodically generates per bank command packets to the scheduler
    - Command packets will be a combination of PR, PCPR, PRPO etc.
    - Which banks and number of Pages refreshed will be configurable
5) DFI
    - Name DFI is misleading, but essentially provides the physical interface to the DiRAM4
    - Takes command and data path signals from scheduler and drives them to DiRAM4
    - Converts from SDR to DDR
    - Converts DDR read data from DiRAM4 to SDR
    - Will control DiRAM4 read and write leveling

More details are show in the Appendix

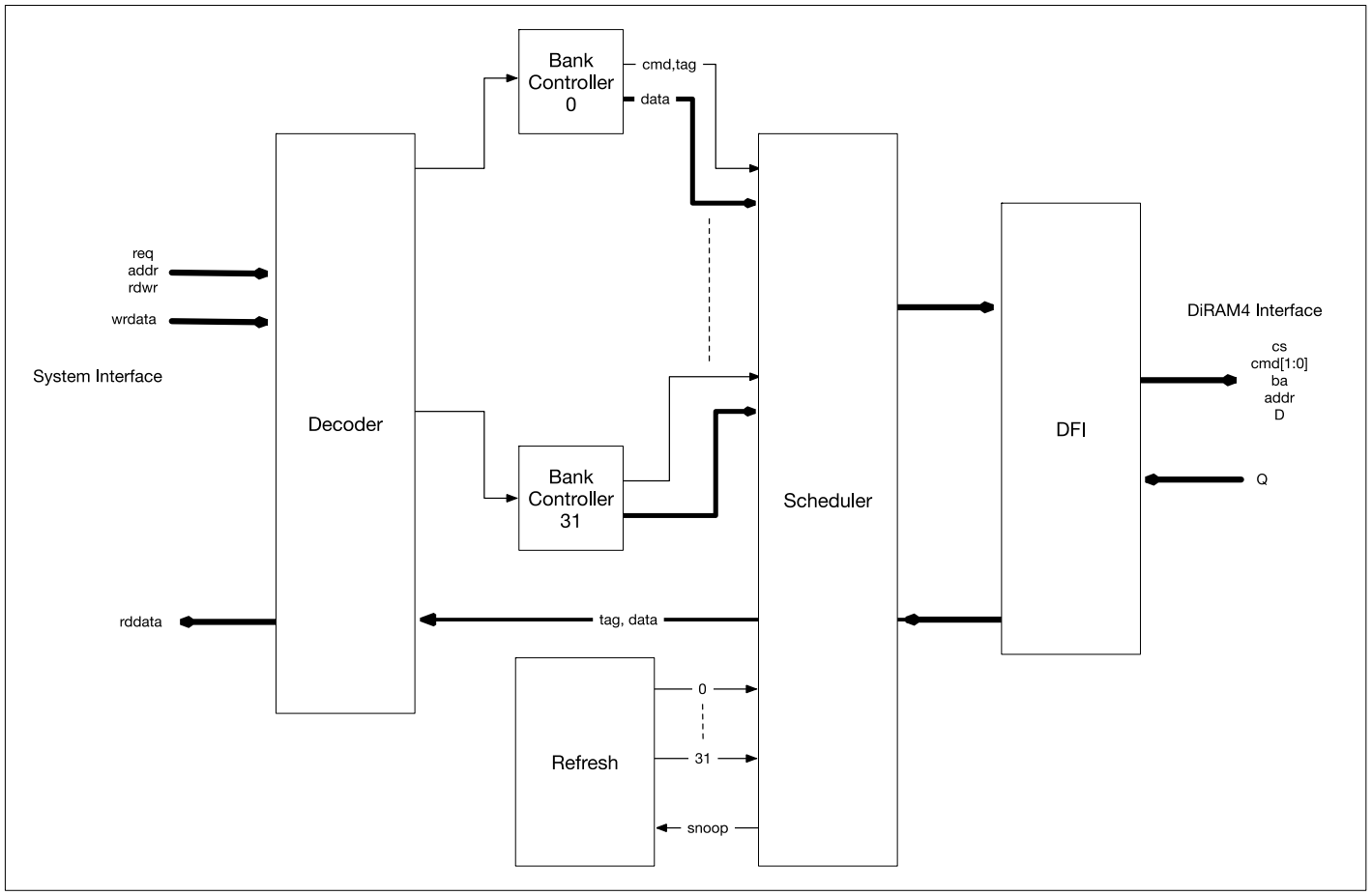


Figure 1. Block diagram of memory controller.

The details of the design, and a trial implementation in GF 65 nm led to the following results:

- Lines of code
  - http://cloc.sourceforge.net v 1.64 T=1.37 s (76.8 files/s, 24771.6 lines/s)
  - ---
  - Language files blank comment code
  - ---
  - Verilog-SystemVerilog 84 4709 4520 22380
  - Python 11 365 138 1247
  - Perl 9 16 235 197
  - Bourne Again Shell 1 16 14 49
  - ---
  - SUM: 105 5106 4907 23873
  - ---

- 596419 registers, 1153185 gates
- ~9 sqmm
- All code was synthesized

4.2 Package Design

Two interposers were design. One with the CPU on top of a DiRAM, and the second with a CPU and DiRAM implemented side by side. The first is actually a Redistribution Layer (RDL). It was found that a three layer design with 5 um features performed best from a routing and noise perspective. The second design required 6 layers (2 power/ground and 4 signal) again at 5 um feature sizes. More details are found in Appendix B.

5. Deliverables

All code was delivered as well and quarterly reports.

6. Publications

1. T. Nigussie and P. D. Franzon, "RDL and interposer design for DiRAM4 interfaces," *2016 IEEE 25th Conference on Electrical Performance Of Electronic Packaging And Systems (EPEPS)*, San Diego, CA, 2016, pp. 17-20.

7. Technology Transfer

All code was delivered directly to AFRL. The recipient hereby grants to the US Government a royalty free, world-wide, nonexclusive, irrevovable licenses to use, modify, reproduce, perform, display or disclose any data or code generated in theis project.

8. Conclusions

A memory controller for the DiRAM4 was designed and verified.

Appendix

Detailed RTL Description

Decoder

The decoder block takes read or write commands from the system interface and directs them to the appropriate bank.
The current mapping of system address to DiRAM4 bank and page address are based on:

- DiRAM4 bank address = sys__mc__dram_adr[28:24]
- DiRAM4 page address = sys__mc__dram_adr[23:12]
- DiRAM4 cache line address = sys__mc__dram_adr[11:7]

Although the system address provides 29-bits, only bits 28 thru 7 correspond to a system cache line address.
We ignore the bottom seven bits. The decode signals are sys_bank_addr, sys_page_addr and sys_page_addr.

A system cacheline of 160 bytes is larger than a DiRAM4 cacheline of 80 bytes.
Therefore, the decoder block converts each system transaction to two DiRAM4 transactions. The decoder block generates the DiRAM4 cacheline address bit-0 by setting it to ‘0’ on the first DiRAM4 transaction and ‘1’ for the second transaction.

The decoder block takes read or write commands from the system interface and directs them to the appropriate bank. The current mapping of system address to DiRAM4 bank and page address are based on:

- DiRAM4 bank address = sys__mc__dram_adr[28:24]
- DiRAM4 page address = sys__mc__dram_adr[23:12]
- DiRAM4 cache line address = sys__mc__dram_adr[11:7]

Although the system address provides 29-bits, only bits 28 thru 7 correspond to a system cache line address.
We ignore the bottom seven bits.
The decode signals are sys_bank_addr, sys_page_addr and sys_page_addr.

A system cacheline of 160 bytes is larger than a DiRAM4 cacheline of 80 bytes.
Therefore, the decoder block converts each system transaction to two DiRAM4 transactions. The decoder block generates the DiRAM4 cacheline address bit-0 by setting it to ‘0’ on the first DiRAM4 transaction and ‘1’ for the second transaction.

Implementation

System transactions are loaded directly into a FIFO. The almost full level of this FIFO is used to flow control the system interface.

A small pipeline pulls data from the input FIFO.
An FSM, dec_sys2bank_cntl_state is used to examine the pipeline and determine which bank is being accessed. If the target bank is able to accept data, the FSM transfers the pipeline information directly to the bank interface. Each bank interface also has a small FIFO and the bank interface is able to accept data as long as the FIFO level is below the almost full threshold.

The FSM does not know specifically which bank is being accessed, it gets a common ready signal from the selected bank FIFO which is a mix of all the bank interface almost full flags. The FSM assumes that once a bank FIFO is ready, it is able to accept the entire transaction. This is controlled by setting the almost full threshold appropriately.

The FSM starts the transfer to the bank interface but breaks the system transaction into two transactions. This can be seen in the state names *SEND1* and *SEND2*.
The FSM is also designed to allow adjustment of the database width, so a 640-bit width will transfer two 1-cycle transactions whilst a 320-bit width will transfer two 2-cycle transactions. Note: Only the 320-bit wide transfers have been exercised.
The ability to convert a 1280-bit system transaction into two transactions has not been fully implemented.

In addition to the address/data transferred to the bank interface, the decoder also generates a tag for each individual transaction. This allows reordering to occur within the memory controller and the decoder will need to match return data with the ordered tags to ensure read data is returned to the system in the correct order.

Bank Controller

The function of the bank controller is to receive requests from the “decoder module”, re-arrange those requests if possible, then pass them on-to the “fairness module”. The fairness module then transfers the requests to the “scheduler module”.

Implementation

The incoming requests from the decoder are stored in two sets of 16 FIFOs i.e., 16 FIFOs for storing the address/tag/age signals and 16 FIFOs for storing the write data.

1. The read/write request bit, page address, column address, tag, and age are stored in one set of FIFOs (for all read/write requests)

2. The data corresponding to a write request are stored in the other FIFO set (only for write requests)

FIFO select logic - The FIFO with highest number of free entries is selected for storing these incoming requests.

The read/write request bit, page address, column address, tag, and write data are the inputs from the decoder module. The “age” field is manually assigned to a request whenever it is issued to a

bank, and is set to zero. This field is incremented whenever there is another request to the same bank.

For example, suppose R0, R1, and R2 are the requests to a particular bank (say, bank0) in that order:

a) When R0 is issued, its age is set to 0.
b) When R1 is issued, its age is set to 0; and R0's age is incremented to 1.
c) When R2 is issued, its age is set to 0; R1's age is incremented to 1 and R0's age is incremented to 2.

The requests are dispatched to the fairness module only when "fns__bnc__ready" bit is asserted. The prioritization for re-arranging the requests is as follows:

1. An age threshold is set (in bank_controller.vh defines file), and whenever the age of a request crosses this threshold, it is dispatched with highest priority. "req_age_gt_thr" task indicates if there is any such request.
2. If there is no request whose age exceeds the threshold limit, then the request with an "open page" in the bank is scheduled to be dispatched. "req_to_open_page" task indicates if there is any such request.
3. If the above two conditions fail, then the request with greatest "age" is dispatched. "req_age_gt_thr" task also indicates if this scenario exists.

The above prioritization logic is used for dispatching the requests to the "fairness module". Based on the page address of the request that is selected for dispatch, a command packet is generated by "cmd_packet_generator" task.

The command packet can be CR, CW, POCR, POCW, PCPOCR, PCPOCW, PCPOCRPC, PCPOCWPC, POCRPC, POCWPC.

- PC corresponds to page close.
- PO corresponds to page open.
- CR corresponds to cache read.
- CW corresponds to cache write.

Note: Whenever there is a request to the last page of the bank, the page has to be closed after accessing it. (This is a design specification)

Scheduler

The scheduler is the most complex module in the memory controller. It takes requests from the bank controller and refresh controller and interleaves those requests onto the DFI interface to the DiRAM4 physical interface (to be supplied by AFRL) whilst enforcing the DiRAM4 timing and access protocol.

The interfaces to the scheduler are as follows:

1. Bank controllers 0..31
   a) Page address input
   b) Bank address input

c) Column address input
d) Tag input
e) Command packet input
f) Data input
g) Control input
h) Valid input
i) Ready output

2. **Refresh Controller 0..31**
   a) Page address input
   b) Command packet input
   c) Valid input
   d) Request input
   e) Grant output
   f) Ready output
   g) Command packet(snoop) output
   h) Address(snoop) output
3. **DFI Interface**
   a) Init Done input
   a) CS (chip select) output
   b) Command0 output
   c) Command1 output
   d) Data0 output
   e) Data1 output
   f) Bank Address output
   g) Refresh address (refresh and column address multiplexed) output

The scheduler module has top level connections using FIFOs to internal modules such as CAM, CAM_REFRESH, GLOBAL_TIMER, ARBITER, BANK_UPDATE and DRIVER.

There are 8 FIFOs towards the input side of which 6 are interfaced with bank controller module for command packet, page address, bank address, column address, tag, data and 2 are interfaced with refresh controller for page address and command packet.

There are 5 final FIFOs (page commands, read/write commands, page, bank and column address) which interfaces between CAM modules and the driver which contains the final command operations to be driven onto DFI Interface

Other than FIFOs and modules there is control logic related to final FIFOs and global timer module

As there is a single global timer for both cam and refresh cam they share some signals such as cam_gtr_col, cam_gtr_row and cam_gtr_set which are multiplexed based on refresh in progress signal generated by bank update module

Write enable and write data to final FIFOs is multiplexed output from all the banks whose select logic is based on the arbiter request.

Bank Update Module

Interface Inputs

1. Grant from the arbiter
2. Bank Cam (next bank cam instruction) – 0:31 instances: need the bank cam instruction to determine if the grant was for a page request
3. Refresh Cam (next refresh cam instruction) -0:31 instances: need the refresh cam instruction to determine if there are pending refresh requests which are valid and ready to be scheduled.
4. Signal from refresh controller to stop bank-requests from getting scheduled
    a. Refresh in progress
    b. Refresh Queue Empty

Interface Outputs

1. Conflict timer set – 32bit register corresponding to each bank to stop adjacent banks from sending page requests to scheduler-arbiter
2. Global_ready – 64bit wire, granting ready to each bank or refresh request to arbiter

Implementation

1. Sets the Conflict timer to adjacent banks when the current arbiter grant is to a page-request
2. Generates the conflict timer to turn-off one cycle after assertion
3. Allocates higher priority to Refresh requests over Bank requests
    a. Gates the bank-ready signals with refresh-requests
    b. Generates refresh request based on refresh-cam and refresh-in-progress
4. Assigns Global ready signals to requests to Arbiter

Round Robin Arbiter

Functionality:

This module arbitrates among the 64 global ready requests from the Bank update module in a Round Robin manner. Out of these 64 requests, 32 belong to the bank CAMs & 32 belong to refresh CAMs. This module ensures controlled access to the final queues (shared resource) by providing Grant signal to at most ONE valid request each cycle.

Input ports:

- clock: freely running clock of 1GHz frequency.
- reset: Active high signal that sends the device into reset state

- request: 64 bit wide signal consisting of requests from 32Bank CAMs & 32 Refresh CAMs

Output ports:

- grant: 64bit wide signal indicating the winner of arbitration. At most one bit can be set any given time.

Implementation

A pointer points to the requestor that is currently eligible for access to the final queues (shared resource). Initially after reset, the pointer points to the first requestor (request[0]) and is incremented by 1 every cycle to point to the next requestor in round robin fashion. If the requestor indicated by the pointer is ready, then it is granted access by asserting the grant signal associated with that requestor. If it is not ready, then the requestor with the next highest priority that is ready is granted access.

Local variable bit_mask[k] corresponds to the bit mask of requestor 'k'. Task gen_mask returns a 64 bit mask with all the bits, between the requestor pointed by the pointer & the requestor 'k', set. This mask is useful to decide the requestor with the next highest priority when the requestor pointed by the pointer is not ready.

Possible Future enhancements:

Currently all requests are treated equally. Ideally we would like to have a priority based arbiter that places higher priority to requests from Refresh CAMs such that they are serviced before requests from Bank CAMs.

Driver

Functionality

The driver is responsible for popping data out of the final queues whenever available & driving the corresponding address, command & data signals on the DFI interface. The specification document of DiRAM memory places a constraint on the type of commands that can be driven each cycle. All odd cycles are reserved for Page commands & all even cycles are reserved for Cache commands. The Driver also keeps track of the order in which read commands were sent on the interface by storing their tags in a queue.

Note: - The first cycle after reset starts from 1.

Input ports:

- clock, reset.
- empty & read data signals of:
    - Tag queue

    - Command queue
    - Address queue
    - Bank data queue
- dfi_init_done : To indicate initialization completion of DiRAM memory.

Output ports:

- Read enable signal of:
    - Tag queue
    - Command queue
    - Address queue
    - Bank data queue
- Write enable & write data of final tag queue
- Data, address & Command signals of DFI interface

Implementation:

A finite state machine (FSM) with the following tasks performed in each state

WAIT

- Check for reset & whether initialization is complete
- If Page Command FIFO is not empty & we are in even cycle, transition to PAGE_CMD state
- If Cache Command FIFO is not empty & we are in odd cycle, then peek into the Cache commands fifo & transition to:
    - RD_CMD if we see a Cache read command
    - WR_CMD1 if we see a Cache write command
- Remain in wait state if none of the above conditions satisfy.

PAGE_CMD

- Parse the command type & drive the DFI command interface appropriately.
    - Page Open : {cs, cmd1, cmd0} = 011
    - Page Close : {cs, cmd1, cmd0} = 010
    - Page Refresh : {cs, cmd1, cmd0} = 001
- Drive the address read from Address fifos onto DFI bank address interface
- If Page Commands Fifo is not empty & we are in even cycle, remain in PAGE_CMD state
- If Cache Commands Fifo is not empty & we are in odd cycle, then peek into the Cache commands fifo & transition to:
    - RD_CMD if we see a Cache read command
    - WR_CMD1 if we see a Cache write command
- Transition to wait state if none of the above conditions satisfy.

RD_CMD

- Drive the DFI command interface {cs, cmd1, cmd0} to 010 indicating Read command.
- Drive the address read from Address fifos onto DFI cache address interface
- Write the tag value into the tag queue. This is required to know the order in which read requests were sent to the DiRAM.
- If Page Commands Fifo is not empty & we are in even cycle, transition to PAGE_CMD state
- If Cache Commands Fifo is not empty & we are in odd cycle, then peek into the Cache commands fifo & transition to
    - RD_CMD if we see a Cache read command
    - WR_CMD1 if we see a Cache write command
- Transition to wait state if none of the above conditions satisfy.

WR_CMD1

- Write the tag value into the tag queues
- Transition to WR_CMD2 state

WR_CMD2

- Drive the first beat of data on to the DFI data interface.
- Transition to WR_CMD3 state

WR_CMD3

- Drive the second beat of data on to the DFI data interface.
- Drive the DFI command interface {cs, cmd1, cmd0} to 011 indicating Write command.
- Drive the address read from Address fifos onto DFI cache address interface
- If Page Commands Fifo is not empty & we are in even cycle, transition to PAGE_CMD state
- If Cache Commands Fifo is not empty & we are in odd cycle, then peek into the Cache commands fifo & transition to:
    - RD_CMD if we see a Cache read command
    - WR_CMD1 if we see a Cache write command
- Transition to wait state if none of the above conditions satisfy.

Refresh Controller

Refresh Controller is responsible for sending page refresh commands. It is closely tied to scheduler. It generates and sends commands to scheduler which propagates them to DiRAM. The frequency with which the commands are generated and number of pages that are refreshed successively can be set by user and are modeled as inputs from configuration block. The only constraint on these inputs is, they should be configured in a way that all the pages must be refreshed within the refresh rate period of the DiRAM.

Implementation

The refresh controller block is interfaced to scheduler with 32 separate interfaces, each interface per bank (total 32 interfaces for 32 banks). The top level block is R*efresh Controller*. It contains *Refresh Counter* block and *Page Table* block.

Refresh counter block essentially contains a counter. It starts at zero and saturates near the value set by the configuration block (currently saturates at value given by configuration block $-\ 100$). All 32 banks raises n page refresh requests before the counter saturates. The value of n comes from configuration block, which is equal to number of back to back refresh commands to be sent for one bank and is same for all banks. (bank0, bank1, bank2, ..., bank31 all raise n commands)

Currently, refresh requests for banks 0 – 4 – 8 – 12 – 16 – 20 – 24 – 28, say batch1 are raised at a time. Similarly, requests for other banks are raised. (1 – 5 – 9 – 13 – 17 – 21 – 25 – 29 (batch2), 2 – 6 – 10 – 14 – 18 – 22 – 26 – 30 (batch3), 3 – 7 – 11 – 15 – 19 – 23 – 27 – 31 (batch4), all banks in a batch raises page refresh requests at a time).

The controller working can be understood from a single bank view, say bank 0 and same can be applied to all other banks. Bank 0 raises page refresh request to scheduler when the counter's count is 1 and waits for a grant from it. The significance of this grant signal is explained below, in a paragraph before calculation section. After a grant is received, a *packetized command* is generated by controller and is sent to scheduler. A valid signal is also made high after receiving the grant and **packet generation is completed**. This signal says scheduler to accept the command generated and scheduler only accepts the command from refresh controller when **valid is high**. The packetized command is of two types which are explained below. Refresh request signal, valid signal stays high till n pages are refreshed from this bank and goes low after that. Similarly, other banks work.

Snooping the scheduler to DFI interface and command types

The Refresh controller receives a signal from scheduler which is essentially the final interface from scheduler block to DFI block. Refresh controller constantly observes this interface and checks for page open and page close commands. This part is done by *Page Table* block. If there is any page open command, it stores the address of the open page and address of the bank to which this page belongs to in a *look up table*. A look up table containing bank addresses and

page addresses is maintained in the ***page table block***. At the time grant is received, this look up table is checked to see if any page is open against the bank for which the grant is received. If there is a page open, it generates a *type 2* packetized command otherwise *type 1* packetized command is generated.

Type 1: This packetized command informs scheduler to only refresh the page, **PR** operation is performed.

Type 2: This packetized command informs scheduler to close the page, refresh the page and reopen the closed page, a sequence of **PC-PR-PO** page operations are performed.

When a type 2 command is generated, scheduler gets page address and bank address from the page address and bank address interfaces respectively to scheduler from refresh controller.

The significance of refresh grant signal from scheduler is that, once this signal goes high, the values contained in look up table become final and no more page operations will occur till all n pages are refreshed. This makes sure that **correct** *packetized command* is constructed.

Refresh period Calculation

The Configuration Block provides two inputs to refresh controller block. Input2 is chosen by the user and input1 is calculated in the following way.

Input1 = value near which the counter saturates, say x

Input2 = number of pages to be refreshed in a bank per refresh request, say n

If clock period is equal to 1ns, refresh rate is equal to 32ms and if n = 1, i.e., input2 = 1 then $x =$ input1 is calculated in the following way:

$$4096(1 + x) = 32 \times 10^6$$

The generalized equation is:

$$\frac{4096}{n}(n + x) = \beta \times \gamma \times 10^6$$

where,

- n is input2
- x is input1
- β Is refresh rate in milliseconds
- γ Is clock period in nanoseconds

Note: Here, refresh rate has the same meaning as given in technical specification document (Table 5, page 12).

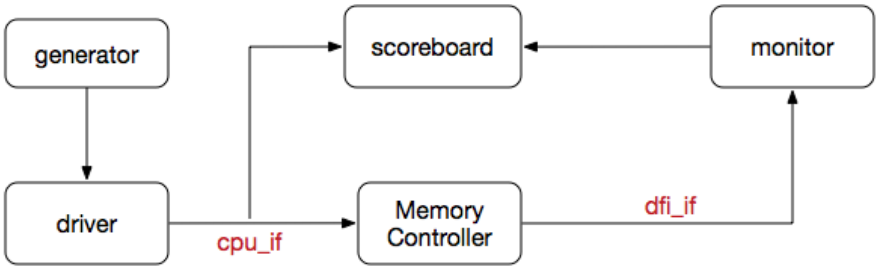
Summary

Currently, when a page is open and if input2 from configuration block is greater than 1, type2 command is generated by controller and the same is sent for all n. If sending type2 command for all n can be avoided in any way and is made sure that the data is not lost, the controller's performance would improve as there might not be any need to perform 3 operations (*PC-PR-PO*) for all n.

Currently, 64 clock cycles gap is kept between adjacent batches of banks. This can be adjusted or may be configured as an input from configuration block. Also, a batch contains 8 banks, this may also be changed or configured as an input.

Verification

Block Diagram



Basic test bench structure

The testbench is connected with the whole memory controller, mc.v file. Three probing interfaces are added to test the internal function of the memory controller, which are dut_probe_dec, dut_probe_bnc, and dut_probe_sch.

Generator

The generator creates constrained stimulus which will is sent to the driver using a mailbox. In this stage test, the stimulus generated is a write followed by a read instruction. These same set of instructions will be sent to all 32 banks, and between each bank, there is a cycle of empty instruction.

Driver

The Driver will receives the instructions generated by generator and sends those same instructions to the scoreboard and memory controller. The interface connected to the driver and memory controller is cpu_ifc. There is a callback to connect the scoreboard and driver.

Scoreboard

The scoreboard is made up of three parts, golden_dec, golden_bnc, and a simplified golden reference for scheduler.

The golden_dec implements the functions of the decoder module in memory controller.

The golden_bnc implements the basic function of the bank controller module. Currently the re-order function is not modeled.

The golden_sch is essentially a queue which keeps track of the cmd packet which have been sent to the scheduler and keeps comparing the input of the monitor to the content of the queue to see if the right instructions are send out of the scheduler.

Monitor

The Monitor is connected with the memory controller using dfi_ifc interface. It also connects with the scoreboard using callback.

Implementation

Top Level Module

In the top level verification module (top.sv), I have made an instance of the Memory Controller (MC) module (RTL) and made connections via interface to the System and Memory sides of the MC module. The system side is the users' side, i.e. we send transactions through system end of the MC. While, the memory side is where DiRAM4 is connected. Along with the MC, an instance of DiRAM4 memory is also instanced and is connected to the memory side of the MC.

NOTE: There is a README.txt for instructions on how to run the verification testbench.

I have made instances of various interfaces (discussed in detail below) which I have then connected to the required modules and/or passed to the testbench.

There is an instance of testbench as well in the top module. And in this instance, I have passed all the interfaces (System, Memory and Probes).

Testbench:

The function of testbench (testbench.sv) is to receive the interface signals, passed from top module, into its module and pass it to the environment while allocating a new environment object. Also, there is a small initial block inside the testbench which performs the following functions in a sequential manner:

a) env = new (Sys2Dec, probe_dec, probe_bnc, probe_fns, probe_sch, probe_rfc, Mc2Mem); //This statement passes the interface to the environment by overriding its default constructor.
b) env.build();
   //This is an in-built function in environment class which allocates space and passes required interface to various components in the test environment (generator, driver etc).
c) env.run();
   //This function runs the test environment and spawns parallel operations of all the environment components.
d) env.wrap_up();
   //This function marks the end of the verification test simulation and it simply displays "Complete!" on the transcript at the end of complete simulation.

Environment

Environment (environment.sv) is a class which creates objects of generator (gen), driver (drv) and scoreboards & checkers of decoder (dec), bank controller (bnc), scheduler (sch) and refresh (rfc) sub-modules.

Environment's new function retrieves the interfaces passed from the testbench into their corresponding virtual interface copies. Since we can deal with only virtual interfaces in the driver and scoreboard classes, hence we define virtual interface separately (definition.sv) which we import in the classes where we need the interface signals.

Build function allocates space to generator, driver and scoreboards, as well as, passes the required virtual interface, events and mailboxes to the required classes.

Run function basically starts the verification environment by spawning the respective functions of generator, driver and scoreboards in parallel. If any golden reference is to be disabled, it can simply be commented out and it will not affect anything (make sure to NOT comment out g_dec.run() as this function is directly linked to driver and commenting it would pause driver for eternity, instead comment g_dec.check() to disable golden model of decoder). When the event "final_tran" is triggered by the generator (indicating that final transaction is passed), the run function runs for a fixed number of cycles (to let that final transaction to go into the MC) after that and stops.

After the run function stops, wrap_up function simply displays "Complete!" showing that the verification simulation has ended.

Generator

The dynamic generator (generator.sv) creates the transaction based on certain constraints and sends them to the driver. This generator is slightly unconventional in its working, i.e. it is capable of generating 3 different types of transactions based on the users' specification. It can generate fully random transactions, i.e. random read/write requests to random address with random data. This is done mainly to test overall functionality of MC. Secondly, it can generate

transactions corresponding to a simple testbench, i.e. one read and one write request to each bank. This is required to test the working of Scheduler block. Also, the page and column address remain same throughout this type of testing. Finally, we can generate bank specific transactions, i.e. random transactions (random page, column, read/write and data) corresponding only one bank are generated. This type of testing is required to ease out the verification of Decoder and Bank controller modules.

After generating an instruction, the generator puts that into a mailbox connected between generator and driver. After putting it in the mailbox, it waits for an acknowledgment signal from driver to proceed further.

If a write transaction is generated, 4 transactions are sent out (each transaction contains 320 bits of data) for the same address. While, only 1 transaction is sent for a read transaction (since we need only 1 cycle to send the address).

To toggle between different types of transactions, search for the declaration of the variable "instr_type". If its value is set to 0, then the generator generates random transactions; for value equal to 1 it generates simple testbench scenarios and for value equal to 2 it generates bank specific transactions. The bank number (for third case) is specified by changing the value of the variable "bank_type" (declared just below instr_type).

There are 3 different classes for these 3 different transaction type (found in base_instr.sv). The page number and column number can be modified in their respective class definitions.

Driver

The driver (driver.sv) performs two basic functions, namely driving the instruction to the DUT (drv.run()) and to maintain the request signal from system side based on the behavior of busy signal from MC to system.

run();
//This task takes the object from generator to driver mailbox and passes it to the DUT via virtual system interface at every positive edge of the clock, depending upon "request" signal, i.e. if request is high. This task also passes this same transaction to the golden model of decoder via mailbox between driver and golden decoder and waits for the acknowledgment from the golden decoder module.

run_req();
//This task continuously checks the busy signal from the MC and asserts/deserts the request signal based on that only. If the MC is busy, then request goes low, else it becomes high.

Scoreboard and Checkers

Apart from generator and driver, the environment contains golden models (scoreboards) and checkers of respective sub-modules of the MC. The environment passes the required virtual

interface probes to the objects of the scoreboards requiring for operation and comparison. Following is the description of the scoreboards designed for various sub-modules.

Decoder Scoreboard

Decoder scoreboard and checker are encapsulated in the class “golden_dec” (golden_dec.sv). The new function retrieves the mailbox from driver containing transactions sent out by driver to the DUT and probe interface connections coming out of the actual Decoder DUT (for comparison).

Run function in this class takes these transactions from the driver mailbox and break it into two memory transactions (since one system transaction is broken into two memory transactions). This is done since every column address corresponds to 640 bits of data (but the request is of 1280 bits). Therefore, suppose for a write transaction, one memory transaction would have even column address and the subsequent transaction would contain its odd counterpart (LSB of column address changes from 0 to 1 from memory transaction 1 to memory transaction 2). Also, each memory transaction is broken into two bursts. This is so because the data bus is of 320 bits, hence, it takes 2 cycles to transfer 640 bits of data to one column address.

Similarly, for a read transaction (one cycle), we break it into two transactions (two cycles) such that the first transaction has even column address and the second one has odd column address.

The decoder takes out the bank number from the address (addr[28:24) and sends the decoded transaction to that particular bank controller mailbox. In my golden model, I have created 32 checker mailboxes (one for each corresponding bank controller) and after decoding the instruction, that object is pushed in the respective mailbox for checking. Also, the decoder signals the driver after it has taken out the transaction from the mailbox.

The signals that decoder generates are:

- dec__bncx__valid: To make sure that the decoded transaction goes into the correct bank controller “x”.
- dec__bncx__page_addr: Page address of the decoded transaction to bnc x.
  dec__bncx__col_addr: Column address of the decoded transaction to bnc x.
  dec__bncx__rdwr: Read/write signal to bnc x.
- dec__bncx__cntl: Control signal of the decoded transaction to bnc x. This indicates whether the burst of a particular memory transaction is the initial burst (Start of Message) or last (End of Message) or both (i.e. Start of Message and End of Message).
- dec__bncx__tag: Tag bits indicate the ID (sequence) of the transaction. This value is assigned using a universal counter which increments whenever a new memory transaction is received.

“Check()” function of golden_dec class takes the transaction from all (32) the checker mailboxes and compares with the actual DUT signals. The checker waits until any valid signal changes from low to high (signaling a new transaction has come out of decoder) and then compares the value with the golden model values.

If there is any error in the decoder module, the checker displays it in the following format:

<time(ns)>: DECODER :: <incorrect signal> for <respective bank> FROM GOLDEN MODEL: <expected value>.

This format does not display the DUT value (to reduce confusion), but the time instant can be used to find it on the waveform viewer or can be modified accordingly.

The class containing the list of signals coming out of decoder are specified in the file "dec2bnc_instr.sv".

Bank Controller Scoreboard

Bank controller scoreboard (golden_bnc.sv) replicates the behavior of all the 32 bank controller DUTs to produce the output sequence in a similar fashion. Bank controller essentially stores the incoming transactions from the decoder, rearranges them if possible and passes on to the fairness module.

Rearranging is performed on the basis of a priority logic. The priority logic states that any request that corresponds to the open page should be sent out prior to other transactions. Whenever a new transaction from decoder is received by the bank controller, it assigns an age parameter to it. The age increments whenever a new transaction is received.

According to the priority logic, the transaction whose age has crossed a threshold age should be sent out first irrespective of open page. Second priority is given to the transaction that has the page number same as open page, also has the highest age among such transactions. Finally, if the above two conditions fail, then we send out the transaction with the highest age.

Based on the open page information, bank controller associates a command packet signal with each transaction. Consider an example, suppose an initial request to a certain bank, page and column address is a read transaction, then, the bank controller would send out the command packet "POCR" (meaning Page Open Cache Read). If the next transaction is corresponding to the same page, then we send the command packet as "CR/CW" depending whether it is a read or write. If the next transaction would have been corresponding to another page (not open page), then the command packet would have been "PCPOCR/PCPOCW" (meaning Page Close Page Open Cache Read/Write). Hence, rearranging is required to remove PCPO extra commands.

Also, if the cache line is last in that particular page (i.e. 6'b111111), then we close the page after doing read/write.

Bank controller golden model contains 3 basic functions:

- Ins() function takes the new transactions coming out of the actual Decoder DUT and puts them into queue. Also it assigns an age parameter to the stored transactions.
- Del() function performs the task of removing the required transaction from this queue based on the priority logic employed. Removal takes place only when the ready signal

from fairness module is high. After removal, the open page information gets updated. After removal, it is put into a checker mailbox and sent for checking.
- Check() function compares the actual DUT value from the bank controller and compares the value coming from the golden model mailbox.

Checker would display the error in the following format:

<time(ns)>: BNC :: <Incorrect signal> from <bank controller no.> <expected value>

Apart from the list of signals that were coming out of the decoder module, an additional "bncx__fns__cntl" signal is also produced by the bank controller which contains that command packet information.

Golden models of Schedule and Refresh are designed by Sachin and Pralekha respectively and can be found in their description reports.

Interfaces

All the interfaces are available in the file "interface.sv". This file contains the system interface, i.e. signals corresponding to System side of MC. DFI interface signals contain signals coming out of the MC which go into the DiRAM4 memory instance. It also contains the probe signals, i.e. the signals coming out every sub-module.

Possible Future enhancements:

Generator: A much more comprehensive generator to test the design rigorously. A strongly constrained generator is required to achieve a good functional coverage.

- Coverage: Comprehensive Coverage parameters have to be defined to test the design in maximum possible scenario. As of now, the coverage parameters are defined (though not much) are defined in the file "coverage.svh".
- Cycle to Cycle check: After significant bug reduction in the design, the MC should be tested cycle-to-cycle.
- Golden model of read path: Since read path is a latest design, hence a golden model of this sub-module should also be designed.

List of Acronyms

CAM	Content Addressable Memory
CPU	Central Processing Unit
Cpu_ifc	CPU Interface Controller
CR	Cache Read
CS	Chip Select
CW	Cache Write
DDR	Double Data Rate
DiRAM	Disintegrated Random Access Memory
DFI	DDR PHY interface
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DUT	Device Under Test
Dut_probe_bnc	Device Under Test probe bank controller
Dut_probe_dec	Device Under test probe decoder
Dut_probe_sch	Device under test probe scheduler
FIFO	First In First Out
FSM	Finite State Machine
Golden_bnc	Golden bank controller
Golden_dec	Golden decoder
Golden_sch	Golden scheduler
Instr_type	instruction type
MC	Memory Controller
Mc2Mem	Memory Controller to memory
PC	Page Close
PCPOCR	Page Close Page Open Cache Read
PCPOCRPC	Page Close Page Open Cache Read Page Close
PCPOCW	Page Close Page Open Cache Write
PCPOCWPC	Page Close Page Open Cache Write Page Close
PC-PR-PO	Page Close-Page Read – Page Open

PHY	Physical (interface)
PO	Page Open
POCPR	Page Open
POPCCW	Page Open Page Close Cache Write
POCR	Page Open Cache Read
POCW	Page Open Cache Write
probe_bnc	probe bank controller
probe_dec	test probe decoder
probe_sch	test probe scheduler
RD_CMD	Read Command
RDL	Redistribution Layer
Rfc	Refresh controller
RTL	Register Transfer Language
SDR	Single Data Rate
SV	System Verilog
Sys2Dec	System to Decode module
WR_CMD	Write Command