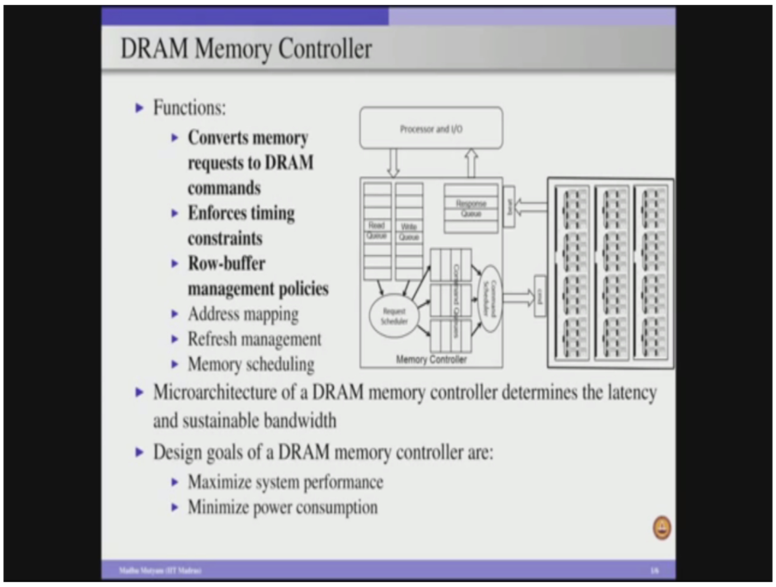
Computer Architecture

Prof. Madhu Mutyam
Department of Computer Science and Engineering
Indian Institute of Technology, Madras.

Module – 04
Lecture - 11
Memory Hierarchy Design (Part 6)

So, in the last module, we discussed the basics of DRAM based memory, and in this module, and the next module we are going to look at the details of DRAM based memory controller design. And the DRAM based memory controller provides, an interface between processor and I/O devices and the DRAM based memory. And through this interface the processor and I/O devices can access data, from the DRAM memory systems.

(Refer Time Slide:00:45)



DRAM Memory Controller

- Functions:
  - **Converts memory requests to DRAM commands**
  - **Enforces timing constraints**
  - **Row-buffer management policies**
  - Address mapping
  - Refresh management
  - Memory scheduling
- Microarchitecture of a DRAM memory controller determines the latency and sustainable bandwidth
- Design goals of a DRAM memory controller are:
  - Maximize system performance
  - Minimize power consumption

The DRAM memory controller needs to equip with several functionalities, and these functionalities include converting the memory request to DRAM commands, enforcing timing constraints, efficient row buffer management policies, and the address mapping mechanisms, the refresh managements and finally the memory scheduling. Because the DRAM based memory controller needs to do all these things. So, as a result the microarchitecture of the DRAM memory controller need to have the several components, to ensure that the proper functioning of these different things.

So, if we look at microarchitecture of a DRAM based memory controller, it consists of collection of queues. It consists of schedulers, there is a request scheduler, there is a command scheduler, and there is a queue which stores all the responses coming from the DRAM device, and there is a command queue, which stores all the commands, which are converted from the processor requests. And similarly, we have set of queues to store read requests and write requests from the processor and the I/O devices.

So, in this particular the microarchitecture, we consider two separate queues for reads and writes from the processor and I/O devices. But, again so we may come up with different microarchitectural designs. So, similar to the processor microarchitecture, where for the same ISA, we can have variety of processor microarchitectures.

In the same way, in this memory controller design also, to perform the basic operations such as processing the processor and I/O requests and supplying the data from DRAM devices back to processor and I/O, memory controllers can be designed using variety of microarchitectures, and each one of the microarchitecture will have a specific power performance values.

So, effectively as I mentioned earlier, the DRAM based memory controller provides a design space exploration consists of various performance, and power points. So, we need to come up with an efficient memory controller design, for reducing the overall latency and maintain sustainable bandwidth. So, effectively the design goals for any memory controller are maximizing the system performance, minimizing power consumption and also we may consider minimizing the overall Die area.

So, we can design a DRAM based memory controller to optimize any of these design points, or we can come up with a memory controller design, that can optimize a combination these design points. So, in this module we are going to look at three key functionalities of DRAM based memory controller, such as converting the memory request into DRAM commands, enforcing the timing constraints and row buffer management policies.

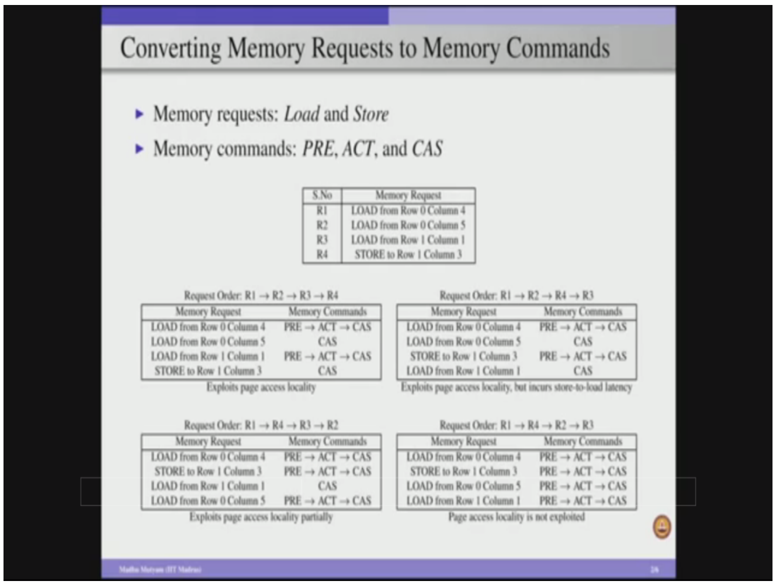
And in the next module, we are going to look at address mapping, the refresh management, as well as the memory scheduling techniques. So, first we start with converting processor generated memory requests to DRAM commands. We know that a processor or I/O generates a memory request, such as load and store to access data from the memory.

So, typically when there is a load request, it specifies a load and an address, indicating that we have to go to that particular address in the memory, and get the data stored in that address. Similarly, when there is a store instruction, we specify the address and the content indicating that, this content needs to be written to the address specified in the store instruction. Now, we need to convert these load and store instructions into DRAM understandable memory commands.

We already discussed in the previous module, the basic DRAM commands are PRE, ACT and CAS. We know that, whenever we want to perform any access operation on DRAM array, we need to make the DRAM array or the entire DRAM bank into a certain voltage level so that the activation command can be performed successfully. In other words we need to make, we need to reset the sense amplifiers and the bit lines associated with the DRAM bank on which we are going to perform an operation.

To do that, we issue a command called a PRE or a pre-charge. Once, we issue this pre-charge, the bit lines and the sense amplifiers of the corresponding DRAM bank will be reset, so that following ACT command can be performed. So, we need to apply a PRE before issuing an ACT command, and after issuing an ACT command, the data from a selected row is written on to the sense amplifiers. And then we can select the required data from these sense amplifiers, by using a command called as CAS, Column Access Strobe.

(Refer Time Slide:07:04)



Converting Memory Requests to Memory Commands

- Memory requests: *Load* and *Store*
- Memory commands: *PRE*, *ACT*, and *CAS*

S.No	Memory Request
R1	LOAD from Row 0 Column 4
R2	LOAD from Row 0 Column 5
R3	LOAD from Row 1 Column 1
R4	STORE to Row 1 Column 3

Request Order: R1 → R2 → R3 → R4

Memory Request	Memory Commands
LOAD from Row 0 Column 4	PRE → ACT → CAS
LOAD from Row 0 Column 5	CAS
LOAD from Row 1 Column 1	PRE → ACT → CAS
STORE to Row 1 Column 3	CAS

Exploits page access locality

Request Order: R1 → R2 → R4 → R3

Memory Request	Memory Commands
LOAD from Row 0 Column 4	PRE → ACT → CAS
LOAD from Row 0 Column 5	CAS
STORE to Row 1 Column 3	PRE → ACT → CAS
LOAD from Row 1 Column 1	CAS

Exploits page access locality, but incurs store-to-load latency

Request Order: R1 → R4 → R3 → R2

Memory Request	Memory Commands
LOAD from Row 0 Column 4	PRE → ACT → CAS
STORE to Row 1 Column 3	PRE → ACT → CAS
LOAD from Row 1 Column 1	CAS
LOAD from Row 0 Column 5	PRE → ACT → CAS

Exploits page access locality partially

Request Order: R1 → R4 → R2 → R3

Memory Request	Memory Commands
LOAD from Row 0 Column 4	PRE → ACT → CAS
STORE to Row 1 Column 3	PRE → ACT → CAS
LOAD from Row 0 Column 5	PRE → ACT → CAS
LOAD from Row 1 Column 1	PRE → ACT → CAS

Page access locality is not exploited

Madhu Mutyam (IIT Madras) 28

Effectively for each load and store, we have to convert these instructions into memory commands which are PRE, ACT and CAS. But, again it is not necessary that each of load and store instructions need to be converted into PRE, ACT and CAS. So, depending on the context, we can convert our load and store instruction either into a one memory command in two or all the three.

So, we will discuss this with an example, consider a scenario where we have four requests, the first 3 are load requests and the fourth one is a store request. And all these 4 requests are to the same bank, but within that bank these requests can go to different ranks. For example, request R1 wants to read the data from row 0 column 4, while request R2 wants to read the data from the same row, but from a different column and request R3 wants to read from a different row from the same bank and from column 1. And finally, the request R4 which is a store instruction, wants to write to row 1 at column 3. So, now once we have these set of requests, now we will see what happens if we issue these requests in the same order that is R1, R2, R3, and R4.

Now what happens and how many memory commands we have to issue to a particular command queue, which is associated with a particular bank. And remember that all these requests are to the same bank. So, we are going to insert the corresponding memory commands into the same command queue. So, let us consider the first scenario, where we are going to follow the request order as R1, R2, R3, and R4.

First, we issue R1, followed by R2, followed by R3 and finally we issue R4. Whenever, we want to process the first request R1, we assume that we need to issue a PRE command, so that the entire DRAM bank can be reset in such a way that our following ACT command can be performed successfully. So, effectively when I issue R1, we need to perform a PRE to reset the sense amplifiers and the bit lines of the bank. An ACT command, so that we can select row 0 from the bank and finally, we have to issue a CAS command to read the data from column 4 from the selected row which is stored in the sense amplifiers. Now, after this request R1 is converted into a combination of memory commands, which consists of PRE, ACT and CAS. Now, when we go for the next request, which is R2.

As we know that R2 is also to the same row, but to a different column. So, for this request R2, we do not have to again go for precharge activation. Because, we can still exploit the row buffer locality, that is present in the sense amplifiers because already the same row is read into sense amplifiers by using the previous request. So, when we want to issue the second request R2, we can directly get the data from the sense amplifiers.

So, as a result so, when we want to convert an R2 which is following R1. So, all we require is, we just have to give only CAS command, and here we are assuming an open page policy. Again this open page and closed page policy, we are going to discuss after a couple of foils. And when we come to the third request, which is a load request to row 1 and column 1.

Now, unlike the previous request R2, request R3 cannot exploit the row buffer locality because request R3 is to a different row as compared to the previous two requests, those are to row, row 0. So, as a result there is a row conflict. So, first thing we have to do whenever there is a row conflict is that, we have to write the data from the sense amplifiers back to the row 0 and we have to reset the sense amplifiers and the bit lines of the bank.

So, for that we issue our PRE command, so after issuing the PRE command now, we can go ahead with activating row 1 by using an ACT command so that row 1 will be read into sense amplifiers. And then finally, we can apply CAS command to read the data from column 1 and finally, when there is a store instruction that is R4 which is also to the same row as that of a request R3.

So, we can still again exploit, we can exploit the row buffer locality, so that all we have to do is, we can issue only CAS command for this request R4. Effectively for these 4 requests, if they are issued in this order R1, R2, R3 and R4, we have to convert this 4 requests into a total of 8 memory commands, and these 8 memory commands are stored in our bank wise command queue in the order.

So, that we start selecting a request from this bank one after another, the commands and then issue that to the DRAM memory system. So, that the DRAM based memory system will supply the data. And remember here, we are for load and store instructions we are considering just the CAS command. But, actually there are two types of CAS commands one is the read CAS, and the write CAS.

Whenever, we want to perform a load operation we have to issue a read CAS and whenever we want to perform a store operation we have to issue write CAS. But, for the simplicity we are considering just the CAS command. Now, we will consider another scenario, where we change the request order. So, here the first 2 requests are following the same order, but the last 2 requests are interchanged. So, as a result we issue R1 first, followed by R2, followed by R4 and finally R3 is selected.

If I consider this, since there is no change in the ordering of R1, R2. So, there is no change in the type and the number of memory commands as that of the previous one. But, since R4 is coming immediately after R2, again R4 is to a different rank, so we have a row buffer conflict. So, we have to issue a PRE command. So, effectively PRE is in the critical path for performing our R4 operation, and after that we issue an ACT command to activate row 1.

And finally, we can apply CAS command to perform a store operation on column 3, and after that we issue an R3, which is also to the same row as that of the store instruction. So, as a result we can exploit row buffer locality, and convert this r our last instruction into a single command, which is a CAS. So, effectively here also we convert these 4 instructions into 8 memory commands as that of the previous one. But, there is a catch. In the first ordering, we have a load followed by a store. But, where as in the second ordering a store followed by a load. So, effectively whenever we are performing a store operation and after that if we are going to issue a load operation because the I/O gating resources are occupied by previous store operation.

So, we have to wait for some amount of time so that this write operation is going to release this I/O getting resources, and then we can perform the following load operation. So, that is the reason why even though these two orderings are going to give you the same number of memory commands into the command queue but, the second ordering is going to take slightly more time mainly because, of mainly the store to load latency.

Now, we will consider the third ordering which is R1, followed by R4, followed by R3, followed by R2. So, in this case so, as usual there is no change in the issue of the first and second which is always R1. So, we start with a pre, ACT and CAS. But, now there is an R4 which is actually a store instruction to a different row. So, as a result we incur a row conflict so again we have to precharge the bank so that we can activate row 1 to perform a store operation.

And after that we will perform a CAS operation, and then there is a load instruction to the same row that is row 1 so, we can exploit row buffer locality. But, still we incur store to load latency and finally, there is a request R2 which is to a different row as that of the previous request R3. So, again we need to close the previous opened page and we perform a PRE, followed by ACT and CAS. Effectively if we use this ordering so, we are going to exploit locality to a certain extent.

But, not a full extent as that of the first combination, and we also incur store to load latency, and effectively these 4 requests are converted into a total of 10 DRAM memory commands, and which are stored in the command queue. And finally, when we consider the other reordering request ordering, which is R1, followed by R4, followed by R2, followed by R3.

So, here if we see each of these requests are addressing to a different row. So, as a result, for each of these things, each of these requests, we have to pre-charge before performing activation. So, as result we incur a significant penalty, and this is the worst request ordering among all these 4, and we require a total of 12 memory commands for these 4 requests.

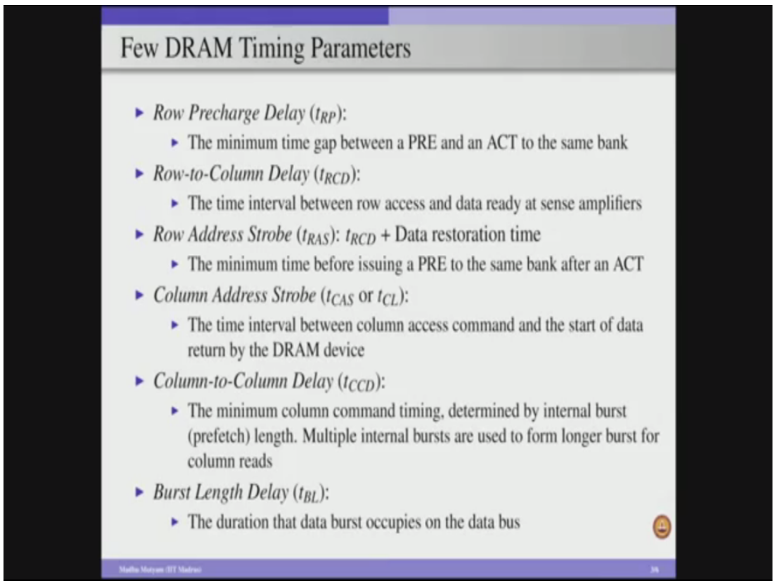
So, effectively this says that the selecting a request from a request queue matters a lot in terms of the number of memory commands, we are going to push on to the command queues. And note that the command queues have a limited number of entries, and we have to efficiently utilize this limited space, available in these command queues. And also another thing is if we issue an unnecessary PRE, we going to degrade our overall performance as well as it consume extra power consumptions.

And similarly, unnecessary ACT is also going to increase our overall, the access time as well as the power consumption. We have to select always a proper request ordering from the read and write request queues, such that the total number of memory number of memory commands generated from these requests should be minimal. So, having discussed this converting the memory request into memory commands now, we are going to look at few DRAM timing parameters.

Note that, the DRAM memory controller needs to satisfy several timing constraints. But, we are not going to discuss all the timing parameters that are associated with the DRAM memory controller due to the lack of time. So, we are going to concentrate on few very important DRAM timing parameters, and whenever the DRAM controller is going to issue a command, memory command, to the DRAM device, it has to respect several timing parameters because we know that the DRAM devices are passive components.

And it just relies on the memory controller to supply requests and as soon as there is a request from the memory controller, DRAM device is going to supply the requested data. So, as a result in order to ensure the proper functioning of the entire system, the memory controller needs to take care of several timing parameters and it has to respect several timing parameters. So, as a result the entire DRAM controller design is constrained by these timing constraints, and this also, these timing parameters also specify the overall performance of the memory system.

(Refer Time Slide: 20:52)



Few DRAM Timing Parameters

- *Row Precharge Delay* (t_{RP}):
  - The minimum time gap between a PRE and an ACT to the same bank
- *Row-to-Column Delay* (t_{RCD}):
  - The time interval between row access and data ready at sense amplifiers
- *Row Address Strobe* (t_{RAS}): t_{RCD} + Data restoration time
  - The minimum time before issuing a PRE to the same bank after an ACT
- *Column Address Strobe* (t_{CAS} or t_{CL}):
  - The time interval between column access command and the start of data return by the DRAM device
- *Column-to-Column Delay* (t_{CCD}):
  - The minimum column command timing, determined by internal burst (prefetch) length. Multiple internal bursts are used to form longer burst for column reads
- *Burst Length Delay* (t_{BL}):
  - The duration that data burst occupies on the data bus

So, we start with row precharge delay. So, we know that the DRAM device needs to be pre-charge before, we perform any activation command on the bank. This is mainly because, whenever we perform a previous ACT command, the bit lines and the sense amplifier voltage levels are different. So, we need to reset the voltage levels on the bit lines, and the sense amplifiers before, performing activation to a different row in the same bank.

So, that is specified by t_{RP} and this is the minimum time gap between a PRE command and an ACT command to the same bank. Which is nothing but, if I issue a PRE command to a DRAM bank at time t, I can issue an ACT command to the same bank only after $t + t_{RP}$ time. I cannot issue my ACT command before, $t + t_{RP}$.

If I issue an ACT command before that, the DRAM system may supply wrong data. So, in order to ensure that the correct data is transferred by the DRAM device, we have to wait, we have to instruct the memory controller to issue the ACT command, only after t_{RP} time from the issuance of the PRE command to the same bank. So, the next one is row to column delay. So, this is once I perform a PRE command so that, the entire DRAM bank is preset so that, we can perform an ACT command on that particular bank.

So, once I issue an ACT command so that is once I assert row address strobe, the address higher bits of the address is sent to the row decoder and which is going to select a particular row in the DRAM bank, and that data is read from this elected row in the DRAM bank, and the data is written on to the sense amplifiers. So, the time it takes between the row access, and the data ready at the sense amplifiers is called as row to column delay.

This specifies that the minimum time you have to wait, before reading the data from the sense amplifiers after issuing an ACT command. In other words, if I issue an ACT command at time t, I can issue a CAS command to the same bank to read the data from the sense amplifiers only after a '$t + t_{RCD}$', where t_{RCD} is the row to column delay. So, we issued a PRE command so that the entire bank is preset or precharged and then we applied an ACT command so that a particular row is selected.

And the data from the selected row is written on to the sense amplifiers. Now, we know that the DRAM cells are self-destructive, when I say DRAM cells are self-destructive, whenever we perform any access operation on DRAM cells, the DRAM cells cannot retain the same charge. And as a result, the value represented by the DRAM cells will be inconsistent with the actual value stored in it, so in other words whenever we perform any activation, it is our duty to ensure that data is restored properly.

And this is specified by Row Access Strobe or Row Access Strobe delay, which is called as a t_{RAS} which is nothing but, the sum of t_{RCD} and the data restoration delay. So, this also says that, so when I issue an ACT command, I cannot issue a PRE command to the same bank within a time of t_{RAS}. In other words, if I issue an ACT command to a particular row in a given bank at time t, I can issue a PRE command to the bank only after '$t + t_{RAS}$' time.

Now, we write a particular row on to the sense amplifiers and now the sense amplifiers have the data. Now, we want to perform a column read from the selected from the sense amplifiers. To do that, there is a timing parameter which is t_{CAS} the t_{CAS} specifies the Column Address Strobe delay, and this is the time interval between the column access command, and the start of the data written by the DRAM device.

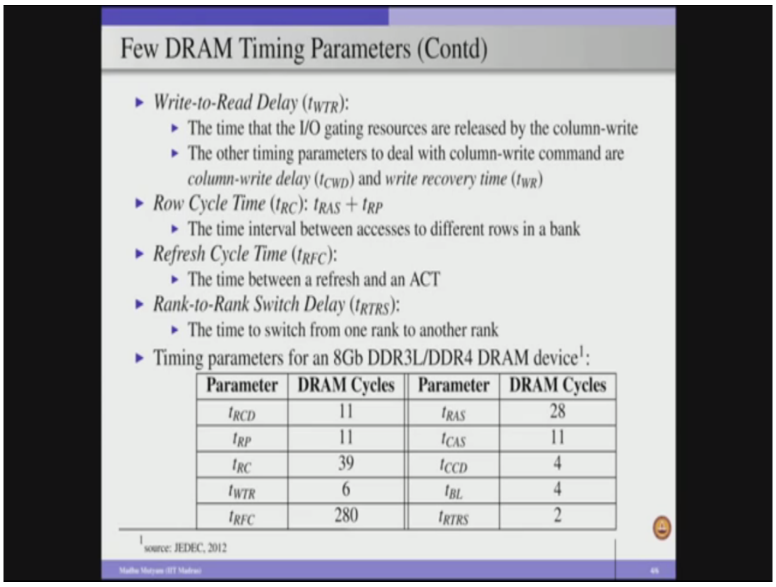
Once I issue a column address command, once I assert CAS - Column Address Strobe, the lower part of the address is sent to the column decoder and the column decoder will select the appropriate data from appropriate position in the sense amplifiers, and it starts writing the data to the output. So, effectively the t_{CAS} specifies that the time interval between a column access command, and the start of the data written by the DRAM device. It takes some amount of time after which data can be start returning by the DRAM device.

Now, there is another key parameter, which is column to column delay. So, note that internally the DRAM device consists of a collection of DRAM arrays and these DRAM arrays will supply data in a prefetched manner. When I issue a single column read command, based on the prefetch width of the DRAM device, we are going to get multiple bits read from the DRAM arrays, and the prefetch width is going to specify the time gap between a readings from one column to the other column.

So, that time parameter is called as the column to column delay or t_{CCD} and this is the minimum column command timing determined by the internal bus or a prefetch length. So, we internally transfer the data from this DRAM arrays in a connection of internal bus length, which is specified by the prefetch, and once we have this data then we will pack everything into our data output unit.

And from there, we can transfer the data on to the data bus by using our external the burst length which is specified by t_{BL} which is going to be discussed, in the next point. As the next point, the t_{BL} specifies the duration that the data bus occupies on the data bus so, as a result this data bus cannot be available for performing any other operation within this t_{BL} time. But, this is with respect to performing a read operation.

(Refer Time Slide: 28:26)



Few DRAM Timing Parameters (Contd)

- *Write-to-Read Delay* (t_{WTR}):
  - The time that the I/O gating resources are released by the column-write
  - The other timing parameters to deal with column-write command are *column-write delay* (t_{CWD}) and *write recovery time* (t_{WR})
- *Row Cycle Time* (t_{RC}): $t_{RAS} + t_{RP}$
  - The time interval between accesses to different rows in a bank
- *Refresh Cycle Time* (t_{RFC}):
  - The time between a refresh and an ACT
- *Rank-to-Rank Switch Delay* (t_{RTRS}):
  - The time to switch from one rank to another rank
- Timing parameters for an 8Gb DDR3L/DDR4 DRAM device[1]:

Parameter	DRAM Cycles	Parameter	DRAM Cycles
t_{RCD}	11	t_{RAS}	28
t_{RP}	11	t_{CAS}	11
t_{RC}	39	t_{CCD}	4
t_{WTR}	6	t_{BL}	4
t_{RFC}	280	t_{RTRS}	2

[1] source: JEDEC, 2012

But, when we are performing a write operation, we need to consider an important parameter which is called as write to read delay. So, in this write to read delay t_{WTR} whenever, we perform a write operation, so this write operation, occupies write operation is holding this I/O gating resources. So, whenever we want to issue a read operation following this writes operation, we have to ensure that this I/O gating resources have to be released by the column writes command.

And to do that, we have to incur some amount of time that is called as write to read delay or store to load delay, which is specified by t_{WTR}. So, as a result whenever we are performing read and write operations, we have to ensure that these read and write operations should not be interleaved. We have to perform a contiguous continuous stream of read operations, and then make a switch over to write operation and then perform sequence of write operations, and then switch back to read operations.

If we do that, then we can eliminate this t_{WTR} delay from our critical path. These are the two other important timing parameters, we have to consider for performing write operations. One is the column write delay, and the other one is the write recovery time. So, next we will consider row cycle time, and in this row cycle time so, this is like we know that whenever, we perform an ACT command we read the data, and then we restore the data back.

And finally, if we want to perform any other activation command, we have to ensure that the bank is precharged. So, that is specified by t_{RP}. Effectively, between two activation commands, we have to complete a full cycle of operations, which are activating a row, restoring the data back to the activated row, and then precharging the bank which is specified by '$t_{RAS} + t_{RP}$'. Effectively our t_{RC}, row cycle time is equal to sum of t_{RAS} and t_{RP}.

So, this specifies that, whenever we want to issue a subsequent ACT command to a particular bank, we have to wait for t_{RC} time. And we know that we have to perform a refresh of the DRAM cells at regular intervals, to ensure that the data integrity is maintained. In order to do that, we will perform a refresh operation and there is a time associated with this refresh operation, and that time is called as a refresh cycle time.

And this refresh cycle time, this specifies as the time at which, time by which the DRAM device is not available for performing any other operations on the DRAM device. When I issue a refresh command at time t, I cannot perform any other useful operations on the DRAM device, until '$t + t_{RFC}$' and this refresh cycle time depends on the capacity of the DRAM device.

For example, if the DRAM device has 8k rows associated with it, and as specified by the JEDEC standard that for each DRAM device we issue 8k refresh commands. So, for each refresh command, we can refresh one row associated with that. So, as a result our t_{RFC} is going to be the time associated with refreshing a single row.

When I say refreshing a row, we have to read the data from a row and restore it back and then perform a pre-operation. And finally, we consider a rank to rank switch delay, we know that a DRAM controller can consists of multiple ranks and data can be placed on multiple ranks, and if we are going to read the data from these multiple ranks in an interleaved fashion then we have to incur rank to rank delay.

This delay is required mainly because, our rank is sharing the command bus, data bus, and the address bus and so on. So, in order to release this shared resources associated with one rank to be mapped to the second rank, we incur some amount of time and that time is called as, the rank to rank switch delay. Now, having discussed these different the timing parameters, we will see what these values in terms of DRAM cycles are.

When we consider the latest DDR4, the DRAM devices, and these values are taken from JEDEC 2012 reference manual. And you can see here t_{RFC} takes almost 280 DRAM cycles, and t_{RCD} row to column delay takes 11 DRAM cycles, Row pre-charge takes 11 cycles, and t_{RAS} row address row strobe takes 28 cycles, and since t_{RC} is the sum of t_{RP} and t_{RAS}, which is going to take 39 cycles, and rank to rank switch takes 2 cycle time and column to column delay takes 4 DRAM cycles and write to read is going to take 6 DRAM cycles.

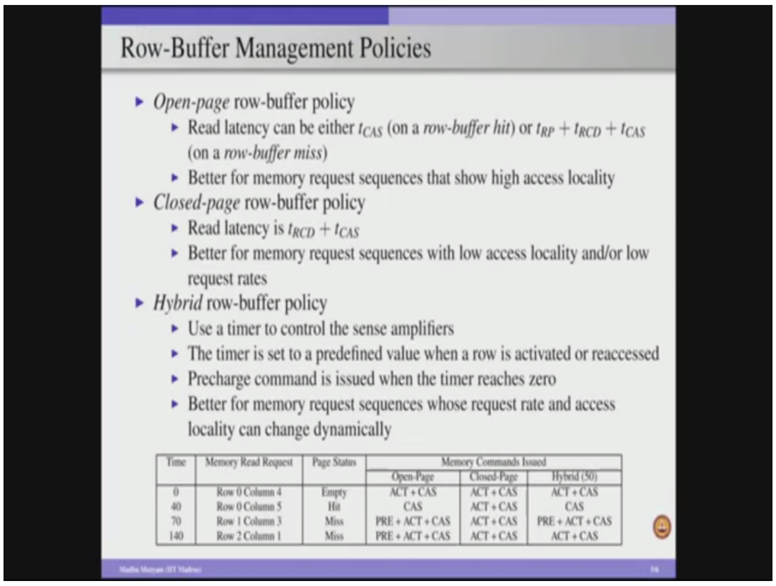
And similarly, the bus length is going to take 4 DRAM cycles. So, as a result like the main reason, why the DRAM access latency is in terms of hundreds of processor cycles is mainly because of these timing parameters. Actual data transfer on the bus is going to take place within few cycles. But, to get the data from the DRAM device, we need to take care of all these parameters, and which are going to take significant amount of DRAM cycles, and which translates into hundreds of processor cycles at the end.

So, that is the reason why the DRAM access latency is taking hundreds of processor cycles and that is actually widening the performance gap between the processor and the DRAM. So, as a result in order to overcome this memory wall problem, there is a significant amount of research going on in the architectural community to come up with efficient memory controller designs. And finally, we are going to look at the row buffer management policies.

So, we know that when we activate a row, the data stored in that row will be stored in the sense amplifiers and these sense amplifiers can act as a buffer. Since, these sense amplifiers are going to store the entire data stored in a row of a DRAM bank. We are going to say, we are going to call these sense amplifiers also as a row buffer. And now once the data is there in the row buffer, now I can service subsequent requests which can get a hit in this row buffer.

So that, we can eliminate further activation and a pre-charge of the DRAM bank that improves our overall performance as well as the energy consumption. Now, we are going to see what the different types of row buffer management policies are, which are applied in the current memory controller designs.

(Refer Time Slide: 35:58)



Row-Buffer Management Policies

- *Open-page* row-buffer policy
  - Read latency can be either t_{CAS} (on a *row-buffer hit*) or $t_{RP} + t_{RCD} + t_{CAS}$ (on a *row-buffer miss*)
  - Better for memory request sequences that show high access locality
- *Closed-page* row-buffer policy
  - Read latency is $t_{RCD} + t_{CAS}$
  - Better for memory request sequences with low access locality and/or low request rates
- *Hybrid* row-buffer policy
  - Use a timer to control the sense amplifiers
  - The timer is set to a predefined value when a row is activated or reaccessed
  - Precharge command is issued when the timer reaches zero
  - Better for memory request sequences whose request rate and access locality can change dynamically

Time	Memory Read Request	Page Status	Memory Commands Issued		
			Open-Page	Closed-Page	Hybrid (50)
0	Row 0 Column 4	Empty	ACT + CAS	ACT + CAS	ACT + CAS
40	Row 0 Column 5	Hit	CAS	ACT + CAS	CAS
70	Row 1 Column 3	Miss	PRE + ACT + CAS	ACT + CAS	PRE + ACT + CAS
140	Row 2 Column 1	Miss	PRE + ACT + CAS	ACT + CAS	ACT + CAS

Madhu Mutyam (IIT Madras) 56

So, we first start with an open page row buffer policy. As the name says open page, so once a row is activated, the data of that row is stored in the sense amplifiers, and the data from the sense amplifiers will not be written back to the DRAM bank to the specific row location until there is a row conflict because of new request coming from the processor.

So, whenever my next request from the processor is to the same row as that of the previously opened row, we call it as a row hit, and whenever there is a row hit, we can supply the data with a latency of t_{CAS}, and if the next request is to a different row, then we are going to get a row conflict. And because of a row conflict, we incur a row miss. And whenever there is a row miss, we have to close the previously opened row by using a PRE command which is going to take a t_{PRE} time.

Then, we have to activate the new row, that is going to take t_{RCD} time, and finally we have to read the data from a specified location from the opened row. That is going to take t_{CAS} time. So, effectively whenever there is a row conflict or a row miss, we are going to incur a time of '$t_{RP} + t_{RCD} + t_{CAS}$'. But, this open page policy is efficient for all the applications which are going to exhibit high row locality.

If there is a high row locality, all you can do is, all you can do is you can activate a row once and then service subsequent requests from the sense amplifiers or the row buffer. So, the other extreme of this open page policy is a closed page policy. As the name suggests closed page policy, as soon as you perform an operation you close this particular row. So, closing this page is as soon as a CAS is performed, we apply a PRE command so, that the entire bank is in a precharged state.

So that if there is subsequent request then we can start with the activate command. We do not have to perform a PRE. Effectively, PRE can be eliminated from the critical path of an accessing or activating a row. So, effectively in the case of closed page policy our read latency is the sum of t_{RCD} which is nothing but the time taken for an activating a row, and then t_{CAS} which is the time taken for reading the data from the sense amplifiers or a row buffer. And this type of policy is efficient for all those applications, which exhibit a very low access locality or low request rates. If applications are not generating frequent requests to a DRAM bank, then we can always go for this closed page policy, and in intermediate to this open page, and closed page policy is a hybrid row buffer policy.

In this hybrid page policy whenever we activate a row, the data is stored in the sense amplifiers and rather than closing this row after performing a CAS operation immediately as that of a closed page policy, we keep this row open for some amount of time. And there is a fixed amount of time defined, and once this time is elapsed, and if there is no request happens to the opened row, then we are going to close this opened page. And, if within that time interval if there is a request to the opened row, then again we reset the counter, so that again this page will be opened for again the fixed amount of time and it continues further. So, effectively rather than closing an opened row immediately after a CAS, we keep the opened row for some amount of time. But, we are not going to keep the row indefinitely until there is a row conflict like an open page policy.

So, we can say the fixed amount of duration, and we specify this by using a timer associated with our memory controller and the timer decrements its count, and whenever the counter becomes 0, then we are going to close the open page. And whenever, if there is an access to the opened row, the timer reset to the fixed value again. So, this way we can implement this hybrid row buffer policy. And this can take care of the dynamic nature of access requests and access locality of applications. And this is mainly, it is better for memory request sequences whose request rates, and the access locality changes dynamically. So, having discussed these 3 row buffer management policies now, we are going to see the effectiveness of these 3 policies using an example. So, consider a set of requests which are generated at different times.

For example, the first request is generated at time 0, second request is generated time 40, third request is generated at time 70 and last request is generated at time 140. And initially the Page status is empty, here when I say page status is empty. So, the bank is precharged so that there is no content in the sense amplifiers, and the bit lines and the sense amplifiers are at proper voltage levels. So that, whenever we want to issue a new request, we can straight away start with an ACT command to that.

So, that is what is called as the page empty status. When I say page hit, that indicates that my next request is going to hit in the open page which is stored in the row buffer. And similarly, if there is a page miss that indicates that, my next request is conflicting with the page that is opened in the row buffer. Now initially, we assume that the page is empty, and when I issue the first request which is to row 0 and column 4 in the case of open page policy.

I have to issue an ACT command and a CAS command, because I know that the state of the page is empty. So, I can start with an ACT command so that we can activate a row and store the data on to the sense amplifiers. So, the row buffer is now going to have data associated with row 0 and then we can apply a CAS command so that, we can read the data from column 4 stored in this particular open row buffer. In the case of the closed page policy, because the page status is empty again I have to go for ACT and CAS.

Similarly, even in the case of hybrid policy also for my initial request, I have to go for ACT and CAS. So, effectively since the page status is empty in all these three policies from the first request, I am going to incur the same number of commands memory commands, and also we assume that in this hybrid policy. We consider a counter value of 50, that indicates that a recently opened page will be opened for 50 time units and within that time frame, if there is any access to the page again then the counter again reset to 50, and if there is no access to the open row within this fifty time units, then I am going to close that page. Now, when we consider the second request which comes at time unit 40 and this is to row 0 and column 5.

So, since this is to the same row so in the case of open page policy I am going to get a row hit and effectively I incur only one memory command that is CAS. Effectively using a CAS command, I can service my second request. But, whereas in the case of closed page policy, I know that once I perform a previous CAS command, I am going to perform a PRE immediately, and which may take may be some 20 or 30 units of time.

After that, again it going to bring, the closed page policy is going to bring, the state of the page to be an empty. So, as a result you have to again perform an ACT command to activate row 0 and then perform a CAS command to read the data from column 5. So, effectively, even when the application exhibits row locality, the closed page policy cannot exploit that.

But, whereas, open page policy can efficiently exploit the row buffer locality, that is present in the memory access request. And in the case of hybrid policy, because previously we accessed the row at time 0, this page will be stored in the, will be retained in the, sense amplifiers for the next 50 units, and our second request came only at 40 time units. So, as a result our hybrid policy also gets a row hit, and using a CAS command, we can service the second request in the case of hybrid policy.

Now, when the third request comes at time unit 70, which is to a row one, which is effectively a miss, there is arrow conflict. And in the case of open page policy, we have to close the previously opened row, which incurs a PRE command, and then an ACT command to activate row one, and then a CAS command to read the data from column 3. But, whereas in the case of closed page policy, after the previous request is done, we issue a PRE command and by that time the third request the PRE is already completed.

So, the row buffer is in the empty state. So, we can again go for an ACT command and CAS command. Effectively, now, for the third request, our open page policy is incurring an extra delay, because of a PRE in the critical path, whereas in the closed page policy, PRE is not in the critical path. So, effectively we can service the third request efficiently if we use a closed page policy. When we consider hybrid policy, we know that in the hybrid policy, the row is retained for a fixed amount of time duration in the sense amplifiers. But, our third request is to a different row. As a result, we incur a penalty in terms of issuing a PRE command, and then issuing an ACT and CAS. So, effectively the hybrid policy is going to take the same amount of time, as that of open page policy.

So, effectively for the third request, the closed page policy is efficient as compared to the other 2 policies. Now, consider the last request which is issued at 140 times which is effectively after 70 time units, compared to with respect to the previous request, and this is to another row. So, effectively again we incur a penalty, because of a row conflict open page policy going to take a overall time of t_{PRE}, t_{RAS} and t_{CAS}.

But, whereas in the case of closed page policy, because there is a 70 unit time gap between the third request and the fourth request, and by that time the PRE is already completed in the closed page policy. So, still the closed page policy requires only ACT and the CAS even for the fourth request also like the other 3 requests. But, in the case of hybrid policy, we know that the previous row that is row 1 is opened at time 70 and the timer for the hybrid policy is having a value of 50.

So, that means the row 1 will be, row 1 is written in the sense amplifier for 50 time unit starting from time unit 70. So, effectively it was there in the sense amplifier until the time unit 120 and then after 120 time unit, the sense amplifier is emptying the contents by applying a PRE. So that, by that time the last request which is coming at 140, our row buffer maintains an empty status in the hybrid policy.

So, as a result we incur only ACT and the CAS. So, from the last request we can clearly see that hybrid policy is better than the open page policy, and from the second request we can clearly see that hybrid policy is better than the closed page policy. Effectively hybrid policy can exploit the changes in the dynamic nature of the access requests rate and the access locality. Whereas, the open page policy and the closed page policies cannot adapt dynamically to the changes happen in the memory request.

So, because of that typically, we can go for hybrid based row management policy, instead of open page and the closed page policy. But, to implement the hybrid based policy, we require extra the hardware in terms of the timers and incrementing the timers, and resetting these timers and so on. But, those things are not there with open page and the closed page policies. So, with that I am concluding this module and in the next module, we are going to see the other three important functions associated with the memory controller such as address mapping, refresh mechanism, and memory scheduling.

Thank you