10

Main Memory

Till now, we have been treating the main memory as a static and passive array of bytes. We have been assuming that once there is a miss in the last level cache (LLC), we send a request to main memory. It takes 100-300 cycles to get the answer back, and thus sending a request to main memory should be avoided at all costs. Unfortunately, the main memory is just not a block of DRAM. There is much more to designing main memory these days. In fact, inside the main memory we have a microcosm of DRAM banks, interconnections, and controllers. There is a small component of the main memory within the CPU chip as well. It is called the *memory controller*. The role of the memory controller is to take all the memory requests from the caches, queue them, schedule them, and send them to the main memory. We shall see in this chapter that the scheduling algorithm for the memory controller is very crucial. It is a very important determinant of the overall performance.

There are several challenges in managing large memories. As of 2020, it is not uncommon to find 1 TB memories in server class systems. Managing such a large memory in terms of scheduling accesses, and distributing the bandwidth among different cache banks and I/O devices is in itself a fairly complex problem. We need to understand that memory capacity has been increasing with Moore's law (refer to Section 1); however, DRAM access latency has traditionally reduced very slowly. Hence, there is a need to design effective strategies to bridge this gap – known as the *memory wall*.

Moreover, DRAM based memories sadly lose all their data once the system is powered down. The next time that we turn on the system, all the data needs to be read from the hard disk once again. This causes an unacceptable delay. Additionally, we need to periodically refresh a DRAM, which means that we need to periodically read all the blocks, and write them back again. If we do not refresh the values, the capacitors that hold the values will gradually lose their charge and the stored data will be lost. In modern DRAMs the refresh operation causes unacceptable delays, and thus there is a need to create memory that is nonvolatile in nature, which means that it maintains its state even after the system is powered off. Such modern memories are already being used in USB drives and many chips containing them are being produced commercially. In the future, we expect them to become commonplace in computing systems starting from small embedded systems to large servers. The latter half of the chapter will focus on such nonvolatile memories.

10.1 Dynamic RAMs: Devices, Circuits, and Systems

10.1.1 DRAM Cell

Recall our discussion in Section 7.3.1 where we created a 6-transistor SRAM cell; let us do something similar here. The most significant drawback of an SRAM cell is that each cell requires 6 transistors and thus the storage capacity is limited. If we could design a memory with a much smaller cell, then we could store more bits per unit area. Keeping this in mind, let us design a very simple memory structure, which is known as a dynamic memory cell or DRAM cell. It is shown in Figure 10.1. Note that the subsequent discussion assumes that the reader is well versed with the material presented in Chapter 7.

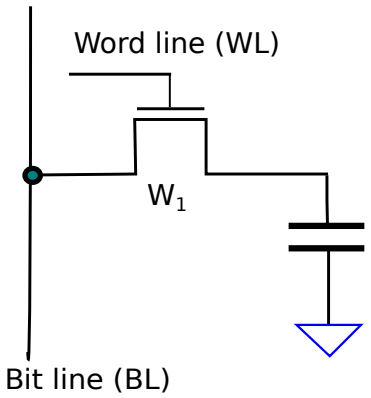

Figure 10.1: A DRAM cell

The charge is stored across a single capacitor and there is only one access transistor: W_1 in Figure 10.1. This is controlled by the word line. Recall that in the case of an SRAM cell there were two access transistors that were controlled by the word line. This is because the inverter-pair had two outputs. In this case, the capacitor has only one input/output terminal. Hence, only one word line transistor and one bit line are required.

The capacitor is particularly very important in this case because it is the charge storage device. Moreover, unlike an SRAM cell, a capacitor cannot maintain a steady voltage for a long period of time. Due to some current leakage between the parallel plates, ultimately all the stored charge will leak out. Even if the leakage current is very small, these capacitors will ultimately lose their stored charge. It is thus necessary to reduce the leakage current to as small a value as possible. The standard technique to handle this situation is that we periodically read the value of a DRAM cell and write it back. This ensures that even if the potential has dropped due to a leakage of charge, the voltage across the capacitor can be restored to the ideal level. This process is known as a *refresh*, and DRAM cells require periodic refresh operations to ensure that we do not lose any data.

Definition 89
The process of periodically reading the values of blocks in DRAM memory and writing them back is known as a refresh operation. *The capacitors in the DRAM cells gradually lose their stored charge; hence, it is necessary to periodically read their state and then restore the voltage across the capacitors to the ideal values.*

Keeping these considerations in mind, let us quickly look at the technology used to build capacitors for DRAM cells.

10.1.2 Capacitors used in DRAM Cells

To create a capacitor in silicon, we need to create two parallel plates and put a dielectric material in between. Given that we wish to maximize the density of DRAM devices, we need to make the capacitor as small as possible. Note that if the capacitor is very small, there is a possibility that its charge might leak out very quickly, and we will need more refresh operations. Hence, there is a need to strike an optimal trade-off between storage density and the maximum time between two refresh operations issued to the same cell. There are two main methods that are used to create capacitors for DRAM processes. The first type of capacitor is called a trench capacitor that is embedded in the silicon substrate. The second type is called a stacked capacitor that is above the silicon surface.

Trench Capacitors

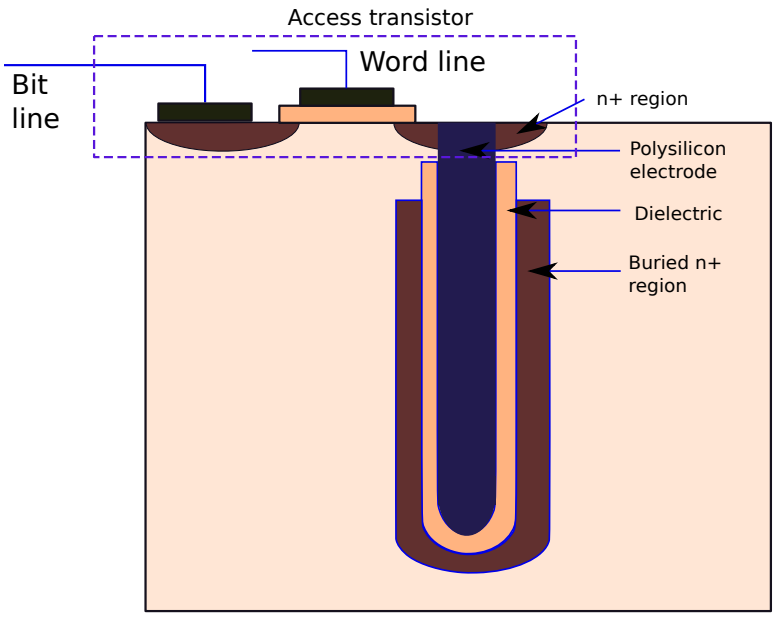

Figure 10.2: A trench capacitor along with the word line transistor.

The structure of a trench capacitor is shown in Figure 10.2. It is literally shaped as a trench or rather a deep hole in silicon. The hole is filled with a conducting material such as polysilicon. This acts like one of the plates of the capacitor, which is connected to a terminal of the access transistor of the DRAM. Often one of its electrodes is embedded within one of the terminals of the access transistor such that we do not need additional metallic connections between them. The next inner layer is made of an insulating dielectric such as Al_2O_3, HfO_2, or Ta_2O_5. This dielectric layer is typically very thin. For a 40 nm wide trench, it is typically in the range of 15-20 nm [Gutsche et al., 2005]. The only way to scale such designs is to have very deep trenches and have thin layers of dielectrics such that we can pack more capacitors per unit area. For a feature size of 40 nm, the trenches can be several microns deep (typically 4 to 6 microns), which means that the trench is 100 times as deep as it is wide! This allows us to increase its capacitance. The advantage of this design is that we can pack many such deep trenches in silicon without increasing the cross-sectional area. The dielectric is enclosed by a buried plate (or a region) made of n-type doped silicon. This acts as the other electrode, which is connected to the ground terminal.

Such trench capacitors are embedded in silicon and are ideal for embedded DRAMs in 3D chips.

We can have transistor layers or metal layers over the memory layer. These layers can have their own connections. The memory layer will not introduce any congestion or wire routing problems because it is below them.

Stacked Capacitors

Even though trench capacitors have many advantages, they have a few disadvantages as well. The major disadvantage is that the trench is hard to fabricate. Particularly at the deepest point, it is hard to guarantee the parameters of the trench. Hence, for many commercial processes, a stacked capacitor is preferred even though it requires more area and has structures above the silicon layer.

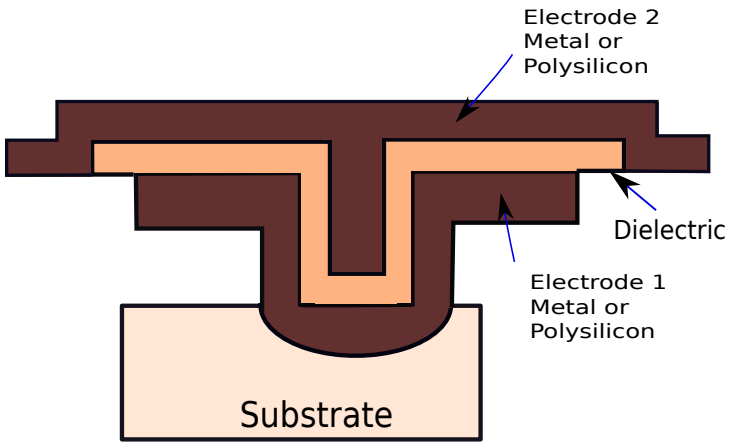

Figure 10.3: A stacked capacitor. The electrode touching the substrate is connected to one of the terminals of the word line transistor.

As shown in Figure 10.3, the stacked capacitor does not have deep trenches. It is a 3D structure, where the capacitor is fabricated in layers above the access transistor. One of the terminals of the access transistor is connected to a polysilicon electrode that is vertically stacked above it. The other electrode is also a polysilicon electrode, which is separated by a dielectric. Note that it is possible to replace polysilicon with metallic electrodes as well (depends on the process).

A stacked capacitor is still much better than a regular planar capacitor that is made on silicon because it is a 3D structure. The capacitor can be fabricated above one of the terminals of the access transistor, and thus we can increase density. Modern avatars of stacked capacitors have multiple fins and some designs have a cylindrical structure. Such designs increase the density such that we can store more bits per unit area in a DRAM.

10.1.3 Array of DRAM Cells

Let us now create an array of DRAM cells on the same lines as an array of SRAM cells. Note that in this section we shall refrain from describing the basic underlying concepts of the design of an array of memory cells. The reader is referred to Section 7.3.1.

Recall that in an array of cells, the address is first sent to a row decoder, which enables one of the word lines in the 2D array of DRAM cells. This enables all the cells in a given row, which is also called a *page*[1]. They can then be accessed by the bit lines. Note that in this simplistic design we have a single bit line per memory cell. Each bit line is connected to a sense amplifier that senses the voltage and converts it to a logical 0 or 1. Unlike SRAM arrays, in a DRAM array we can only read a single column

[1]This is different from a page in virtual memory

in one cycle. We have a column multiplexer/demultiplexer (mux/demux) that chooses the right column to read or write. It is controlled by a column decoder that takes as input a set of bits from the address. The column mux/demux is connected to read and write buffers that buffer the values that are read or need to be written. The value that is read then needs to be sent on the CPU-memory bus.

An important point to note here is that in DRAM arrays the sense amplifiers appear between the bit lines and the column mux/demux. This was not the case in SRAM arrays. In SRAM arrays the bit lines were directly connected to the muxes/demuxes, and this structure was then connected to the sense amplifiers. The reasons for this will gradually become clear over the next few sections.

Important Point 17
A row in a DRAM array is also called a page.

Read Access

Figure 10.4 shows an array of DRAM cells. Let us consider a read access. The address first arrives at the row decoder. Recall that a decoder takes n inputs and produces 2^n outputs. The n inputs encode, in binary, the number of the output that needs to be set to a logical 1. For example, if $n = 3$, and the input bits are equal to 101, then it means that the 5^{th} output (word line) is set to 1 (count starts from 0). This enables the corresponding word line, which enables all the cells in its row. The cells start setting the values of their attached bit lines. In a DRAM array we typically read an entire row at a time and buffer its contents.

Here also we can use the precharging trick, where we first set all the bit lines to a fixed voltage, which is typically half of the supply voltage ($V_{dd}/2$). Subsequently, we monitor the direction in which the voltage on the bit line is gravitating towards. If it is gravitating towards a logical 0, then we declare the bit to be 0, much before the voltage actually reaches 0 Volts, and vice versa for the case when the cell stores a logical 1. The advantage of precharging (see Section 7.3.1) is that we do not have to wait for the voltage to swing to either 0 or V_{dd}). We simply need to ensure that the voltage difference between the current voltage and the precharged voltage is more than the noise margin. This helps us significantly speed up the operation of a memory array. The reason that we can precharge the bit lines quickly is because we can use strong precharge drivers to pump in current into the bit lines; however, we do not have this luxury when a bit line's voltage is set by a feeble DRAM cell.

Now, consider the case where the capacitor in the DRAM cell stores a logical 1. When we enable the access transistor via the word line, the capacitor starts to charge the bit line. This means that stored charge from the capacitor flows towards the bit line and increases its voltage. This further means that the voltage across the capacitor in the DRAM decreases. The next time that we read this cell, the voltage across it might not be enough to infer a logical 1. This means that the DRAM cell will lose its value, which is not desirable. This phenomenon is known as a *destructive read*. The only way to avoid this situation is to ensure that we rewrite the value after it is read. This is known as *restoring* the value that has been read. This is essential in a DRAM and adds to the latency of a read operation.

Definition 90 *Once a DRAM cell is read, its capacitor loses its charge, and the cell cannot be read again. This phenomenon is known as a* destructive read. *It is thus necessary to* restore *the potential across the capacitor if it stored a logical 1.*

The circuit to detect these small voltage swings is called a sense amplifier (similar to sense amplifiers in SRAM arrays). Recall that a sense amplifier is a differential voltage amplifier that converts a small

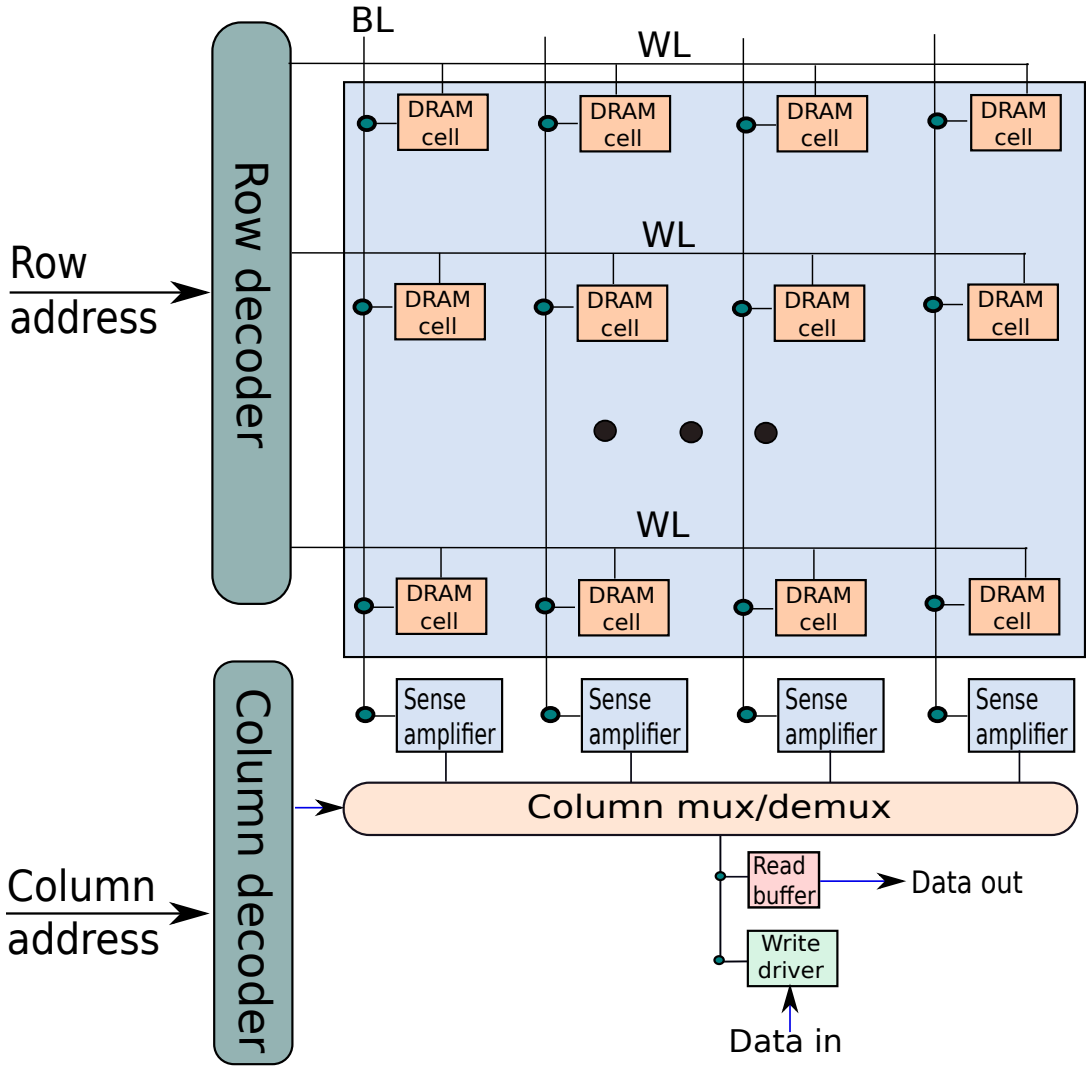

Figure 10.4: Array of DRAM cells

voltage swing to a logic level: 0 or 1. Once we have read the data and converted it into appropriate logic levels, it is buffered in the sense amplifiers. DRAM sense amplifiers are special in the sense that they function both as differential amplifiers as well as buffers. We can then choose the subset of the DRAM row that we are interested in. In a quintessential DRAM array, we typically choose a single bit to read or write to. This bit is selected using a column mux/demux that internally uses a column decoder. This data is sent to powerful driver circuits that send the data over the bus to the CPU.

Sense amplifiers for SRAM arrays have been discussed extensively in Section 7.3.1; however, DRAM sense amplifiers are slightly different. The specific differences are as follows. In an SRAM array, the sense amplifiers are placed after the column multiplexers. We first choose the appropriate set of columns, and then we sense their logic levels. However, in a DRAM array, sense amplifiers are placed before the column multiplexers. We first convert all the voltage values on the bit lines to logical 0s or 1s, and then we choose a subset of these values. The reasons for this are as follows. In a DRAM array, along with sensing the values, the sense amplifiers are also used to buffer the data, and even restore the values. Since

we need to buffer the entire row of data, we need a sense amplifier for each column. As compared to an SRAM array, this design choice does increase the number of sense amplifiers that are required; however, this is a necessity in a DRAM array because a lot of DRAM access schemes try to serve data directly from the sense amplifiers as opposed to accessing the DRAM row once again. The sense amplifiers thus act as a small cache that is much faster to access as compared to making a fresh DRAM access. With this vision in mind, let us discuss sense amplifiers next.

DRAM Sense Amplifiers

A sense amplifier has two inputs. It compares the voltage difference between them and senses the direction in which the voltage difference is progressing. If the voltage difference exceeds a positive threshold, then we can infer a logical 1, and likewise if it decreases beyond a negative threshold, then we infer a logical 0. In the case of SRAM arrays, the bit lines BL and $\overline{BL}$ were the inputs to a sense amplifier. However, in this case, we have a single bit line, and thus we have a single input. It is thus necessary to add an additional input. The naive solution is to use a line with a fixed voltage, which is the base voltage that all the bit lines are precharged to. However, this is expensive in terms of area and wiring overhead. Since we never activate two rows in the same array at once, we can divide each bit line into multiple subsections. We can then use these *bit line segments* as inputs to the sense amplifiers. Let us elaborate by discussing the two broad design paradigms in this space.

Open Bit Line Array Architecture

In this design, we split the entire array by dividing each bit line into multiple segments as shown in Figure 10.5. We then connect the bit lines for segments i and $i+1$ to the same sense amplifier. Recall that since we never activate two rows of the array at the same time, at most one sense amplifier will be activated at any given point of time. This design has two advantages: each sense amplifier is connected to two inputs without adding any additional wires, and the number of transistors connected to each bit line can be kept within limits. The latter effect is important because it limits the capacitive loading and consequent latency of each bit line.

In the DRAM world, we typically describe the area of a memory cell as a function of the feature size, F, which is the minimum size of a feature that can be reliably fabricated in a given process. The area of each cell is at least $4F^2$. This is because the DRAM cell's minimum dimensions are $F \times F$. In addition, it needs to be separated by a distance of at least F from the nearest cell. This means that the area that needs to be apportioned for each cell is $2F \times 2F$. However, we need to additionally account for the area taken by bit lines, circuitry, and also the fact that the capacitor and the transistor cannot be completely vertically stacked. Taking all of these overheads into account, the area of each cell in the open bit line architecture is around $6F^2$.

The main disadvantage of this design is that it has reduced noise tolerance. Bit lines are large structures that can pick up a lot of inductive noise. Since the bit lines that are inputs to a sense amplifier are not co-located, they can pick up different degrees of noise. Thus, this design is susceptible to more noise-induced errors.

Folded Bit Line Array Architecture

Such noise-induced errors are mitigated by folded bit line architectures that try to co-locate the bit lines that are inputs to the sense amplifiers.

Figure 10.6 shows an architecture that twists two bit lines to cover a column of DRAM cells. Counting from the top, cells 1, 2, 5, and 6 are connected to the first bit line, whereas cells 3 and 4 are connected to the second bit line. The bit lines change their direction, and intersect in the figure after every two DRAM cells in a column. Note that they do not actually intersect – they just seem to do so when viewed from the top. This ensures that for every group of two cells, one bit line is connected, and the other is disconnected. The disconnected bit line always runs parallel to the connected bit line.

The advantage of this design is that both the bit lines are in close proximity to each other. As a result,

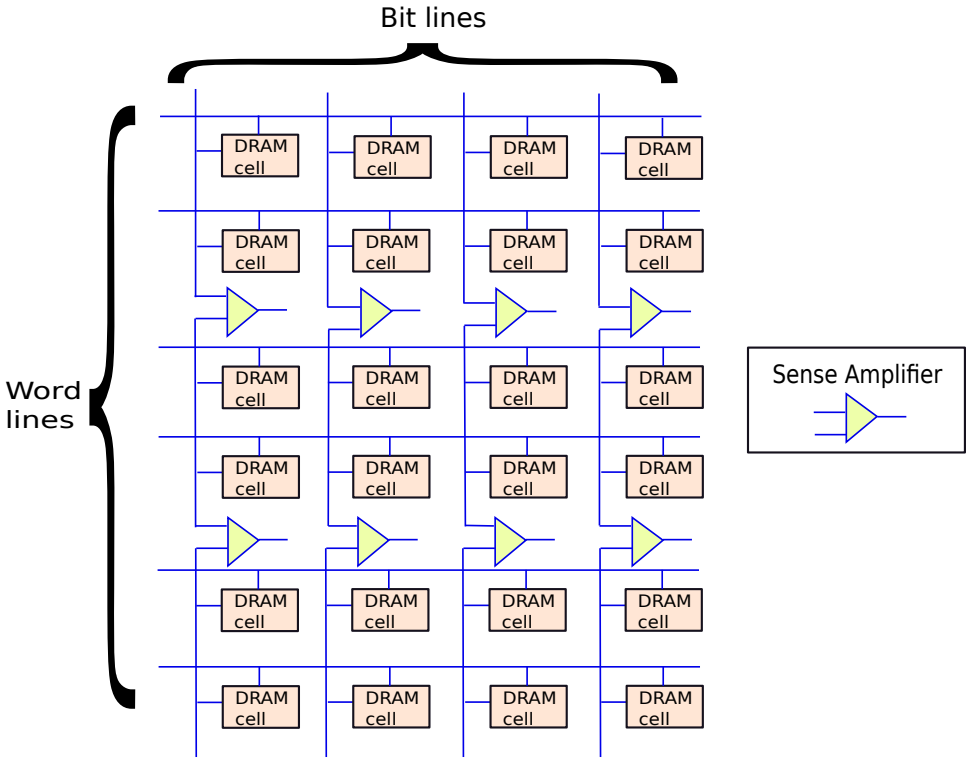

Figure 10.5: Open bit line array architecture.

they accumulate roughly the same amount of noise. Since the sense amplifier senses the difference in voltage between the bit lines, any noise that is common will get rejected. As a result, the noise tolerance of this design is much more than the architecture with open bit lines.

However, there are several shortcomings of this design as well. The first is that we need additional area for the second bit line that is disconnected. This increases the cell area even though the design is not planar. The area increases to $8F^2$ (from $6F^2$ in the open bit line array architecture). Secondly, the number of cells connected to each bit line is roughly equal to half the number of rows. This can increase the capacitance of a bit line significantly and slow it down.

Many designs for bit line array architectures have been proposed to extend these schemes and use different combinations of splitting the bit lines and folding.

Design of a Sense Amplifier

Sensing the voltage difference is a two-stage process. We first equalize the voltages of the two bit lines. This is done using the circuit shown in Figure 10.7. This is a very simple circuit that is connected to the two bit lines. When the EQ line is set to a logical 1, transistor $T1$ gets enabled. After this the potential difference between the two bit lines (1 and 2) becomes roughly zero. Next, we need to ensure that it is equal to the precharged voltage: $V_{dd}/2$. Look at transistors $T2$ and $T3$. After EQ is set to V_{dd}, transistors $T2$ and $T3$ will turn on and the bit lines will get set to the voltage $V_{dd}/2$. Once this is done, both the bit lines are said to be precharged.

Then in the second stage, we enable a row of the DRAM array and allow the bit lines to gradually get charged or discharged. Next, we need to sense the difference in the voltages between Bit line 1 and Bit line 2 (see Figure 10.8). Note that we are deliberately avoiding the notation BL and $\overline{BL}$ over here because these are two separate bit lines that are connected to different sets of DRAM cells. Assume that

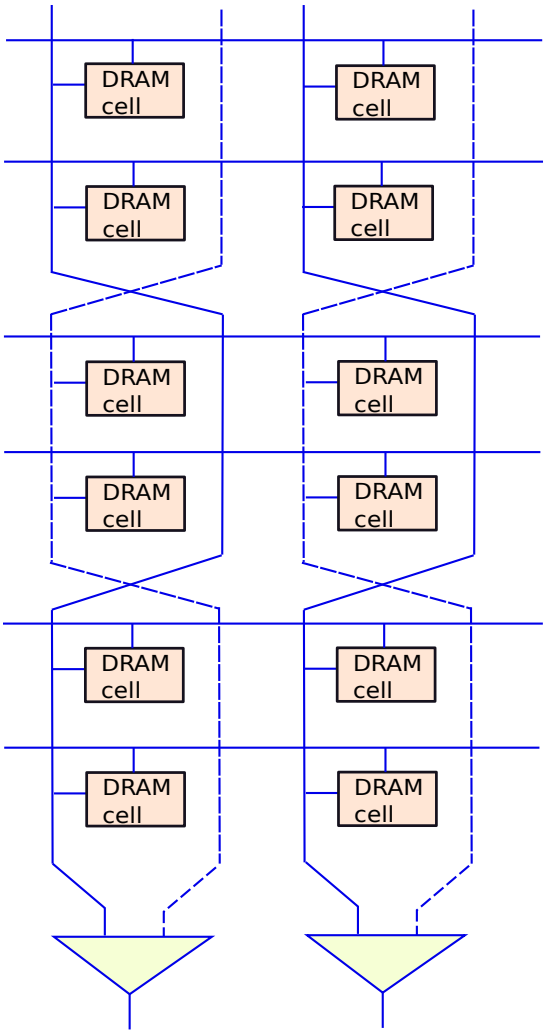

Figure 10.6: Folded bit line array architecture

the voltage on Bit line 1 (V_1) is slightly higher than the voltage on Bit line 2 (V_2), where $V_2 = V_{dd}/2$. At this point of time let us set the voltage on SAN to 0 V and the voltage on SAP to V_{dd} (assume logical 1 is V_{dd} volts). This enables the sensing operation.

The sequence of actions is as follows. Gradually, $T2$ starts becoming more conducting. As a result, the voltage on Bit line 2 dips because SAN is set to 0 V. Because of this, the voltage at the gate of $T3$ also starts dipping and this makes $T3$ more conductive. Since SAP is set to V_{dd}, the voltage on Bit line 1 starts to increase. Very quickly the voltage on Bit line 1 reaches V_{dd} and the voltage on Bit line 2 reaches 0 V. At this point, the voltages on the bit lines have reached the maximum and minimum levels respectively. We have a reverse case when the voltage on Bit line 1 stays at $V_{dd}/2$ and the voltage on Bit line 2 increases slightly. We leave it as an exercise for the reader to reason about what happens when the voltage on any bit line decreases slightly from the reference value ($V_{dd}/2$) because the value stored in the DRAM cell is a logical 0. In all cases the bit lines swing to the maximum and minimum voltage values. Also note that they always have complementary voltages. This is a stable state for the sense amplifier. The bit lines will continue to maintain their state. This design of a sense amplifier has thus

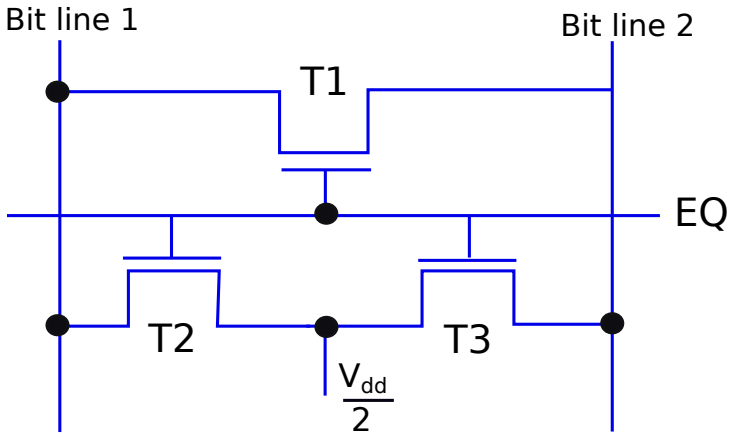

Figure 10.7: A voltage equalizer

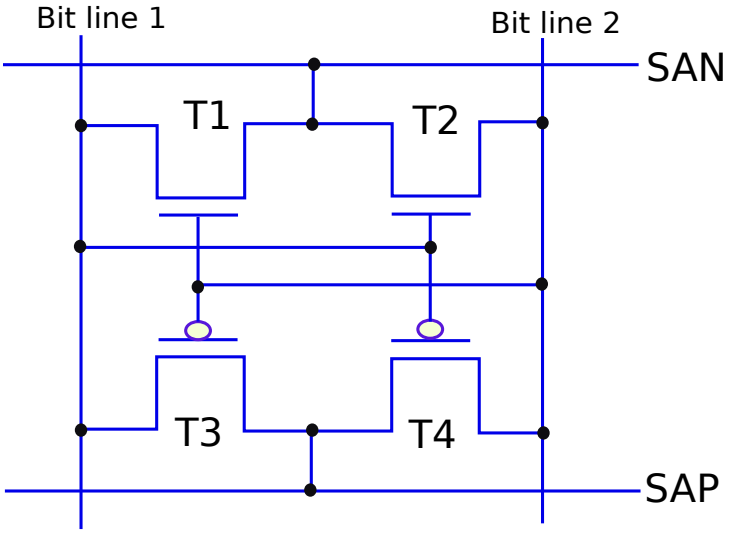

Figure 10.8: A DRAM sense amplifier

helped us to store a bit as well. Finally, note that once the respective bit line gets charged or discharged, the DRAM cell can also "restore" its value. For example, if the cell stored a logical 1, the charged bit line can restore the charge of the capacitor.

We thus see that the sense amplifier serves several purposes at the same time. First, it senses small changes in the voltages of the bit lines and amplifies the difference such that the bit lines quickly get fully charged or discharged. Once, the voltages of the bit lines have been set, they will remain that way and keep restoring the value of the DRAM cell till we disable the word line. For accessing a new row, we need to activate the equalizer circuit once again and set the voltages of both the bit lines back to the precharge voltage: $V_{dd}/2$. To disable the sense amplifier at this point, we can set the voltages of SAN and SAP to $V_{dd}/2$; this will ensure that all the four transistors are in the cut-off state.

The sense amplifier and the precharge circuit are connected to powerful write drivers via access transistors as shown in Figure 10.9. The access transistors are controlled by a chip select line (CS), which effectively enables the DRAM chip. To summarize, to read a row we perform the following actions

in sequence.

1. Precharge the bit lines. Set the voltages on the SAN and SAP lines to $V_{dd}/2$.
2. Enable the corresponding word line.
3. Set the values of the SAN and SAP lines.
4. Enable the "Chip select" and "Read enable" signals.
5. Send the column address to the column decoder within the column mux/demux unit. Read the data out.

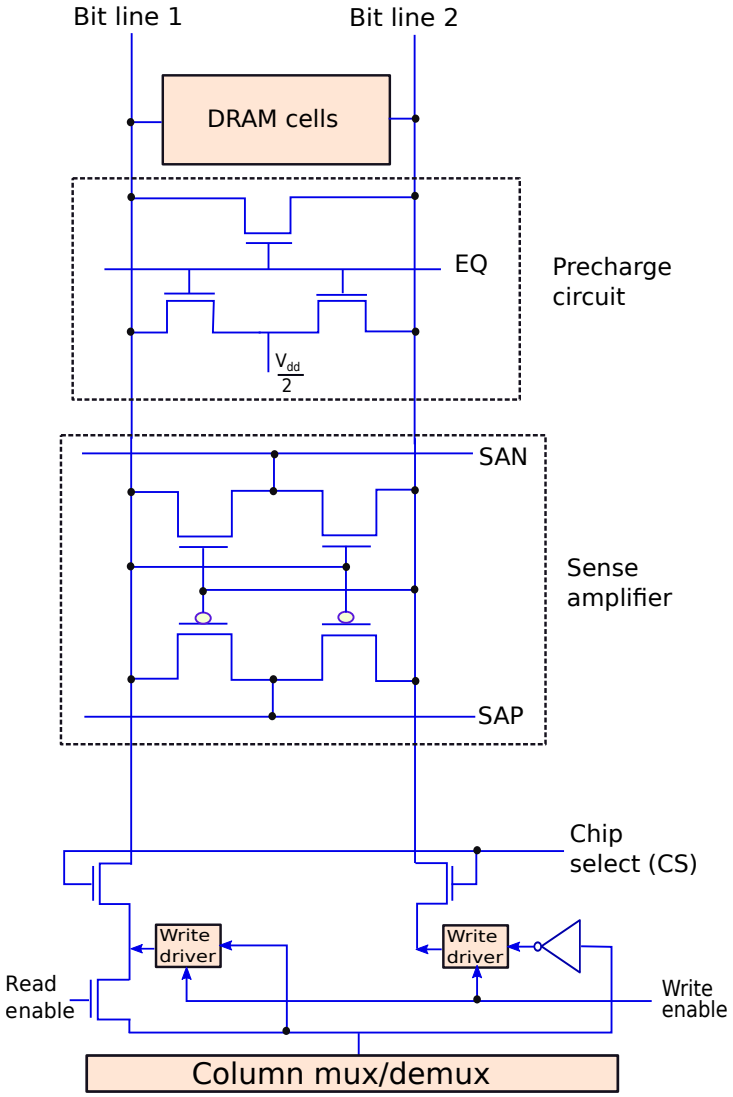

Figure 10.9: Layout of a part of a DRAM array with the precharge circuit, sense amplifier, and write drivers

Write Access

The process of writing in DRAMs is different as compared to SRAMs. We can divide the overall process into two broad stages. The first stage is the same as most of the read process where we precharge the bit lines, send the address to the row address decoder, enable the row, sense and restore all the values.

The actual write part is the second stage of this process. After all the cells in a row have been sensed and restored, the column address is sent to the column decoder. Then we enable the two write drivers (refer to Figure 10.9). It is assumed that the write drivers are strong enough to override the sense amplifiers. They set the state of the corresponding bit lines, the DRAM cell, and the corresponding sense amplifier. This finishes the write. The additional time required to do this is known as the write recovery time.

Refresh Operation

It is necessary to refresh the values of DRAM cells periodically (once every 32 to 64 ms), otherwise any charge stored across the capacitor will gradually leak out and the cell will lose its value. Thankfully, the refresh operation by itself is very simple – it is just the regular sense and restore operation. Recall that the sense and restore operations read the values of all the cells in a row, then use the sense amplifiers to set the values of the bit lines to either the maximum or minimum voltage. This process in effect refreshes the value that is stored in each cell by restoring the charge on the capacitor to the ideal value. Just in case some charge has leaked out, the corresponding capacitor gets fully charged after this operation. Hence, a refresh can be thought of as a dummy read operation. Note that we do not need to use the column decoder or enable the chip select line.

There are two types of refresh operations: burst and distributed. In the *burst* operation, we freeze the entire DRAM array and refresh all the rows one after the other. During this time, it is not possible for the DRAM array to process any requests. This is inefficient, hence, advanced processors use the *distributed* refresh mode. In this case, refresh accesses are interspersed with regular memory accesses. This is done to hide the overhead of refresh operations as much as possible. Moreover, it is possible to further optimize this process by not refreshing the rows that do not contain any valid data. Additionally, in modern DRAMs it is possible to slightly overshoot the maximum refresh interval without causing any correctness issues. This allows us to schedule critically important read requests.

10.1.4 A Computer System with DRAM Arrays

Since DRAM arrays have a very high storage density, they are typically used as off-chip memories. They can be used to store a large amount of data off chip and are thus ideally suited for a last level memory system. They are not particularly suitable for on-chip caches because SRAM arrays tend to be faster. There are several reasons for this.

1. A DRAM cell is very feeble. It has a single capacitor that needs to charge a very long bit line. In comparison, in an SRAM, the bit line is connected directly to either the ground or supply terminals via the transistors in the SRAM cell. As a result, it is possible to supply much more current and thus charge the bit lines more quickly.

2. In a DRAM, a read access is *destructive*. This means that we need to write the original value back to the cell that we read from. This requires additional time because in a DRAM array a read is actually a read and a write. This overhead is absent in an SRAM.

3. We need to spend some time doing a refresh on a compulsory basis, otherwise we run the risk of losing data.

Given these factors, it is almost always advisable to have a large off-chip DRAM memory, which is typically at the lowest level in the memory hierarchy. Recently, embedded DRAM (eDRAM) devices

have arrived where we can integrate DRAM memory into the same die as the processor or have a separate module within the same package. The main advantage of eDRAM devices is that they allow shorter and higher bandwidth connections between the LLC (last level cache) and the eDRAM memory.

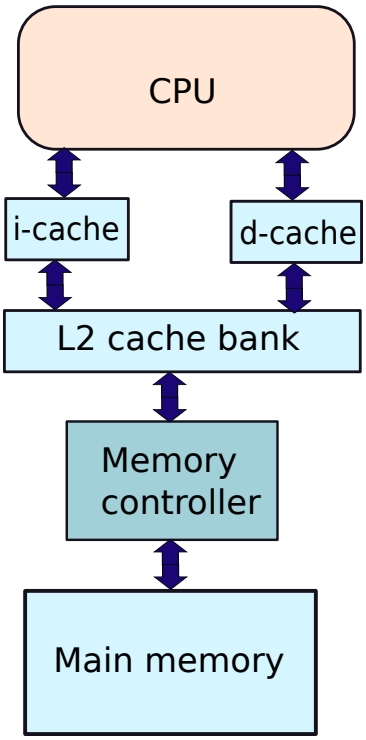

Figure 10.10: Processor + memory controller + DRAM

Generic Architecture

In this section, let us discuss a generic architecture for all kinds of DRAM devices that are used with modern processors (refer to Figure 10.10). The processor is connected to a memory hierarchy that consists of layers of caches of increasing sizes. The i-cache and the d-cache occupy the highest levels, then we have the L2 cache, and some processors might optionally have an L3 or L4 cache. The last layer of caches in a processor is known as the last level cache – abbreviated as *LLC*. The layer below the LLC is the off-chip memory, which is made of DRAM arrays. In a chip we have multiple memory controllers, which act as mediators between the LLC and the DRAM memory. If we have a miss in the LLC, then a request is sent to a memory controller, whose job is to interact with the DRAM arrays and complete the memory access.

Let us look at this in some more detail. A processor can have many memory controllers. The physical address space needs to be partitioned across these memory controllers. For example, we can use the MSB bits of the memory address. Consider a system with two memory controllers. If the MSB is 0, we access memory controller 0, else if it is 1, we access memory controller 1. Each memory controller is connected to a set of DRAM arrays via a set of copper wires. These sets of wires are known as *channels*. A channel is typically 32-128 bits wide. Channel widths are getting shorter with time mainly because if we are sending data at a high frequency, it is hard to keep the data across the different copper wires in the channel synchronized. The channels are connected to a set of printed circuit boards (PCBs) that contain DRAM chips. These PCBs are known as DIMMs (dual inline memory modules). The picture of

a DIMM is one of the most recognizable images for DRAMs. It is shown in Figure 10.11. Note that both sides of a DIMM have DRAM chips. The DIMMs are inserted into the motherboard, which has dedicated slots for them. Refer to Figure 10.12 that shows a motherboard having multiple DIMMs installed in its slots. Installing a DIMM is as simple as aligning the DIMM with the slot and then pressing it such that it fits snugly in the slot. Many desktops and servers are often sold with a few empty DIMM slots such that if later on there is a need, the user can buy new DIMMs and install them. This will increase the memory capacity. If some DIMMs develop faults, they can be replaced as well.

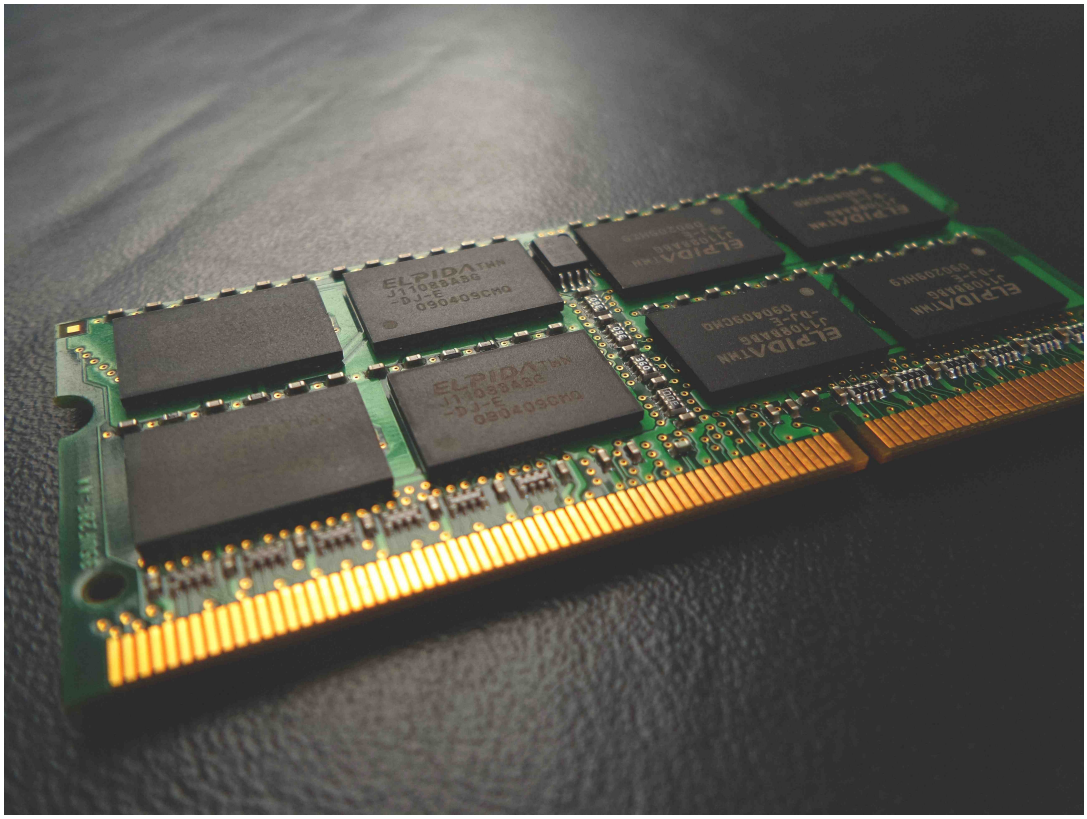

Figure 10.11: Photograph of a DIMM (Photo by Franck V. on Unsplash)

Each DIMM contains a set of DRAM chips. We typically divide a DIMM into multiple ranks (typically 1 to 4), where each *rank* contains a set of DRAM chips that execute in lockstep. Moreover, it is assumed that the chips in a rank are equidistant from the memory controller: it takes the same amount of time for signals to reach all the chips from the memory controller. Typically, the memory controller issues a command to a given rank. All the DRAM chips that are a part of the rank work in lock-step to execute the command. The main advantage of grouping DRAM chips together is to provide a high bandwidth memory. For example, if we need to supply 64 bits every cycle, then it makes sense to create a rank of 16 chips, where each chip supplies 4 bits. This keeps each individual DRAM device small and power efficient.

Subsequently, each rank has multiple *banks* (grouped into bank groups in the DDR4 protocol). A bank is a set of arrays within a DRAM chip that operates independently with respect to other banks on the same chip. A bank typically contains multiple arrays that cannot be independently addressed.

The arrays within each bank work in synchrony. For example, if we have 4 arrays in a bank, we access the same row and column in each array while performing a bank access. We read 4 bits in parallel.

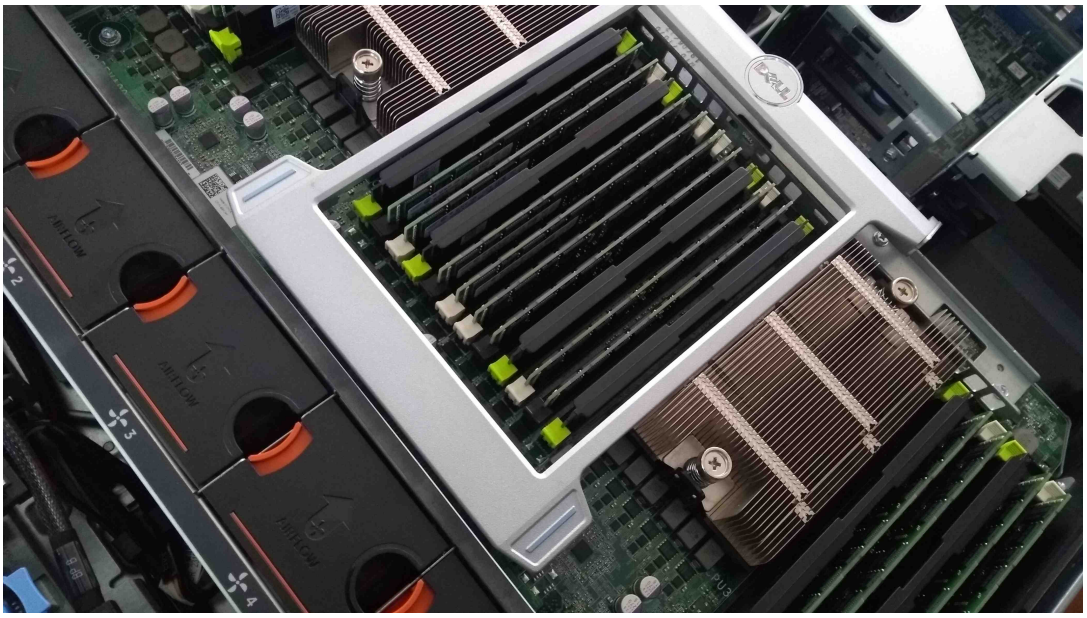

Figure 10.12: Photograph of a motherboard with DIMM slots (Photo by Stef Westheim on Unsplash)

This is conceptually the same as assuming that we have one large array where each cell or each column is 4 bits wide. Using more arrays increases the bandwidth of a DRAM device because we can read more bits in parallel. In a DRAM chip all the banks have the same number of arrays.

This is typically specified as follows. When we say that we have a x4 DRAM, this means that we have 4 arrays in a bank, and we read and write 4 bits at a time. An xN DRAM has exactly N arrays in a bank. As of 2020, x16 to x128 DRAMs are there in the market. x64 and x128 configurations are typically only present in 3D DRAMs (we shall study them later in Section10.5.6).

Each array is a matrix of DRAM cells. We first access the row and then the column to read or write a single bit. The entire hierarchy of structures in a DRAM is as follows: channel → DIMM → rank → chip or device → bank → array → row → column. This is shown in Figure 10.13.

Topology

There are several methods to connect a memory controller to memory modules (DIMMs). The simplest possible arrangement is that we connect one memory controller to one DIMM using a dedicated channel. Typically, on a channel, we send four kinds of information: address, data, command, and chip-select. The first three are self explanatory, the *chip select signal* is used to enable a specific rank of devices. If we have 4 ranks in a DIMM, then we need a 2-bit chip select signal to select the specific rank. The address and command buses are unidirectional, and so is the chip select bus. However, the data bus is bidirectional because data can flow either from the processor to the memory or in the reverse direction. We can either use separate address and command buses, or have a single bus to carry the information for both memory addresses and commands. It is possible to fuse them because we typically send the address and commands at different points of time.

It is additionally possible to connect multiple DIMMs per channel. There are several advantages of doing this.

1. To increase the bandwidth, we can split the channel across the DIMMs. For example, if we have a 128-bit wide data bus, we can split it into two equal halves across two DIMMs that read or

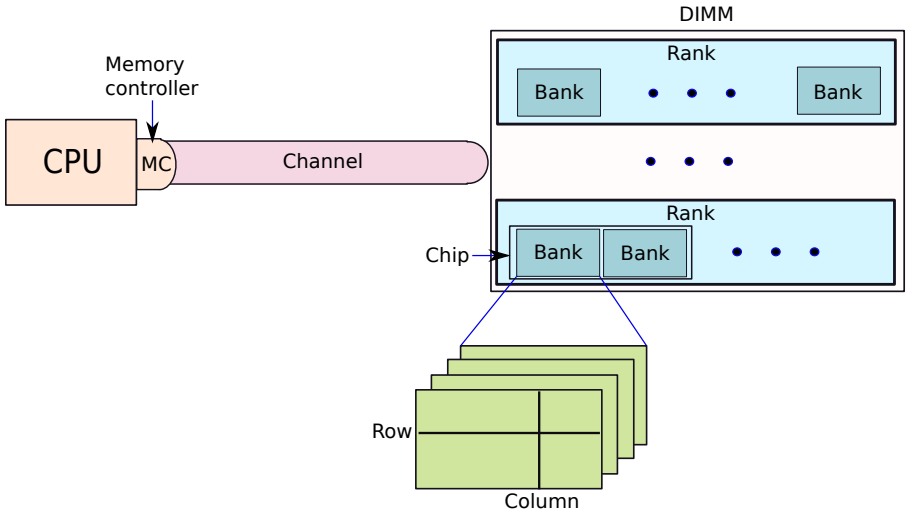

Figure 10.13: Hierarchy of elements in DRAM memory

write 64 bits at a time. Both the DRAMs can operate in lockstep. This will effectively double the bandwidth of the channel assuming the maximum amount of data that we can transfer from each DIMM per cycle is 64 bits.

2. We can also connect different DIMMs that are not similar. For example, if our data bus is 64 bits wide, we can use one DIMM that has a 64-bit interface and another DIMM that has a 32-bit interface. Such topologies can be used to support legacy systems that use older technologies. In this case, we cannot use both the DIMMs simultaneously. We need to interleave their accesses and the achieved bandwidth is the maximum of the bandwidths of the individual DIMMs.

3. If the channel is not being kept busy all the time because the DIMMs take time to perform their accesses, we can use this time to access other DIMMs connected to the same channel. The total bandwidth in this case depends on the degree to which we can interleave the accesses. It can theoretically scale with the number of DIMMs per channel till we are limited by the channel capacity.

Now, let us see how we connect the address, data, command, and chip select buses to the DRAM chips within each DIMM. One of the most common topologies is shown in Figure 10.14. In this topology, the DRAM chips are arranged as a 2D matrix. DRAM chips in the same rank are the columns, and corresponding chips across ranks form the rows. The address/command bus is routed to every DRAM chip, and the chip select lines are connected to each rank separately. The latter are used to either enable or disable the entire rank in one go. The data bus is split into four *lanes* (one-fourth the width of the data bus). Each of these lanes is connected to a row of banks across the ranks. Only one of the ranks can use the lanes of the data bus at a given point in time, and thus this ensures that we can read 32 to 128 bits in parallel from all the banks in the rank.

Important Point 18
There is a very important point to note here. Note that the address/command bus is connected to all the banks across the ranks. In this case, it is connected to 16 banks. Whereas, each data bus lane is connected to only 4 banks. This means that the capacitive loading *on the address/command bus*

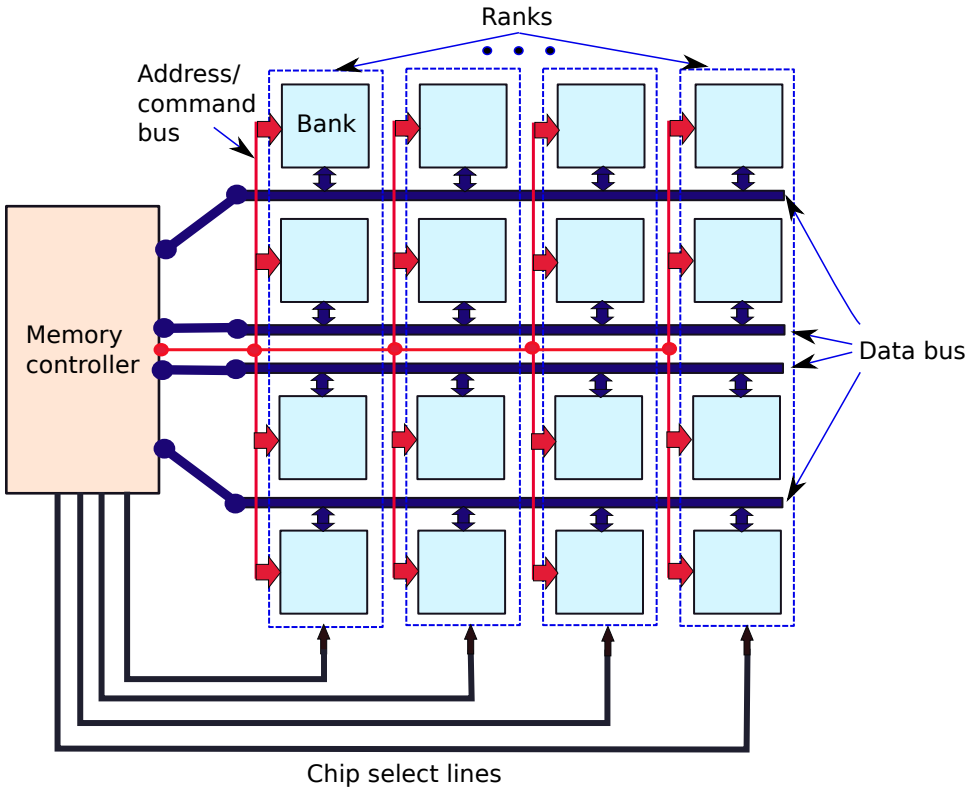

Figure 10.14: Organization of memory banks

is 4 times more than that of a data bus lane. As a result, its RC delay is more, and thus it is a slower bus. In comparison, the data bus is faster, because each of its lanes is connected to a fewer number of devices. It can thus sustain a higher data transfer frequency. This is a **crucial insight**, *which we shall use while designing DDR (double data rate) memory later on.*

10.2 Design Space of DRAMs

10.2.1 DRAM Access Protocols

DRAM memories are passive memories. There are no computing elements embedded inside the memory. As a result, the entire job of scheduling all the accesses, ensuring that there are no conflicts, and providing fairness is left to the memory controller. The memory controller needs to be aware of the timing details of the memory devices, and then it needs to schedule all the requests that it gets from the cores and the cache banks. Previously, the memory controller used to reside in a separate chip on the motherboard; however, nowadays, the memory controller is placed on the same die as the CPUs and caches. It needs to be aware of the details of the devices that are attached to the memory channels, for example, their timing and the commands that they support. Let us look at DRAM devices now in some more details.

Over the last two decades, DRAM technology has been improving at a significant pace. Initially, they were asynchronous devices. The advantage was that they could be connected with any memory

controller because there were no strict constraints on the timing. However, this also introduced additional complexities in orchestrating the data transfer, and it also made buffering commands difficult. Hence, gradually DRAM technology has moved towards synchronous devices where the DRAM devices have their own clocks, and there is some degree of synchrony between the DRAM clock and the clock of the core. This has paved the way for modern synchronous DRAM access protocols that are fast and reliable. In the next few sections, let us look at the evolution of DRAM access protocols over the last two decades.

Asynchronous Transfer

The first memory transfer protocols were asynchronous protocols, where the CPU and the memory did not share a clock. In an asynchronous mechanism there is no common time base, hence, the sender needs to let the receiver know when it can read the data.

Let us first explain a simple scheme. Let us have two buses to transfer data: one each way (simplex mode). The two buses carry two signals: a strobe signal (DQS) and the data signal (DQ). The reason we need a strobe signal is as follows. Whenever the sender sends data, the receiver needs to read the data and store it in a latch. Such latches are typically edge triggered (read data in at a clock transition). Since the sender and receiver do not share a clock, synchronization is an issue. There needs to be a mechanism for the sender to let the receiver know when it can read the data. This is where the strobe signal is used. The receiver monitors the strobe, and whenever there is a transition in its voltage level, it reads the data bus (DQ signal).

If we assume that a transition in the strobe signal happens at $t = 0$, then no transition is allowed in the data bus in the time interval $[-t_{setup}, +t_{hold}]$. The data signal (DQ) needs to be steady in this time window, otherwise we shall have a phenomenon called metastability that leads to unpredictable behavior. t_{setup} and t_{hold} represent the setup time and hold time respectively. Refer to Figure 10.15 for the timing diagram. It is not necessary for strobe signals to always convey information via transitions, many times the level is also used to convey some information such as whether a given unit is enabled or disabled. Note that we can latch data at either the falling edge of the clock (as shown in Figure 10.15(b)) or the rising edge of the clock.

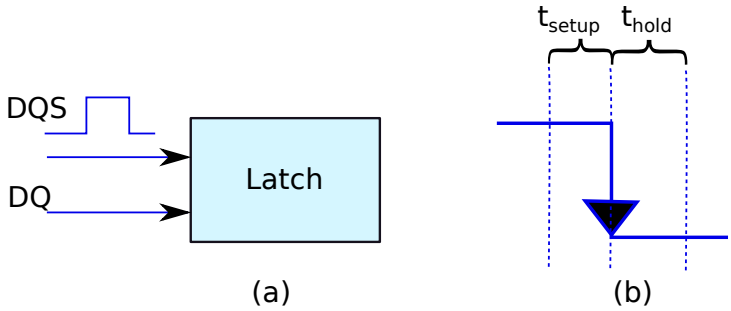

Figure 10.15: Storing a value in a latch with a strobe input

Let us now slightly complicate the scheme and consider two half-duplex buses (only one side can send at a time) between the memory controller (MC) and the DRAM. One of these buses is an *address bus* to carry row or column addresses and the other is a *data bus*, also called DQ. In an asynchronous memory, we have two strobe signals: $\overline{RAS}$ and $\overline{CAS}$. RAS stands for *row address strobe* and CAS stands for *column address strobe*. These are active low signals and are said to be asserted when the voltage is a logical 0. To indicate this fact, we use the symbols $\overline{RAS}$ and $\overline{CAS}$ in our diagrams. They mean that the signals are said to be asserted when the voltage is equal to a logical 0. This can be slightly confusing. Readers should make a note of this. They need to understand that the signals that are being transmitted are $\overline{RAS}$ and $\overline{CAS}$, which are said to be asserted or active when they are equal to a logical 0.

The first action that the memory controller needs to perform is that it needs to activate the DRAM row. This is done by sending the row address on the address bus, and then after some time asserting the $\overline{RAS}$ signal (setting it to 0). The reason that we do this is as follows. We want the data on the address bus to be stable before the device starts reading it. The device will start reading it when it sees the $1 \rightarrow 0$ transition of the $\overline{RAS}$ signal (refer to Figure 10.16). Once the DRAM device sees the row address and the $\overline{RAS}$ signal set to 0, it activates the row decoder, and then activates the row.

Subsequently, the memory controller sets the $\overline{CAS}$ signal to 0, and then after some time sends the column address on the address bus. Along with this, it can send one bit indicating if it wants to read or write. This activates the column decoder, which then prepares the column for reading or writing.

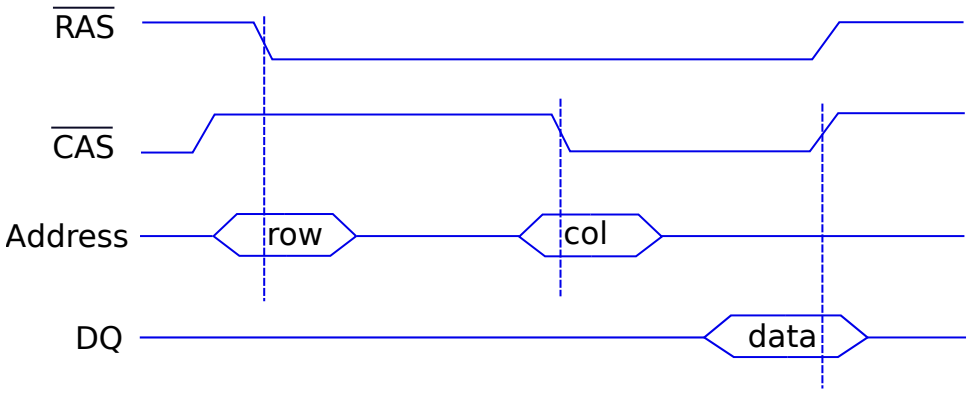

Figure 10.16: Timing diagram of an asynchronous DRAM

Consider a read access. The DQ bus is set to the value of the column by the DRAM device when the data is ready to be sent to the memory controller. After that the device sets $\overline{CAS}$ to 1, which indicates to the memory controller that it can start reading the DQ bus.

In the case of a write access, after sending the column address, the memory controller sends the data bit, which is then written to the device. In some protocols, after a write is done, an acknowledgement is sent to the CPU. The timing of the write is also managed by the strobe signals.

Fast Page Mode (FPM)

The basic asynchronous DRAM protocol is inefficient because we need to send the row and column address for every data transfer. This can be improved by operating the DRAM devices in *fast page mode*. In this mode, an entire row of data (page) is stored in the sense amplifiers, and then read over subsequent cycles. It is not necessary to send a separate row address for reading a different set of columns. This process is shown in Figure 10.17. The same holds for writes.

We first send the row address by asserting (setting to 0 in this case) the $\overline{RAS}$ signal. The DRAM banks read the entire row (page) and store the contents in the sense amplifiers. Subsequently, we send a sequence of column addresses to the DRAM banks. Each bank then chooses the right column using the column multiplexers and sends the data back. Since the entire page is stored in the sense amplifiers, it is not necessary to send the row address again if we intend to read more data from the same row, which is often the case because we read an entire 64-byte block at a time. In this case, we just need to send subsequent column addresses, and the banks can quickly transmit the data. Since this process is asynchronous, we need a strobe signal. We use the $\overline{CAS}$ signal to provide the timing for this process.

The key insight in this scheme is that we are using the sense amplifiers as a buffer. A typical FPM device can have 1024 columns per row, where each column is 16 bits wide. We can realize such a design by having a total of 16 arrays in each rank. We first read the entire row and store it in the sense amplifiers. We then select one column at a time, and in each array we read a single bit. The entire rank

can thus provide 16 bits at a time, and we do not need to incur the overhead of row activations several times. We activate the row only once, and then read out all the columns that we are interested in.

We shall henceforth not discuss how we handle writes because they are handled in a very similar manner.

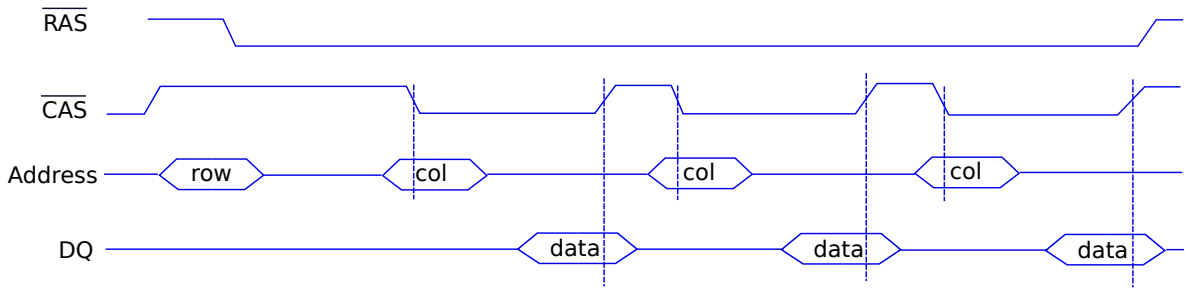

Figure 10.17: Timing diagram of a Fast Page Mode (FPM) DRAM

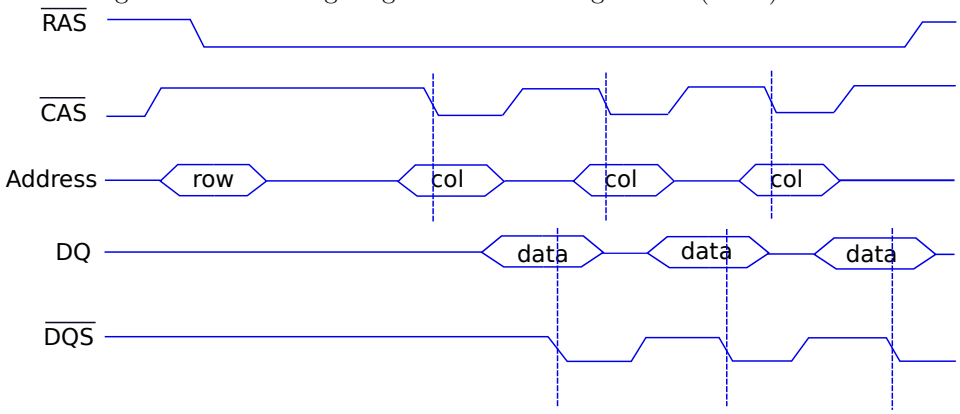

Figure 10.18: Timing diagram of an Extended Data-Out (EDO) DRAM

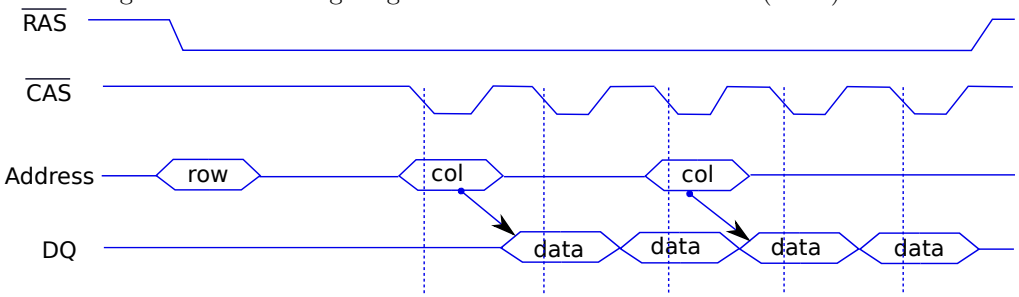

Figure 10.19: Timing diagram of a Burst Extended Data-Out (BEDO) DRAM

Extended Data Out (EDO)

In the FPM protocol, we send the column address, then we wait for the data to be read, and then we send the next column address, and so on. This process can be optimized further. We can send a column address, internally buffer the data that has been read, and while that data is being transferred, send the address of the next column. This process is shown in Figure 10.18.

This will however require another strobe signal ($\overline{DQS}$) as shown in the figure. This strobe signal indicates to the memory controller that the data is available on the data bus. If we compare Figures 10.18

and 10.17, we observe that the throughput has increased because of a reduced column-to-column delay.

Burst Extended Data Out (BEDO)

Most of the time, we do not access DRAM devices with random addresses. We typically wish to read consecutive columns because we transfer 64-byte blocks to the CPU. This takes multiple cycles, where we issue reads to consecutive columns by sending separate column addresses. It is not necessary to send the column addresses for consecutive sets of bytes in a 64-byte block. They can be generated internally.

In a certain sense, the DRAM device prefetches data words (groups of data bits) and sends them to the CPU via the memory controller. Since most accesses read contiguous sequences of data words, this access pattern is very common in practice. Hence, as shown in Figure 10.19, after one column address is sent, we can generate the next k column addresses internally, read those columns, and send their data over the data bus. We do not have to waste time in sending column addresses separately. In Figure 10.19, our prefetch length is 2 (read two columns after sending a single column address). We do not need a separate strobe signal, we can use the $\overline{CAS}$ signal to provide the timing.

The *prefetch length* or *prefetch width* is the number of bits we read in one go from each DRAM array. For example, if we read 8 bits in one go, the prefetch length is 8. In a clocked bus, we need 8 bus cycles to send these 8 bits (1 bit per cycle).

Synchronous DRAM

Even though asynchronous memory devices became very efficient, they still had numerous drawbacks. In general, maintaining timing is difficult, particularly in complex DRAM systems. As a result, almost all memory devices today use synchronous DRAM (*SDRAM*). In such devices, the memory controller and the DRAM devices use a common time base, which means that they use the same clock. All the latencies are specified in terms of clock signals, and all the messages are aligned with respect to clock boundaries. This simplifies the communication to a large extent and makes it possible to create elaborate and scalable protocols. Some other advantages of synchronous communication are as follows.

1. A synchronous system is simple to design and verify.

2. In asynchronous memory, the $\overline{RAS}$ and $\overline{CAS}$ signals directly control the banks. It is not possible to add additional programmable logic within the DRAM devices. However, with synchronous memory, it is possible to simply send commands to the devices, and let the devices implement them in different ways.

3. SDRAM devices are more configurable. For example, it is possible to switch the mode of an SDRAM device, and also dynamically change its prefetch length (typical values: 1, 2, 4, or 8).

4. SDRAM devices contain multiple banks. It is possible to send different commands to different banks. For example, it is possible to pipeline commands where we can read one bank while precharging another bank.

It is possible that there is a phase difference between the clock of the memory controller and the internal clock of the DRAM device. To ensure clock synchronization, most SDRAM devices have a DLL (delay locked loop) circuit within them. This ensures that the clock of the DRAM devices and the memory controller remain synchronized and the phase difference is reduced to a minimum. The clock of the memory controller can either be recovered from transitions in the data or from a dedicated strobe signal sent by the memory controller.

Figure 10.20 shows the timing diagram for a typical SDRAM device. We have four buses: CLK (clock), $Command$, $Address$, and DQ (data). The commands and the addresses are latched into the SRAM device at the rising edge of the clock.

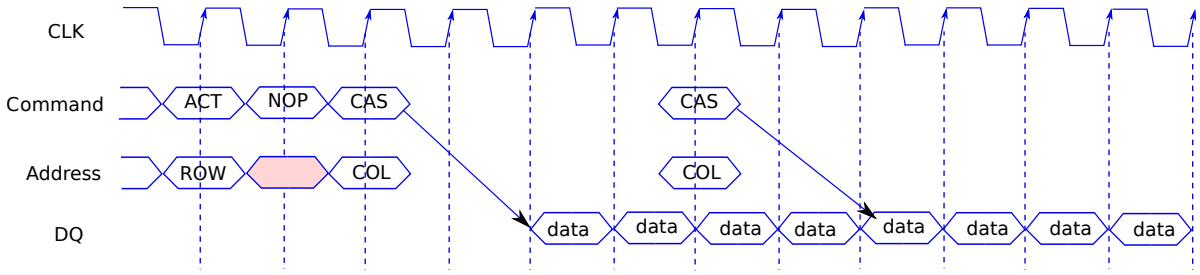

Figure 10.20: Timing diagram of synchronous DRAM memory

We first activate the row by sending the row activate ACT command along with the row address on the address bus. Subsequently, we assume that it takes one cycle to activate the row, then we send the column activate command CAS, along with the column address. After sending the column address, we wait for one clock cycle, then the memory controller starts receiving data from the DRAM device. In the case of a read transaction, the data transmission starts at the rising edge of the clock. In this case, the prefetch length is 4. Similar to BEDO DRAM, we can send additional column addresses to read other data words from the row (opened page) in subsequent clock cycles. The advantage here is that we do not need to send the row address and activate the row again.

Note two things. We shall sometimes issue the NOP command indicating that we are not issuing any command; this will be done whenever it is necessary to show inactivity on the command bus. Second, a shaded hexagon on the data or address bus means that we do not care about the data or address being sent.

10.2.2 DDR Generations and Timing

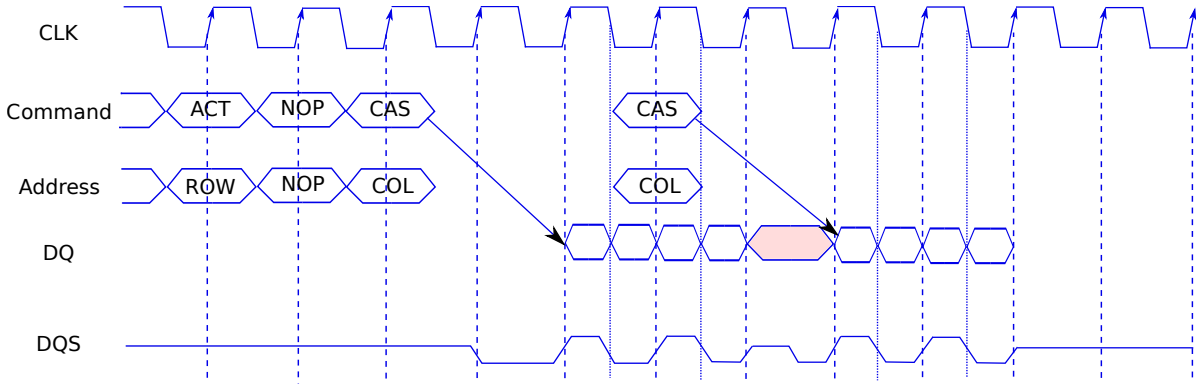

Figure 10.21: Timing diagram of DDR memory

Recall that we had argued that the command and address buses have a much higher capacitive loading, and are consequently much slower because they are connected to all the DRAM banks. In comparison, the data buses are connected to far fewer banks, and thus the loading on each bus is significantly lower (refer to Figure 10.14). Given that the data bus is expected to be much faster than the address or command bus, we can use this fact to further speed up our memory access protocol.

Because of the fundamental asymmetry in the speeds of the data bus and the address and command buses, double data rate memory (DDR memory) was developed. In this memory, the data bus runs at

twice the speed of the address and command buses.

Figure 10.21 shows the timing diagram of a DDR memory device. The clock, command, and address buses have the same functions. The major difference is that we transmit data at twice the rate. Additionally, we have a new signal, a data strobe signal *DQS*, that is sent along with the data. This is because we can have minor clock skews, and for the receiver to read the data correctly, it needs to use the *DQS* signal for clock synchronization. This method of transmission is known as *source synchronous transmission*. As we can see from Figure 10.21 this signal can take three values: logical 0, logical 1, and an intermediate voltage that represents the undefined state. Furthermore, its voltage can be set by either the memory controller or the DRAM device depending upon the direction of data transmission.

Let us now study some subtle aspects of the design of DDR memory. We see a one-cycle bubble between the two data bursts. In this case, we assume that the second burst of data in the figure is provided by another rank of DRAM devices whose corresponding row has been activated in the past. To switch the rank it takes one cycle because the new rank has to re-synchronize its clock with the data strobe.

When we are using this protocol to write data to the DRAM devices, the memory controller sends data such that transitions of the strobe signal happen at the middle of transmitting each bit. This makes it easier for the DRAM device to latch the data. However, it increases the complexity of the circuit at the end of the memory controller. This is acceptable because we want DRAM devices to be simple, and we want to migrate the complexity to the memory controller where we can afford it.

It is important to understand that while reading data, this is not necessary. This is because while reading data, the DRAM device is the sender. It is much easier for it to transmit data at clock boundaries. The memory controller can then recover the clock from the strobe and latch the data at the correct time instants. This requires more circuitry but is an acceptable overhead at the side of the memory controller.

Note that the data bus in this case runs in half-duplex mode, Which means that it does not allow simultaneous transmission of data in both directions. Thu,s DDR memory is the most efficient when we either have only reads, or only writes. To remedy this problem, QDR (quad data rate) memory was proposed that has two data buses: one read bus and one write bus. This allows us to effectively achieve a higher bandwidth by interleaving reads and writes.

DDRX Memory

After the basic DDR memory was proposed, no fundamental changes to the paradigm were made. Instead, there were subsequent improvements in the process and signaling technologies to realize faster memories. Thus, we have several DDR generations: DDR2 to DDR4. They are collectively known as DDRX technologies.

Technology	Bus clock speed	Prefetch length	Transfer rate (MT/s)	Voltage	Maximum DIMM size
DDR	100-200 MHz	2	200-400	2.5-2.6 V	1 GB
DDR2	200-400 MHz	4	400-800	1.8 V	4 GB
DDR3	400-1066 MHz	8	800-2133	1.35/1.5 V	16 GB
DDR4	800-2133 MHz	8	1600-4266	1.2 V	64 GB

Table 10.1: Description of the different DDR technologies

Table 10.1 describes the different DDR technologies. In the list of technologies, DDR is the oldest standard, and DDR4 is the newest standard (as of March 2020). As of March 2020, DDR5 is still in the process of standardization. Each DDR device has an internal clock that runs at a much lower clock speed as compared to the bus frequency. For example, in DDR4, the internal clock can vary from 200 MHz to 533 MHz (increasing in units of $33\frac{1}{3}$ MHz). However, the bus frequency varies from 800 MHz to 2133 MHz.

Note that the bus frequency is always more than the devices' internal clock frequency. This is because buses have gotten faster over the years, whereas DRAM devices have not sped up at that rate. Consider DDR once again, for a bus frequency of 800 MHz, the internal clock is 200 MHz. There is thus a rate mismatch, which can only be equalized if the internal bus width of the DRAM devices is more. Given that we are transmitting data at both the edges of the clock, the DRAM devices need to provide $2 \times 800/200 = 8$ times more data per cycle. This means that, internally, the rank needs to produce data at 8 times the rate per cycle by parallelizing the read/write operations among the arrays and by reading more data per clock cycle.

Let us do the math for this example. In all DDR technologies, the channel width is 64 bits (72 bits with ECC), and we send 64 bits in parallel in each half-cycle. This set of 64 bits is known as a *memory word* and each half-cycle is also known as a *beat*. If the internal frequency is 200 MHz and the bus frequency is 800 MHz, then as argued before, DRAM devices need to produce more data per cycle. Let us consider 8 beats, which is equal to the length of a DRAM device's clock cycle (internal cycle). In 8 beats, we need to transmit 512 bits. If we have 64 arrays in a rank, then we need to read 8 bits in every internal cycle. This can be done by prefetching columns as we had discussed before in the case of BEDO DRAM. The number of bits we prefetch per internal cycle is known as the *prefetch length* in the case of synchronous DRAM. This needs to be equal to 512/64, which is equal to 8.

The prefetch length scales with the ratio of the bus frequency to the internal DIMM frequency. It was 2 for DDR, 4 for DDR2, 8 for DDR3 and DDR4. For example, if the prefetch length is 8, then we will require 8 beats to transmit all the data that has been read from the DRAM devices. This means that the minimum data transfer size in DDR4 is 64 bytes (8×64 bits). A sequence of bits being transmitted is known as a *burst*. In this case, the minimum burst length is 8.

This unfortunately has negative consequences. This stops us from transmitting data that is less than 64 bytes in DDR3 and DDR4. Hence, in DDR3 the *burst chop* mode was introduced. It is possible to program the DRAM devices such that they disregard the second half of an 8-beat burst. We can thus effectively reduce the minimum burst length to 4 beats, even though we are not sending useful data in place of the disregarded beats.

The next column in Table 10.1 shows the transfer rate of the DRAM device measured in millions of transfers per second (MT/s). A *transfer* is defined as the transfer of a 64-bit data packet (equal to the channel width) from the memory controller to DRAM or vice versa. In a DDR memory, the number of transfers per second is equal to twice the bus frequency. For example, if the bus frequency is 400 MHz, then we perform 800 million transfers per second (MT/s) because of double data rate transmission. The standard nomenclature that we use to label DRAM devices is of the form $(\langle Technology \rangle - \langle Transfer\ rate \rangle)$. For example, a DDR3-1600 technology means that we are using the DDR3 technology, and we perform 1600 million transfers per second.

The next column shows the transmission voltage. It has steadily decreased from 2.6 V to 1.2 V. As we lower the voltage, we also reduce the time it takes to transmit a message. However, the susceptibility to noise and crosstalk increases. These need to be managed with technological innovations. Subsequent DDR generations are expected to reduce the supply voltage even further.

Furthermore, due to increased miniaturization and improvements in fabrication technology, the density of bits is increasing. The maximum capacity of a DIMM has also been steadily increasing from 1 GB (DDR) to 64 GB (DDR4).

10.2.3 Buffered DIMMs

An important shortcoming of conventional memory systems is that there is a trade-off between the frequency of the memory bus, the frequency of the DIMMs, and the number of DIMMs that we can connect to each channel. As we connect more DIMMs to the channel, the capacitive loading on the address, command and data buses increases, which adversely impacts their RC delay. Recall that the RC delay or the time constant is the time that it takes to charge the bus to 63% percent of its final value. This means that with more connected devices, it takes more time to effect voltage transitions on

the bus, and this directly places limits on the bus frequency. For the different DDR generations, this is what is happening. For example, in DDR2 where the bus frequency was 400 MHz, we could connect 8 devices to each memory channel, whereas in an 800 MHz DDR3 bus, we can only connect 2 devices. Such trade-offs limit DRAM scaling to a large extent.

As a result, it became necessary to think of new bus technologies that can circumvent such limitations. On the flip side, there are very strong business reasons to keep DIMMs unmodified. The DRAM business is very competitive; therefore, vendors have been averse to adding additional circuitry to the devices. In addition to that, it is necessary for all memory controllers and the RAM chips to be compliant with the DDR standards. Hence, the space for innovation is very restricted.

Keeping all of these constraints in mind, a set of buffered memories were proposed. These classes of DIMMs contain a buffer with every DIMM chip that buffers either the data or the control messages. The effect of such buffers is that it reduces the net capacitive loading on the memory channel. It thus allows for faster data transfer, and we can connect more devices to a channel. There are many classes of buffered memories. We shall discuss two of the most popular classes in this section: fully buffered DIMMs and registered memory.

Fully Buffered DIMMs (FB-DIMMs)

In the 2000-2005 time frame, there was an increasing realization that traditional DDR based memory technologies will not scale. We are limited by the frequency of the bus, and since these are multidrop buses (many DIMMs are connected), the capacitive loading on the command and address buses is a key limiting factor. There was a need to create large server-based systems that could sustain large memories and provide robust communication. At that point of time, an audacious attempt was made to completely re-architect the memory system. One such attempt was the proposal to create Fully Buffered DIMMs (FB-DIMMs). Given that this idea involved significant changes to the memory system, DIMMs, memory controllers, and the motherboards, the industry was not very enthusiastic, and thus this technology is not very popular as of 2020. However, from an educational perspective, this technology has immense theoretical value and provides insights into what it takes to create robust and scalable DRAM based memory systems.

AMBs

The crux of the idea behind FB-DIMMs was a custom chip called the Advanced Memory Buffer (AMB) that was supposed to be a part of each DIMM. A representative diagram of the system is shown in Figure 10.22. The CPU has a dedicated FB-DIMM memory controller, which is connected to the AMB of the first DIMM chip. The AMB is connected to all the DRAM devices within the DIMM. The AMB can use traditional DDR protocols to communicate with the DRAM devices. They need not be aware of the fact that a different protocol is being used.

The AMB in the first DIMM chip is connected with the memory controller via a set of lanes. Akin to high-speed I/O protocols such as USB and PCI-X, each lane is a high-speed serial bus, whose timing is independent of other parallel lanes. The memory controller sends data and commands to the first AMB via a set of lanes known as the southbound lanes. The AMB sends responses to the memory controller via another set of lanes known as the northbound lanes. The advantage of using lanes is as follows.

1. Since each lane is a high-speed serial bus, its timing need not be synchronized with other lanes, and thus we can raise the transmission frequency significantly.

2. The communication architecture is more immune to failures. Assume that a given lane fails, or the timing on a lane changes due to ageing, the system remains unaffected. In the first case, we can simply disregard the lane and transmit on the remaining lanes that are functional. In the second case, since the transmission on the different lanes happens independently, this will not lead to a failure.

The AMBs are connected in a chain using point-to-point links. The role of each AMB is as follows.

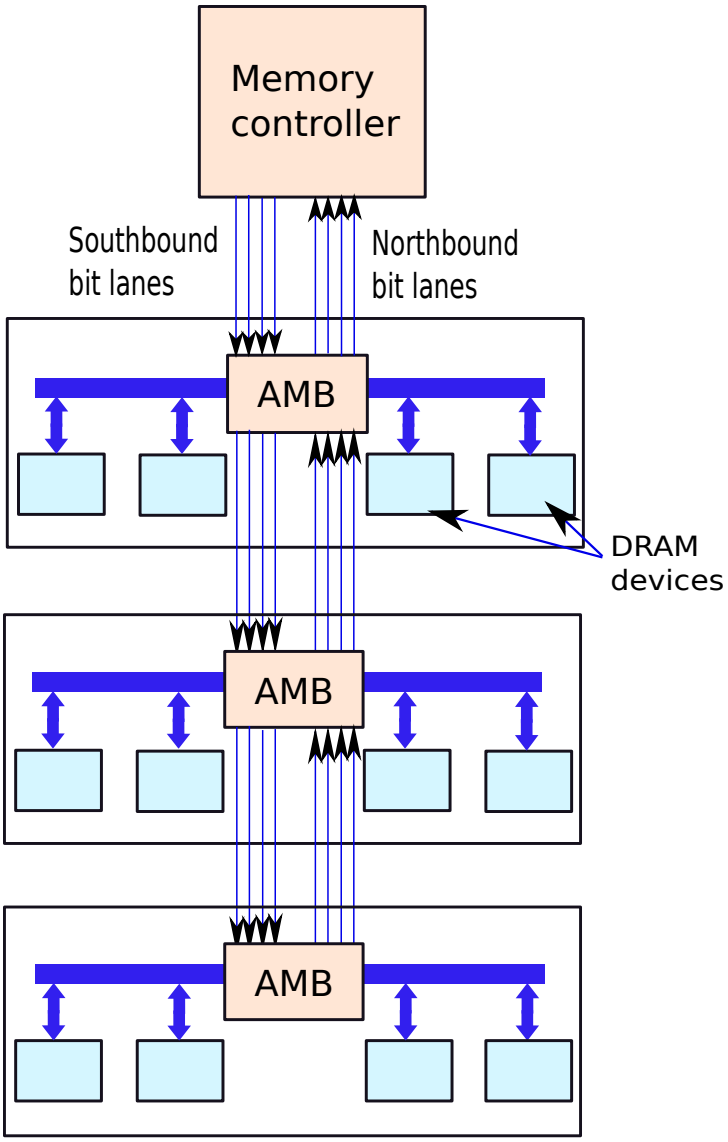

Figure 10.22: An FB-DIMM system

1. Receive data from southbound lanes (from the memory controller).

2. If the data is meant for the DIMM associated with the AMB, then reconstitute the packet by reading data from the serial bit lanes, and send it to the constituent DRAM devices. If the data is meant for an AMB downstream, then forward it to the next AMB in the chain.

3. Get data from northbound lanes, and send it towards the memory controller (northward).

A few of the advantages are obvious.

Scalability This is a scalable system, because we can add a large number of DIMMs.

High-speed The loading on buses is minimized and this ensures that we can have high-speed buses. Furthermore, since we use a set of serial bit lanes, they can use very high-speed signaling to transfer data as quickly as possible.

Reduced Pin Count The increasing number of pins associated with memory controllers was an issue because they needed to support many channels. There are limits to the pin count due to packaging issues. This protocol reduces the number of pins that are required in memory controllers.

Reliability Because we can use a variable number of bit lanes, and the loading per lane is deterministic, the overall reliability is enhanced.

The FB-DIMM technology was unfortunately not adapted at a large scale because of the complexity of the AMB. It needs to act as a router, serialize/deserialize data, buffer data, run the DDR protocols between itself and the DRAM devices, and monitor the reliability of transmission. Instead of being relatively passive devices, which can be produced in bulk, FB-DIMMs are active devices with elaborate AMBs. This increased the cost. However, FB-DIMMs are still attractive solutions for large servers.

Let us now discuss some of the specific technologies that are used in FB-DIMMs.

High Speed Transmission

Normally, we have 10 bit lanes in the southbound channel and 14 bit lanes in the northbound channel. We have more bit lanes in the northbound channel because they are used to transmit data for read operations, which are often on the critical path. Because all of these bit lanes are high-speed serial buses, we can afford to transmit data at a much higher clock rate. On each lane, we can transport 12 bits per DRAM clock cycle. This means that on the southbound channel with 10 bit lanes, we can transmit 120 bits in one DRAM clock cycle. On the northbound channel, we can transmit 168 bits in one DRAM clock cycle. This includes status bits, commands, and bits to perform error correction. The FB-DIMM protocol uses the CRC (Cyclic Redundancy Check) error detection and correction scheme that is particularly useful for detecting a burst of errors.

Let us compare this approach with a DDRX protocol. In such a protocol, we can transmit 144 bits in one DRAM cycle, assuming that the bus is 72 bits wide (64 bits for data and 8 bits for error correction bits). With FB-DIMMs, we can send one write command with 64 bits of data and 8 error correction bits every cycle on the southbound channel. On the northbound channel, we can simultaneously transmit 144 bits of read data inclusive of error correction bits. Thus, the total amount of useful data that we can transfer between the memory controller and the FB-DIMMs is 144+72 = 216 bits. This is one and half times the bandwidth of a DDRX channel. Hence, there is an advantage in terms of bandwidth.

Along with this, we can support more FB-DIMM devices per channel, and more memory channels per memory controller because the number of pins required by each channel is much lower in this case.

Resample and Resync

Recall that AMBs also forward commands, addresses, and data to adjacent AMBs. We can use two methods. The first method called *Resampling* is as follows. An AMB directly forwards the data to the next AMB on the chain. In this case, it is possible that there would be some skew between the signals being sent on the different bit lanes. The skews will tend to accumulate over several hops. Even though this scheme is fast, we can still end up having a large amount of skew between different bit lanes. Note that at the destination, we need to wait for the slowest signal to arrive. We can however be slightly lucky if over the long transmission across several AMBs, the skews get balanced. This is not uncommon; however, we still need to be prepared for the worst case.

The other method is to read the entire data frame sent on the bit lanes, remove all the skew, and then retransmit the entire data frame to the downstream AMB. This method is known as *resync*, and introduces more delay in the protocol. It is good for a network with large skews that are unbalanced across the bit lanes.

Bit Lane Steering

Reliability is a key advantage of FB-DIMMs. Note that reliability is a key requirement of servers that typically use a lot of DRAM memory. Faults can often develop while replacing DIMMs or due to temperature induced stresses.

Let us assume that at some point of the transmission, we find that we are having too many errors (detected with the CRC error detection code). We shall first attempt a channel reset, which means that both ends of the channel discard their state, and try to resynchronize themselves. However, if this is not successful, we need to conclude that one of the bit lanes has developed a fault. FB-DIMMs have various BIST (built-in self test) mechanisms that allow us to determine which bit lanes have developed faults.

We can then use the bit lane steering mechanism to use the rest of the lanes for the communication. This will have a minimal effect on the bandwidth of the bus, however it will increase the reliability significantly.

Summary of the Discussion on FB-DIMMs

Undoubtedly, FB-DIMMs incorporate many technological advances. Their most important advantages include reducing the pin count, tolerating a high amount of skew during transmission, reducing the capacitive load on the bus, and using bit lane steering to use only those lanes that are fault-free. However, any revolutionary technology is still a slave of market economics, and if significant changes need to be made to the memory controller, DIMMs, and the channels on the motherboard, it is necessary for all of those vendors to adapt this technology. This sadly did not happen in the 2005-10 time frame, hence as of today such memories are not very popular. However, simpler variants of FB-DIMMs such as registered memories have become commonplace (as of 2020), and it looks like that the industry is making evolutionary changes in this direction.

Registered Memory

As compared to FB-DIMMs that have large overheads, registered memory is a much simpler technology that is in use today, and in many ways has taken over the space that FB-DIMMs were supposed to occupy. Registered memory modules (RDIMMs) have a register associated with a DIMM. This buffers memory addresses and commands, effectively reducing the capacitive loading on the address and command buses. Some variants of RDIMMs called LRDIMMs (Load-Reduced DIMMs) also place buffers on the data bus as well. Other than placing simple buffers on regular DDRX buses, they do not have any of the sophisticated features of FB-DIMMs.

Given that reads and writes are delayed by an extra cycle, there is an associated performance penalty. However, this is offset by the fact that buses can run at a higher frequency when using RDIMMs and can support more DIMMs. RDIMMs are very popular in the server market. The pitfalls of this technology are that motherboards need to be designed differently to support RDIMMs. Moreover, it is typically not possible to have a mix of regular DIMMs and RDIMMs.

10.3 DRAM Timing

The memory controller and the DRAM banks communicate via an elaborate set of commands. These commands have their own timing requirements, and there are rules that dictate when a command can be issued after another command. In this section, let us look at the world of DRAM commands and the DRAM access protocol in general.

We primarily perform three simple operations: read, write, and refresh. However, to support these operations we need a large portfolio of commands such that we can extract the maximum possible throughput from today's complex DRAM systems. We can divide the life cycle of every operation into four distinct stages: (1) transporting and decoding the command, (2) performing the actual read or write in the DRAM array, (3) moving the data within the DRAM chip, and (4) transmitting the result on the

bus back to the processor. Each of these stages has its own set of commands and timing requirements. We shall broadly describe the latest DDR4 protocol in the next few sections and abstract away many of the details for the ease of explanation. Note that many simplifications have also been made for the sake of clarity. For an accurate description of the DDR4 protocol readers can refer to the corresponding JEDEC standard [JEDEC Solid State Technology Association, 2020]. This section presents only a very small subset of the overall protocol. Note that all the copyrights belong to JEDEC. The material in this section is reproduced with permission from JEDEC.

We assume a single DRAM device (DRAM chip) with 16 banks divided into groups of banks. We can create groups of 4 banks each or create groups with 8 banks each. Assume that all delays are in terms of bus cycles.

10.3.1 State Diagram

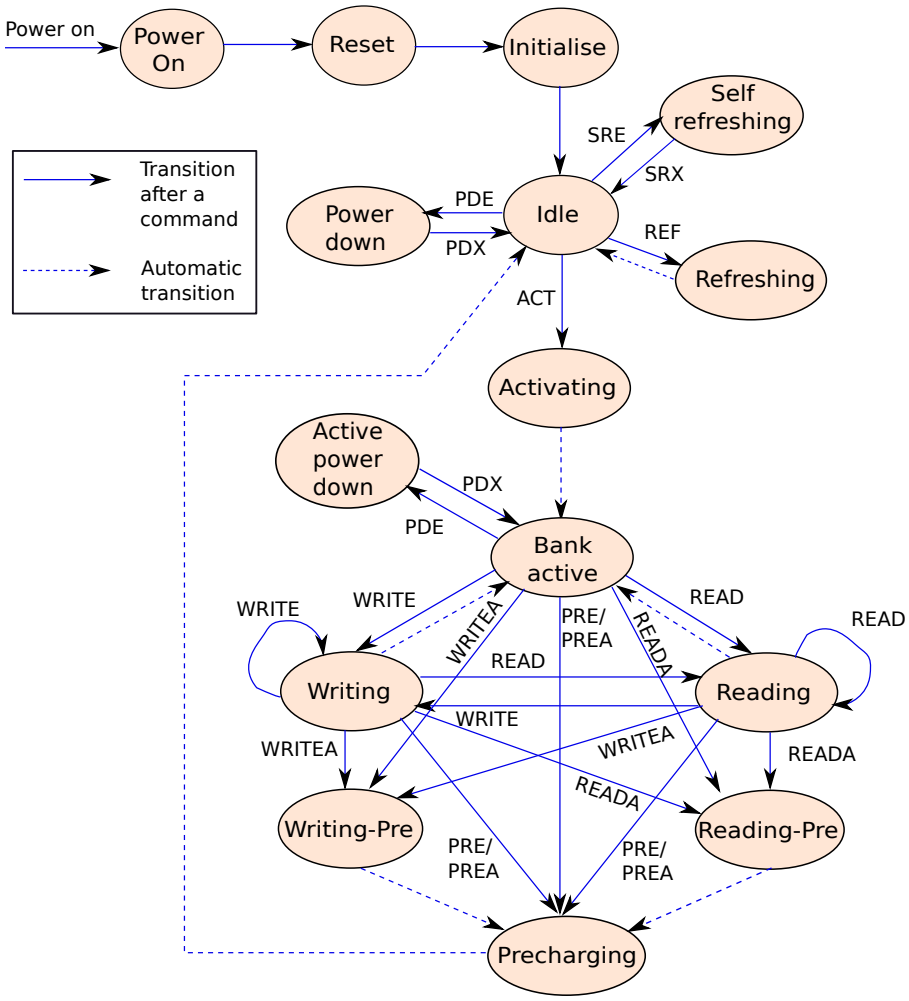

Figure 10.23: State diagram of the DDR4 protocol. Copyright JEDEC. Reproduced with permissions from JEDEC. Source: [JEDEC Solid State Technology Association, 2020]

Figure 10.23 shows the state diagram of the DDR4 protocol from the point of view of a DRAM device (DRAM chip). There are two kinds of arrows: solid and dashed. The solid arrows indicate a state

transition that happens because we either receive a command from the memory controller or a command is generated internally. The dashed arrows indicate a state transition that happens automatically.

We first power the device on, reset its state, initialize it, and then reach the *Idle* state. Along with commands for reading and writing, there are many commands for managing the device. Let us look at them first.

Modern DRAMs support two kinds of refresh modes: external refresh and self refresh. An *external refresh* means that the memory controller sends the *REF* command to the device, and then it enters the *Refreshing* state. However, if the CPU is powered down, the DRAM can still maintain its values by using the self refresh mechanism. In this case, it has a built-in timer that generates two commands *SRE* (self refresh enter) and *SRX* (self refresh exit). These are used to enter and exit the *Self refreshing* state, respectively. If there is no activity, we can save power by entering the *Power Down* state.

Before accessing any row, it is necessary to activate it first. The controller sends the ACT command to the device, the device activates the corresponding row and transitions to the *Bank active* state. In this state, we can also power the device down and transition to the *Active power down* state upon receiving the *PDE* command and later exit this state after receiving the *PDX* command.

For reading and writing, there are two kinds of commands: one without auto-precharge and one with auto-precharge. The former class of commands keep the row open, which means that it is possible for subsequent reads and writes to access columns in the same row. The contents of the row are buffered in the sense amplifiers. In this class, there are two commands: *READ* and *WRITE*. Upon receiving them, the state transitions to the *Reading* and *Writing* states respectively. After the operations are over, the device switches back to the *Bank active* state. The row remains open for future accesses. In the *Writing* state, if the device gets a *READ* command, it transitions to the *Reading* state and vice versa.

The next set of commands automatically precharge the DRAM array after the commands finish executing. These commands are *READA* and *WRITEA* for reading and writing, respectively. Whenever, the device receives a *READA* command, it executes the read and also transitions to the *Reading-Pre* state. It behaves similarly when it receives a *WRITEA* command: it transitions to the *Writing-Pre* state along with performing the write. From both of these states, an automatic transition is made to the *Precharging* state where all the bit lines are precharged, and made ready for a subsequent memory access to another row.

From any state, it is possible to enter the *Precharging* state directly by issuing the *PRE* (precharge one bank) and *PREA* (precharge all banks) commands.

This state diagram determines the operation of each DRAM device. The memory controller also keeps a copy of the state of each DRAM device and also tracks the transitions. This is done such that the memory controller can issue the right commands at the right instances of time.

Next, let us look at the major commands used for controlling and operating DRAM devices, and their associated timing constraints.

10.3.2 Activate and Precharge Commands

Activate Command

The activate command ACT is used to activate a given row in a bank, read all the columns into the sense amplifiers, and wait for a subsequent read or write access. Note that each row (or page) can be fairly large: 1024 or 2048 bits. Whenever we activate a row, all of this data is brought into the sense amplifiers, and subsequently the column multiplexers choose a subset of the bits. This is thus an expensive operation.

The timing parameters associated with the ACT command are shown in Table 10.2. The way to interpret this table is as follows. The first column indicates a pair of events, and the second column shows the minimum time interval between the first event and the second event. The events can either be external commands or internally generated commands. For example, the parameter $tRAS$ refers to the minimum amount of time that needs to elapse between issuing an ACT command and subsequently

Consecutive pair of commands	Time interval
$ACT \rightarrow PRE$	$tRAS$
$ACT \rightarrow \langle IntREAD/IntWRITE \rangle$	$tRCD$
$ACT \rightarrow ACT$	tRC
$ACT \rightarrow REF$	
Relationships: $tRC > tRAS > tRCD$	

Table 10.2: Timing constraints for the ACT command

issuing the PRE (precharge) command. We can also specify the minimum time interval between a command, and issuing an internal command such as $IntREAD$. The second row means that we issue the internal read command $IntREAD$ at least after $tRCD$ (row to column delay) units of time after issuing the ACT command. An internal read command is issued to read the values stored in the sense amplifiers. The internal write command $IntWRITE$ is defined on similar lines.

Next, let us consider tRC (row cycle time). It is the largest among the three parameters because it specifies the duration of the entire process: activate a row, close the row, and precharge.

Precharge Command

The precharge commands, PRE and $PREA$, are used to precharge the bit lines in a given bank, or in all the banks respectively. The row is subsequently deactivated, and the bank enters the *Idle* state. For a subsequent access, we need to activate a row first.

Consecutive pair of commands	Time interval
$PRE \rightarrow ACT$	tRP
Relationships: $tRC = tRAS + tRP$	

Table 10.3: Timing constraints for the PRE command

The timing parameters associated with the precharge command PRE are shown in Table 10.3. The minimum time interval between issuing the PRE command and a subsequent ACT command is tRP. We thus have $tRC = tRAS + tRP$ (refer to Tables 10.2 and 10.3). In other words, the minimum row cycle time is equal to the sum of the time it takes to issue a precharge command after activation and the minimum duration of precharging.

Timing Constraints for Limiting Power Consumption

Consecutive pair of commands	Time interval
Different bank group: $ACT \rightarrow ACT$	$tRRD_S$
Same bank group: $ACT \rightarrow ACT$	$tRRD_L$
Four-bank Activation Window	$tFAW$
Relationships: $tFAW \geq 4 * tRRD_S$ and $tRRD_L \geq tRRD_S$	

Table 10.4: Timing constraints added for limiting power consumption

In modern DRAM systems, power and temperature are important issues. Hence, it is necessary to limit the power consumption of DRAM devices. As a result, there are two timing parameters to limit the power usage of DRAM devices (refer to Table 10.4). Both these parameters target row activations,

because a row activation is an extremely power-hungry operation. We need to read an entire row of 512-2048 cells, and store their contents in the sense amplifiers. Furthermore, since reads are destructive, the data needs to be restored. Consequently, we need to have a minimum delay between row activations such that we can limit the power consumption.

The first is the row-to-row delay (t_{RRD}). This is the minimum time interval between activating a given row and activating another row. There are two types of row-to-row delays: $tRRD_S$ (short) and $tRRD_L$ (long). $tRRD_S$ is the minimum delay when the rows are in different bank groups. Whereas, $tRRD_L$ is the minimum delay when the rows belong to different banks in the same bank group. $tRRD_L \geq tRRD_S$ because we wish to enforce a power constraint for each bank group. Here *same* means *same as the previous access*, and the term *different* is defined likewise. This definition holds for the rest of the commands that use the same nomenclature. This means that we are discouraging consecutive row activations in the same bank group, which needs to be done to limit power consumption and local temperature rise.

Another parameter that limits the device-wide power consumption is $tFAW$ (Four-bank Activation Window). This means that in any sliding window of time that is $tFAW$ cycles wide, we can have at most four row activations. For example, if this window is 50 cycles wide, then there is no 50-cycle window of time in which there are more than four row activations. This limits the overall power consumption of the DRAM device.

10.3.3 Read Operation

To initiate a read, the memory controller sends the $READ$ command along with the number of the column (on the address bus), and the address of the bank group. Subsequently, the device starts to read the values by issuing the internal read command $IntREAD$.

Preamble and Postamble

Modern DRAM devices have extremely high frequencies and thus recovering the clock signal is difficult. We thus have a data strobe signal, DQS, that helps the receiver properly latch the data. However, with extremely high frequencies this also proves to be difficult. Hence, there is the notion of adding a preamble to the data transmission. The preamble is a set of cycles in which we do not transmit data. We allow the receiver to synchronize its clock with the transmitted data. This is known as the *read preamble*.

The read preamble is typically 1-2 cycles. This process of adjusting the clocks before a read operation is known as *read leveling*.

Similar to the preamble, we also have the option of adding a postamble where we wait for a given number of cycles after transmitting the last data bit before starting the next transmission. This allows the receiver to latch all the bits correctly. The preamble and postamble are graphically shown in Figure 10.24.

We have two timing parameters defined for these operations: $RPRE$ and $RPST$ (see Table 10.5). In a burst of read and write commands, the requirement of the preamble and postamble is sometimes relaxed because there is no need to synchronize the clock.

Operation	Duration
Read preamble	$tRPRE$
Read postamble	$tRPST$

Table 10.5: Timing constraints for the preamble and postamble

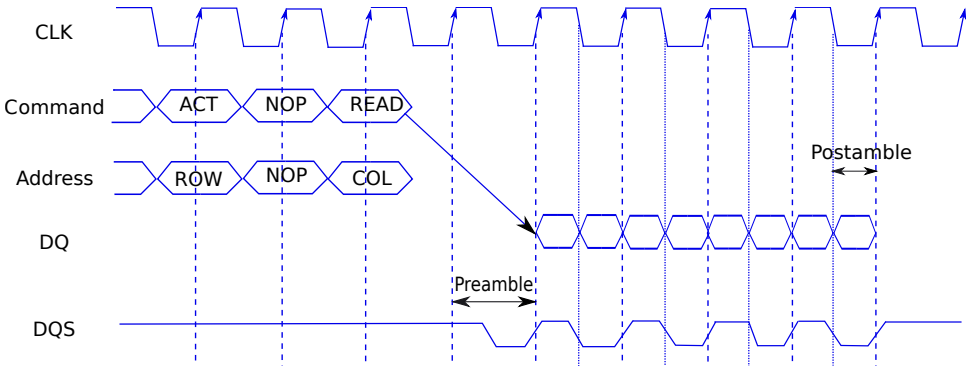

Figure 10.24: The read preamble and postamble in the DDR4 protocol

Consecutive pair of commands	Time interval
$READ \rightarrow \langle$First data bit on the bus$\rangle$	RL
$READ \rightarrow IntREAD$	AL
$IntREAD \rightarrow \langle$First data bit on the bus without parity checking$\rangle$	CL
Time to verify the parity	PL
$IntREAD \rightarrow PRE$	$tRTP$
Relationships: $RL = AL + CL + PL$	

Table 10.6: Timing constraints for the read operation

$READ$ command

The parameters for the read operation are shown in Table 10.6. The time between issuing the $READ$ command and getting the first data bit is the read latency RL. Subsequently, we get 1 bit every half-cycle (beat) depending upon the burst length. The read latency can be broken down into three components: additive latency (AL), CAS latency (CL), and the parity latency(PL). We have $RL = AL + CL + PL$.

The DDR4 protocol allows us to issue a $READ$ command immediately after an ACT command. However, we need to wait for $tRCD$ time units before we can start an internal read operation. This means that the $READ$ command needs to be internally buffered till the device is ready to issue an internal read. This delay is known as the additive latency (AL). Once an internal read command is issued, the time it takes to put the first data bit on the bus is the CAS latency CL. Furthermore, we can program the device to check for any parity errors before the data is sent on the bus. This takes some additional time, which is known as the parity latency (PL). We thus have $RL = AL + PL + CL$.

Finally, we define $tRTP$ as the delay between an internal read command and the precharge command. This is the amount of time that it takes to complete the read operation and send the data to the buffers of the bus. After this operation is done, the precharge command can be issued. We can infer that the minimum spacing between an external read command $READ$ and the PRE command is $AL + tRTP$ time units.

Burst of Read Commands

Till now, we have considered a single read command. Let us now consider a burst of read commands for different columns in the same row (see Table 10.7).

We define the column-to-column delay for access to the same bank group as $tCCD_L$, and for access to different bank groups as $tCCD_S$. This is the minimum duration between issuing two $READ$ commands

Consecutive pair of commands	Time interval
Column to column delay (same bank group)	$tCCD_L$
Column to column delay (different bank group)	$tCCD_S$
Relationships: $tCCD_L \geq tCCD_S$	

Table 10.7: Timing constraints for the column-to-column delay

to two different columns across the same or different bank groups. It is faster to access a different bank group than the same bank group ($tCCD_L \geq tCCD_S$). This is a standard feature of DDR4. The reason is that the same bank group (one that is being currently used) has many resources allocated for the current transfer, whereas, a different bank group has more resources available and is thus faster to access. Additionally, we wish to limit localized power consumption.

10.3.4 Write Operation

The write operation is similar to the read operation. It is initiated by sending a $WRITE$ command. At the same time, we send the column address on the address bus and the address of the bank group. Let us discuss the specifics.

Preamble and Postamble

Similar to the $READ$ command we also have a preamble and postamble (refer to Table 10.8) for the $WRITE$ command. The reasons are similar – detecting the relative skews between the data, strobe, and the internal clock of the DRAM. Once we detect the skews, we can correctly latch the data. This process of training the circuit for correctly latching the data is known as *write leveling*.

Operation	Duration
Write preamble	$tWPRE$
Write postamble	$tWPST$

Table 10.8: Timing constraints for the write preamble and postamble

$WRITE$ command

The $WRITE$ command is similar to that $READ$ command in terms of the way it is issued. In this case, we define a term called the write latency (WL). It is the sum of the additive latency (AL), the parity latency (PL), and the CAS write latency (CWL). Refer to Table 10.9 for a description of the parameters that are relevant to the write operation. We have $WL = AL + PL + CWL$.

Consecutive pair of commands	Duration
$WRITE \rightarrow \langle$First data bit on the bus$\rangle$	WL
$WRITE \rightarrow IntWRITE$	AL
Time to compute the parity	PL
$IntWRITE \rightarrow \langle$First data bit on the bus$\rangle$	CWL
Relationships: $WL = AL + PL + CWL$	

Table 10.9: Timing constraints for the write operation

Burst of Write Commands

In a write burst, we use the same timing constraints as the $READ$ command for issuing different $WRITE$ commands. The parameters $tCCD_S$ and $tCCD_L$ are relevant (with the same meanings) for writes as well.

10.3.5 Interaction between the Read, Write, and Precharge Operations

The $READ$ and $WRITE$ commands have several interactions between them. The timing constraints are shown in Table 10.10.

Consecutive pair of commands	Time interval
$READ \rightarrow WRITE$	$tRTW$
$WRITE \rightarrow READ$	$tWTR$
Burst length	tBL
$WRITE \rightarrow PRE$	tWR

Table 10.10: Interaction between read and write operations

$tRTW$ (read to write) is the minimum time interval between a $READ$ and a subsequent $WRITE$ command. Similarly, $tWTR$ (write to read) is the minimum time interval between a $WRITE$ command, and a subsequent $READ$ command. We also define the parameter tBL, which is the burst length – the number of beats used to transmit data for a single command. As we have discussed in Section 10.2.2, for DDR3 and DDR4, it is normally equal to 8 beats. If the burst-chop mode is used, it is equal to 4 beats.

Read to Write Delay

Let us derive the timing relationship between issuing a $READ$ command and issuing a $WRITE$ command. This is shown in Figure 10.25. Let us make a simplistic assumption that $PL = 0$ for the ease of explanation.

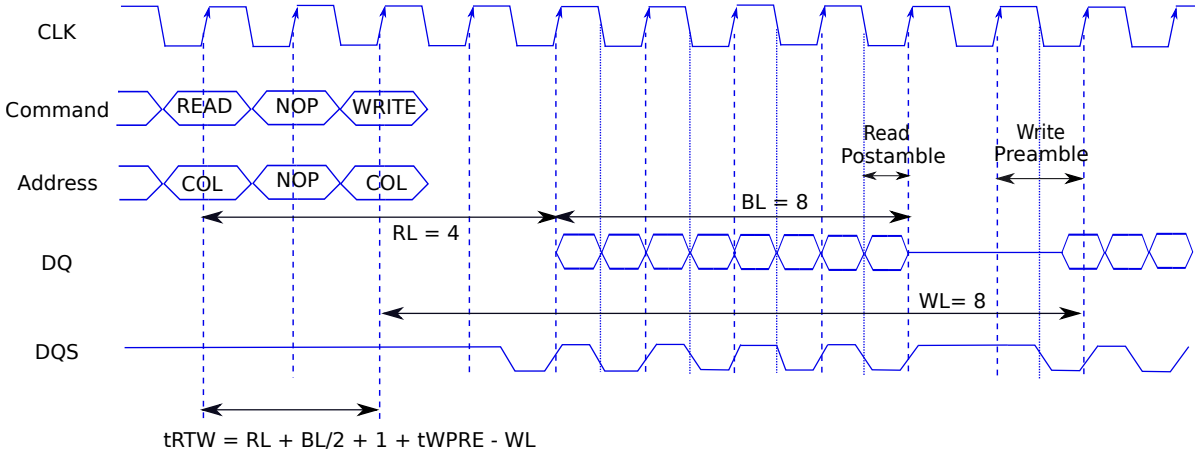

Figure 10.25: Read to write delay ($tRTW$)

If the $READ$ command is issued at the beginning of cycle 0, then the earliest we can write data to the bus is cycle $RL + BL/2 + tWPRE + 1$ (refer to Figure 10.25). We add $BL/2$ because we are

assuming that the unit of BL is half-cycles. In most DDR4 devices, we need to add the additional 1 cycle delay because for starting a write operation we need to change the direction of the data transmission on the bus. Given that the write latency is WL, the minimum spacing between the $READ$ and $WRITE$ commands is therefore equal to $RL + BL/2 + tWPRE + 1 - WL$. Note that in this case the read postamble is getting subsumed within the read burst. Otherwise, we needed to add a component of it as well.

Write to Read Delay

The $tWTR$ (write to read) parameter is defined differently. It refers to the minimum interval between the end of a write operation (including its postamble) and a subsequent $READ$ command. Here also, we can define two parameters $tWTR_S$ (different bank group) and $tWTR_L$ (same bank group). We have the same relationship: $tWTRL_L \geq tWTR_S$. A write operation is slightly slower than a read because after every write we need to do some more work. $tWTR$ cycles are needed to ensure that we are able to write the correct state to the sense amplifiers and the DRAM cells. This requires time because some bit lines need to undergo voltage transitions.

Write Recovery Time

The write recovery time tWR is defined in a similar manner as the write-to-read time. Its corresponding interval starts from the end of a write (including its postamble). It is the minimum duration from this point till we can issue a precharge command. The reason for the write recovery time is similar to the extra time required for starting a read operation after a write operation ($tWTR$).

10.3.6 Refresh Operation

Every time that the memory controller requires a refresh, it issues the REF (refresh) command. DDR4 memories require a refresh cycle once every $tREFI$ cycles. Before a refresh command is sent, all the banks need to be precharged and be idle for at least tRP cycles. Every DRAM device has an internal refresh controller that generates a list of all the addresses that need to be refreshed. The entire process takes $tRFC$ cycles (refresh cycle time). All the banks are set to the idle state and can be accessed.

Modern memory protocols provide some flexibility in scheduling refresh commands particularly if the memory traffic is high. They allow us to postpone refresh commands subject to a maximum limit.

Along with an external refresh mode, most DDR devices also have a self-refresh mode that allows them to retain their data even when the CPU is powered down. The process is similar to that of an external refresh. The timing constraints are summarized in Table 10.11.

Interval	Duration
Maximum refresh interval	$tREFI$
Refresh cycle time	$tRFC$

Table 10.11: Timing constraints for the refresh commands

10.3.7 Example of a Protocol

Let us enumerate the parameters of the DDR 1600 protocol. It can support 1600 transfers per second with an 800 MHz clock rate. The clock cycle time is thus 1.25 ns. Table 10.12 lists all the parameters. The unit is clock cycles.

Parameter	Mnemonic	Value (cycles)
Row cycle time	tRC	38
ACT to PRE command period	$tRAS$	28
PRE to ACT command period	tRP	10
ACT to internal read or write	$tRCD$	10
CAS latency	CL	10
CAS write latency	CWL	9
Column to column delay (same bank group)	$tCCD_L$	5
Column to column delay (different bank group)	$tCCD_S$	4
Row to row delay (same bank group, 1Kb page size)	$tRRD_L$	5
Row to row delay (different bank group, 1Kb page size)	$tRRD_S$	4
Four-bank activation window (1Kb page size)	$tFAW$	20
Write to read latency (same bank group)	$tWTR_L$	6
Write to read latency (different bank group)	$tWTR_S$	2
$IntREAD$ to PRE command period	$tRTP$	6
Write recovery time	tWR	12

Table 10.12: Parameters for the DDR 1600 protocol

10.4 Memory Controller

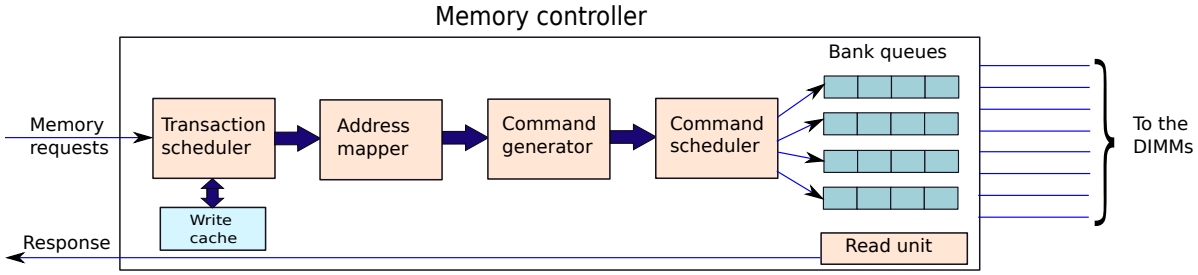

Figure 10.26: Structure of the memory controller

The general structure of the memory controller is shown in Figure 10.26. We typically have one memory controller per channel.

It receives requests from different cache banks. All the requested memory transactions are first handed over to the transaction scheduler. It ascertains the priority of the transaction with respect to other transactions that are being processed. For example, at this stage it can decide that a much-needed refresh operation should have the highest priority. The second stage, address mapper, maps the physical memory address to addresses on the DRAM devices. For a given physical address, this stage maps it to a given rank and a set of banks. There are a range of possibilities here depending on the number of DRAM devices, the number of arrays within each device, and the request pattern. After the address mapping stage, the memory request is converted to a sequence of DRAM commands by the command generator, which are then inserted into a set of bank-specific queues. Finally, we have a command scheduler, that dispatches the commands to the banks. This needs to follow the timing rules that we discussed in Section 10.3.

To summarize, the main components in the memory controller are as follows.

1. A high-level read/write transaction scheduler.
2. Address mapping engine.
3. DRAM command generation unit.
4. Command scheduling unit.

Let us revisit the notion of the *rank*. Multiple DIMMs are connected to each channel, and we have multiple DRAM devices on each DIMM. All of them share the same address and command buses. This makes life complicated. This is because in principle all the devices can have different clock skews because their distance from the memory controller might vary. However, this is not allowed because ensuring proper timing would become a very onerous task. Hence, we divide the set of devices into ranks. All the devices in each rank have roughly the same degree of clock skew and are equally distant from the memory controller in terms of timing. As a result, it is possible to run all the DRAM devices in a rank in unison.

In the DDR4 protocol, we specify three pieces of information with each command: chip-select id, bank group id, and bank id. The chip select signal selects the rank. The rest of the ranks are disabled for that command. This signal is routed to all the devices within the rank such that they all get enabled. The bank group and bank id bits are used to choose a specific bank within each device. Then the command is delivered to all the chosen banks across the DRAM devices. Let us consider a concrete example. Assume we have 8 DRAM devices in a rank with 4 banks per device and 8 arrays per bank. Thus, each bank can read 8 bits at a time (one bit from each array). When we send the read command, we also send the id of the bank (and the bank group, if we have bank groups). For example, if we send bank id 2, then this selects all the banks with id 2 in all the 8 DRAM devices. The read access commences. Once the data is ready each selected bank produces 8 bits (1 bit per array). Since we have selected 8 banks (1 bank per device), all 8 of them produce 64 bits in a single cycle. These bits can then be sent to the memory controller.

Since read accesses take time, we can utilize the time in the middle to activate and schedule requests on other banks. Note that in most memory systems, *accessing a bank* means accessing the same bank id across all the devices. Some advanced memory protocols such as DDR4 do allow commands to be sent to individual devices in a rank; however, that is mainly for setting specific electrical parameters.

Important Point 19
It is important to note that the reason we organize memory devices into a rank is such that we can increase the amount of parallelism and read or write a large number of bits in a single cycle. Furthermore, each memory device has multiple banks where only one bank is allowed to process a command from the bus in a given cycle. For any memory request, we enable one bank from each device, and then send a read/write/activate request to the set of enabled banks across the devices in a rank; they work in lockstep.

10.4.1 DRAM Transaction Scheduling

Each DRAM controller receives memory requests from the NoC. An overwhelming majority of them are either read or write requests. Sometimes it can also generate its own requests such as refresh operations.

The first part of transaction scheduling is to decide the relative priorities of the operations. Typically, memory controllers prioritize read requests because they are most of the time on the critical path, whereas write requests can be deferred. Additionally, in the DDR4 protocol, back-to-back read requests are much faster than request pairs in which the first request is a write. Recall that we need time to recover from a write because along with writing to the sense amplifier array, we also need to write to the memory cells.

As a result, it is a much better idea to schedule write operations when the DRAM devices are relatively free.

Write Caching

Sadly reordering reads and writes can introduce correctness and consistency problems. This can be easily solved by adding a write cache (similar to a write buffer). This stores all the outstanding writes. A read operation first checks for its address in the write cache. If there is a hit, then it returns with the value stored in the write cache, otherwise if there is a miss, then the read operation is sent to the DRAM devices.

Open-Page Access Policy

One major advantage of having a sense amplifier array is that it can store an entire row (512, 1024, or 2048 bits). Recall that we refer to a row as a page as well, and the storage part of the sense amplifier array is synonymously referred to as the *row buffer* or the *page buffer*. In every access, we typically need to access a subset of these bits. Here, the advantage is that we need not perform a precharge operation after every read or write, instead if there is temporal and spatial locality, then this row buffer can be used as a cache. This saves us unnecessary activate and precharge operations. This is efficient in terms of both time and power.

This policy is known as the open-page access policy where after a read or write operation completes, we do not set the bank to the *idle state*. Instead, we keep the row open such that it can serve later requests that map to the same row. Because of temporal and spatial locality, we expect a lot of hits to the open row (or page). To ensure that the hit rate is high, most memory controllers reorder the requests such that contiguous requests map to the same open page.

Once we run out of requests, we precharge the bank and set its state to the *idle* state.

Close-Page Access Policy

The close-page access policy closes the row after it has been accessed once. This favors a random-access pattern, where there is very little spatial and temporal locality. The memory controller issues the $READA$ and $WRITEA$ commands that immediately issue a precharge after completing the read or write respectively.

Hybrid Access Policy

Most memory controllers as of today use a hybrid access policy, where they decide when the row should be kept open and when it should be closed. They monitor the sequence of memory accesses and if they predict that a given row is unlikely to be used again in the near future, then it is immediately closed after the access.

Managing Refreshes

A typical DRAM row can hold its data for 32 to 64 ms. It needs to be refreshed at least once during this period. Let us assume that it can hold its data for 64 ms, and we have 8192 rows. Then we need to send a refresh command to the devices once every 64,000/8192 = 7.8 μs. Each refresh command refreshes a given row. During that period (refresh cycle time: $tRFC$), the bank cannot be used for any other purpose.

Modern DRAM devices have refresh modes: self refresh and external refresh. The self refresh mode is useful when the CPU is powered down or is not in a position to send refresh commands. Otherwise, we rely on external refresh commands, where the memory controller explicitly sends refresh commands to each row. Sometimes it is possible that we may have to delay critical read and write requests to

accommodate a refresh. The DDR4 protocol does give us some flexibility in this regard. We can defer a refresh message by up to 8 refresh intervals (a refresh interval is 7.8 μs in our example). During this period we can finish sending critical read and write messages. After that we need to quickly finish all the pending refresh operations.

10.4.2 Address Mapping

Every physical address needs to be mapped to an equivalent DRAM address. A DRAM address is a combination of the channel id, the rank, the bank, the row, and the set of columns. This determines the available parallelism, number of bank conflicts, and the time it takes to wait between consecutive accesses. The address mapping scheme is dependent on the technology that is used, latencies of different operations, and the memory access pattern.

Basic Terminology

Symbol	Description
c	Channel id
k	Rank id
g	Bank group id
b	Bank id
r	Row id
l	64-byte block id in a row

Table 10.13: Symbols used in address mapping

Table 10.13 shows the symbols that we shall use to describe the address mapping scheme. Let us first go over our main assumptions.

1. We read or write data at the granularity of 64-byte blocks.
2. We assume that each row stores an integral number of 64-byte blocks. The id of a block in a row is denoted by the symbol l.
3. All the counts start from zero.
4. We assume that a physical address given to the DRAM memory controller points to the starting address of a 64–byte block. In each beat, we transfer 64 bits.
5. If the paging mechanism has indicated that a given frame is present in the DRAM, then we are guaranteed to find the frame. Unlike a cache, there are no misses in main memory. We would like to reiterate that a page in DRAM is not the same as a virtual memory page.

We can deduce that the address of each 64-byte block in the DRAM memory is uniquely specified using $c + k + g + b + r + l$ $(= N)$ bits. To operationalize this, once the memory controller gets the address of the block, it retrieves N bits from the address, and discards the rest of the bits. These are typically the least significant bits of the block address. The virtual memory system needs to ensure that no two blocks who have these N bits in common in their physical addresses are present in the DRAM memory at the same time. This will remove the need for having a tag array. This further ensures that our DRAM structure can be used in modern processors very easily. Note that this design choice is not a significant issue when it comes to performance because main memories are typically very large, and we have a lot of locality at the page level.

Now we need to decide which bits we need to use to address the channel, which bits we need to address the rank, and so on. Any addressing scheme can be described using the following format. Let's say we have a scheme, $c : k : g : b : r : l$. This means that in the N-bit block address, we use the least significant l bits to address the block in the row, the next r bits to address the row in the bank, and so on. In comparison, the addressing scheme $k : b : r : l : g : c$ means that we use the least significant c bits to address the channel, the next g bits for the bank group id, and so on.

Let us create addressing schemes for the open-page and close-page policies.

Addressing for the Open-Page Policy

The key insight is that the least significant bits tend to vary more, and there is more randomness in them. In comparison, the more significant bits vary less. If for some reason we want to distribute the accesses, we should use the least significant bits.

Out of the six parameters – c, k, g, b, r, l – we need to first decide which parameter we should map to the least significant bits. Let us look at the most common access pattern, which is accessing the next block address. Most of the time we tend to read consecutive blocks because this is the typical pattern of both instruction accesses and data accesses, notably array or stack accesses. Hence, we should optimize the addressing method for this pattern.

Switching between channels is the cheapest operation in terms of time. We typically have different memory controllers for different channels, and it is possible to operate them independently. We can even read consecutive blocks in parallel if they are mapped to different channels. Hence, our addressing scheme should be of the form $x : x : x : x : x : c$. Here, x, refers to a parameter that is unspecified as of now.

Since we are considering the open-page access policy, the row buffer will continue to maintain the contents of the entire row unless it is explicitly precharged. We can leverage this fact and read the next block from the row buffer. Hence, the next set of bits should be mapped to the parameter l (block id in a row). The addressing scheme thus becomes $x : x : x : x : l : c$.

Let us now make a choice between a different bank and a different rank. We shall discuss why we are discarding the row id at this stage slightly later. To send an access to a different rank, we need to resynchronize the timing of the processor-memory bus. Recall that different banks have different timing requirements and have different clock skews. Some re-synchronization, read leveling, and write leveling needs to be done. This will take time. Consequently, let us reduce the priority of switching the rank.

Let us thus maximize locality at the level of the banks by switching to another bank in the same rank. Recall that banks are independent of each other. We can activate bank 2, while bank 1 is completing a read operation. This feature can be leveraged to send consecutive accesses to other banks. Recall from Section 10.3 that the delay is larger if we send a request to another bank in the same bank group; therefore, we need to send a request to a different bank group. To achieve more randomness in this regard, let us map the next set of bits to the bank group g. After this, let us consider the bank id b. Thus, our addressing scheme now looks like $x : x : b : g : l : c$.

We are now left with the choice of the rank id and row id. We have already discussed that switching to a new rank involves costly re-synchronization. However, switching to a new row is even more expensive. We need to close the row buffer by performing a precharge operation, and then activate the new row. This is very expensive in terms of time. Hence, let us prefer switching to a new rank.

The final addressing scheme is $r : k : b : g : l : c$.

Addressing for the Close-Page Policy

Here also the first priority should be the channel because it is very easy to switch between channels, and requests can be sent to different channels in parallel. Since we do not intend to reuse the row, we need to look at increasing bank-level parallelism. We thus map the next few bits to b and g. The addressing scheme at this stage is $x : x : x : b : g : c$.

Next we can switch the rank because that is much faster than precharging and accessing a new row. Hence, the scheme becomes $x : x : k : b : g : c$.

We now have a choice between the id of the block in a row (l), and the row id (r). Since we close the row after an access, we cannot leverage any locality at the level of the row buffer. However, we can always take advantage of caching schemes at the memory controller, if we stick to the same row. Hence, we map the next few bits to l.

Therefore, the final addressing scheme is $r : l : k : b : g : c$.

10.4.3 Command Scheduling

After the address mapping stage, we generate the DRAM commands and store them in a set of queues. The queues are typically bank specific, which means that each bank has a set of queues. Furthermore, for each bank we can have three separate sub-queues: a read queue, a write queue, and a refresh queue. Let us discuss different scheduling mechanisms for the different bank queues.

Bank Round-Robin (BRR)

This is one of the simplest scheduling mechanisms. For a given rank, the scheduler follows a round-robin algorithm. It visits each bank queue, picks a request, and sends it to the DIMMs. At this stage, several optimizations are possible. For example, we can prioritize reads as compared to writes. The DDR protocols allow us to send a $READ$ command immediately after an ACT command. This can be done at this stage. This is a beneficial feature because we need to otherwise wait for a long time before we revisit the same bank queue once again.

Once we are finished with processing the requests of a given rank, we move to the next rank. Note that at this stage, the memory controller also maintains some state regarding the timing of the commands such that no timing constraints are violated.

Rank Round-Robin (RRR)

The rank round-robin algorithm (RRR) differs slightly from the bank round-robin algorithm (BRR) in terms of the order of accessing the queues.

We first access the same bank id in each rank, and then go to the next bank id. This approach distributes requests between ranks more uniformly as opposed to BRR. Note that the choice of the command scheduling algorithm and the addressing scheme are related. In practice, memory controller designers take all of this into account, conduct extensive simulations, and then create a design that provides the highest aggregate speedup for a wide variety of memory access patterns.

Greedy

Both the scheduling algorithms, BRR and RRR, are relatively oblivious of the timing constraints while making their scheduling decisions. Their main aim is fairness across banks and ranks. Note that they still need to observe timing constraints, and this is done after the scheduling decision has been made.

As opposed to this philosophy, the greedy algorithm works very differently. Out of all the bank queues we choose that request that can be issued (sent to the DIMMs) as soon as possible. This ensures that our system waits as little as possible to issue commands. Even though this approach works well in a lightly loaded system, there can be issues with starvation and fairness in a moderate to heavily loaded system.

Most practical scheduling algorithms in modern processors try to maximize performance by minimizing the waiting time, and simultaneously also try to not compromise on fairness significantly.

10.5 Emerging Memory Technologies

The single largest criticism of modern DRAM memory is that it is *volatile* in nature. This means that when we turn the power off, a DRAM chip loses all of its data. It thus cannot be used as a persistent storage medium. The next time that we restart the machine, it will be necessary to repopulate the DRAM by reading all the pages that it needs to have from the disk. This increases the system startup time and induces page faults during execution. Wouldn't it be nice to have a memory that is *nonvolatile* in nature? This means that it will not lose its data when the power is switched off. In this section, we shall look at the basic physics, and the design of such nonvolatile memories, henceforth referred to as NVMs.

To design a nonvolatile memory device, it needs to have two distinct permanent states. As long as we can switch the device from one state to the other and back, we have a functional memory cell. These memory cells can then be interconnected to create a memory array, which can then be connected to the CPU via a set of memory channels. In this section, we shall look at different kinds of nonvolatile memories.

Please note that a nonvolatile memory is a storage device, which is meant to permanently store data. We still require regular DRAM memories for speed and efficiency. Nonvolatile memories compete with hard disks and other forms of slow secondary storage. Moreover, a nonvolatile memory can be used as a fast cache for a hard disk. Additionally, it can be used in critical applications where we do not want to spend any time in reading *cold data* from the disk.

Definition 91

- *A memory is said to be* volatile *when it loses its data after the power is switched off. DRAM memory is an example of a volatile memory.*
- *In comparison, a memory is said to be* nonvolatile *if it does not lose its data when the power is switched off. The most popular example of such memory is flash memory that we have in USB sticks. Nonvolatile memories will be referred to as NVMs henceforth.*

10.5.1 Flash Memory

Hard disks and optical drives are fairly bulky, and need to be handled carefully because they contain sensitive mechanical parts. An additional shortcoming of optical storage media is that they are very sensitive to scratches and other forms of minor accidental damage. Consequently, these devices are not ideally suited for portable and mobile applications. We need a storage device that does not consist of sensitive mechanical parts, can be carried in a pocket, can be attached to any computer, and is extremely durable. Flash drives such as USB pen drives satisfy all these requirements. A typical pen drive can fit in a wallet, can be attached to all kinds of devices, and is extremely robust and durable. It does not lose its data when it is disconnected from the computer. We have flash based storage devices in most portable devices, medical devices, industrial electronics, disk caches in high-end servers, and small data storage devices. Flash memory is an example of an EEPROM (Electrically Erasable Programmable Read Only Memory) or EPROM (Erasable Programmable Read Only Memory). Note that traditionally EPROM based memories used ultraviolet light for erasing data. They have been superseded by flash based devices.

Let us look at flash based technology in this section. The basic element of storage is a floating gate transistor.

The Floating Gate Transistor

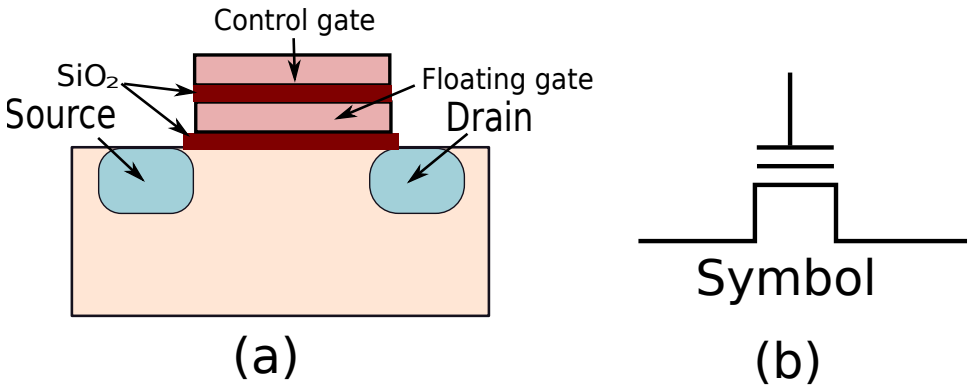

Figure 10.27: A floating gate transistor

Figure 10.27 shows a floating gate transistor. The figure shows a regular NMOS transistor with two gates instead of one. The gate on top is known as the *control gate*, and is equivalent to the gate in normal MOS transistors. The gate below the control gate is known as the floating gate. It is surrounded on all sides by an SiO_2 based electrical insulation layer. Hence, the floating gate is electrically isolated from the rest of the device. By some means if we are able to implant a certain amount of charge in the floating gate, then the floating gate will maintain its potential for a very long time. In practice, there is a negligible amount of current flow between the floating gate and the rest of the components in the floating gate transistor under normal conditions.

Let us now consider two scenarios. In the first scenario, the floating gate is not charged. In this case, the floating gate transistor acts as a regular NMOS transistor. In the second scenario, the floating gate has accumulated electrons containing negative charge (we will discuss how this can happen later). Then we have a negative potential gradient between the channel and the control gate. Recall that to create an n-type channel in the transistor, it is necessary to apply a positive voltage to the gate, where this voltage is greater than the threshold voltage. In this case, the threshold voltage is effectively higher because of the accumulation of electrons in the floating gate. In other words, to induce a channel to form in the substrate, we need to apply a larger positive voltage at the control gate.

Let the threshold voltage when the floating gate is not charged with electrons be V_T, and let the threshold voltage when the floating gate contains negative charge be V_T^+ ($V_T^+ > V_T$). If we apply a voltage that is in between V_T and V_T^+, then the NMOS transistor conducts current if no charge is stored in the floating gate. Otherwise, if charge is stored, the threshold voltage V_T^+ of the transistor is greater than the gate-to-source voltage, and thus the transistor is in the *off* state. It thus does not conduct any current. We typically assume that the default state (no charged electrons in the floating gate) corresponds to the 1 state. When the floating gate is charged with electrons, we assume that the transistor is in the 0 state.

Now, to write a value of 0 or *program* the transistor, we need to deposit electrons in the floating gate. This can be done by applying a strong positive voltage to the control gate, and a smaller positive voltage to the drain terminal. Since there is a positive potential difference between the drain and source, a channel gets established between the drain and source. The control gate has an even higher voltage, and thus the resulting electric field pulls electrons from the n-type channel and deposits some of them in the floating gate.

Similarly, to *erase* the stored 0 bit, we apply a strong negative voltage between the control gate and the source terminal. The resulting electric field pulls the electrons away from the floating gate into the substrate and source terminal. At the end of this process, the floating gate loses all its negative charge,

and the flash device comes back to its original state. It now stores a logical 1.

To summarize, programming a flash cell means writing a logical 0, and erasing it means writing a logical 1. There are two fundamental ways in which we can arrange such floating gate transistors to make a basic flash memory cell. These methods are known as NOR flash and NAND flash respectively.

NOR Flash

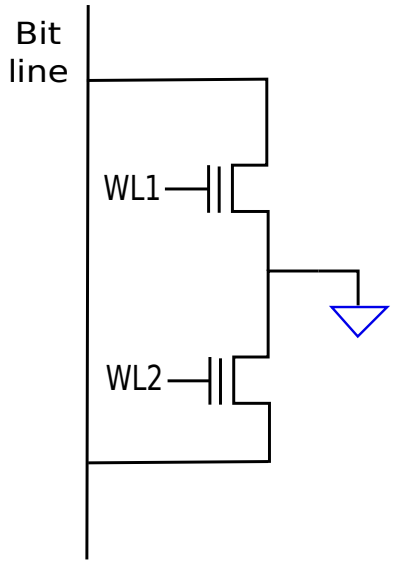

Figure 10.28: NOR Flash Cell

Figure 10.28 shows the topology of a 2-transistor NOR flash cell that saves 2 bits. Each floating gate transistor is connected to a bit line on one side and to the ground on the other side. The control gates are connected to distinct word lines. After we enable a floating gate transistor (set the voltage of the control gate to somewhere between V_T and V_T^+), it pulls the bit line low if it stores a logical 1, otherwise it does not have any effect because it is in the *off* state. Thus, the voltage transition in the bit line is logically the reverse of the value stored in the transistor. The bit line is connected to a sense amplifier that senses its voltage, flips the bit, and reports it as the output. Similarly, for writing and erasing we need to set the word lines and bit lines to appropriate voltages. The advantage of a NOR flash cell is that it is very similar to a traditional DRAM cell. We can build an array of NOR flash cells similar to a DRAM array.

NAND Flash

A NAND flash cell has a different topology. It consists of a set of NMOS floating gate transistors in series similar to series connections in CMOS NAND gates (refer to Figure 10.29). There are two dedicated transistors at both ends known as the *bit line select transistor* and *ground select* transistor, respectively. A typical array of transistors connected in the NAND configuration contains 8 or 16 transistors. To read the value saved in a certain transistor in a NAND flash array, there are three steps. The first step is to set the gate voltages of the ground select and bit line select transistors to a logical 1 such that they are conducting. The second step is to set the voltages of the control gates of the rest of the floating gate transistors other than the one we wish to read by setting their word line voltages to V_T^+. Finally, we

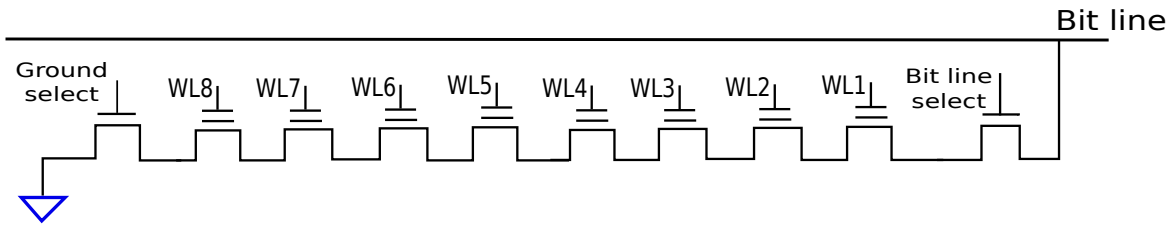

Figure 10.29: NAND flash cell

need to read the specific transistor by setting its word line voltage to some value between V_T and V_T^+. If the cell is not programmed (contains a 1), it drives the bit line low, otherwise it does not change the voltage on the bit line. Sense amplifiers infer the value of the logical bit saved in the transistor. Such arrays of floating gate transistors known as NAND flash cells are connected in a configuration similar to NOR flash cells.

This scheme might look complicated at the outset; however, it has a lot of advantages. Consequently, most of the flash devices in use today use NAND flash memories instead of NOR flash memories. The bit storage density is much higher. A typical NAND flash cell uses a lesser number of wires than a NOR flash cell because all the floating gate transistors are directly connected to each other, and there is just one connection to the bit line and ground terminal. Hence, NAND flash memories have at least 40-60% higher density as compared to NOR flash cells. Of course, accessing a single cell is more complicated. Nevertheless, given the advantages in storage density, market economics has chosen the NAND flash cell.

Blocks and Pages

The most important point to note here is that a (NAND) flash memory device is not a memory device, it is a storage device. Memory devices provide word-level access. In comparison, flash devices typically provide page-level access, where a page's size can be between 512-4096 bytes. Note that a page as defined here is different from a page in virtual memory. Due to temporal and spatial locality in accesses to flash media, the working set of most programs is restricted to a few pages.

Page-level access to flash media is typically not very inefficient. This is because of the following reason. To reduce the number of accesses to storage devices, most operating systems have in-memory storage caches such as hard disk caches. Most of the time, the operating system reads and writes to the in-memory caches. This reduces the I/O access time. Such caches are typically large contiguous regions in physical memory, and thus they can be read or written in bulk – at the level of pages. This naturally aligns with the page-level access paradigm of flash devices. Let us look at an example.

After certain events, it is necessary to synchronize the cache with the underlying storage device. For example, after executing a *sync*() system call in Linux, the hard disk cache writes its updates to the hard disk. Depending on the semantics of the operating system, and file system, writes are sent to the underlying storage media after a variety of events. For example, when we right-click the icon for a USB drive in the "My Computer" screen on Windows and select the eject option, the operating system ensures that all the outstanding write requests are sent to the USB device. This is a bulk operation and uses page-level I/O.

On a humorous note, most of the time users simply unplug a USB device. This practice can occasionally lead to data corruption, and unfortunately your author has committed this mistake several times. This is because, when we pull out a USB drive, some uncommitted changes are still present in the in-memory cache. Consequently, the USB pen drive contains stale and possibly half-written corrupted data. Hence, don't do this !!!

Data in NAND flash devices is organized at the granularity of pages and blocks. Note: the connotation

of the term *block* is different here, its definition is specific to flash devices. As we have discussed, a *page* of data typically contains 512 – 4096 bytes (in powers of 2). Most NAND flash devices can typically read or write data at the granularity of pages. Each page additionally has extra bits for error correction based on CRC codes. A set of pages are organized into a block. Blocks can contain 32 – 128 pages, and their total size ranges from 16-512 KB. Most NAND flash devices can erase data at the level of blocks.

Let us now look at some salient points of NAND flash devices.

Program/Erase Cycles

Writing to a flash device essentially means writing a logical 0 bit since by default each floating gate transistor contains a logical 1. In general, after we have written data to a block, we cannot write data again to the same block without performing additional steps. For example, if we have written 0110 to a set of locations in a block, we cannot write 1001 to the same set of locations without erasing the original data. This is because, we cannot convert a 0 to a 1 without erasing data. Erasing is a slow operation and consumes a lot of power. Hence, the designers of NAND flash memories decided to erase data at large granularities, i.e., at the granularity of a block. We can think of accesses to flash memory as consisting of a *program phase*, where data is written at the granularity of pages, and an *erase phase*, where the data stored in all the transistors of the block is erased. After an erase operation, each transistor in the block contains a logical 1. We can have an indefinite number of read accesses between the program phase, and the erase phase. Let us define a new term here: a pair of program and erase operations is known as a program/erase cycle or P/E cycle.

Unfortunately, flash devices can endure a finite number of P/E cycles. As of 2020, this number is between 50,000 to 150,000. This is because each P/E cycle damages the silicon dioxide layer surrounding the floating gate. There is a gradual breakdown of this layer, and ultimately after hundreds of thousands of P/E cycles it does not remain an electrical insulator anymore. It starts to conduct current, and thus a flash cell loses its ability to hold charge. This gradual damage to the insulator layer is known as *wear and tear*. To mitigate this problem, designers use a technique called *wear leveling*.

Wear Leveling

The main objective of *wear leveling* is to ensure that accesses are uniformly distributed across blocks. If accesses are non-uniformly distributed, then the blocks that receive a large number of requests will wear out faster, and develop faults. Since data accesses follow both temporal and spatial locality, we expect a small set of blocks to be accessed most often. This is precisely the behavior that we wish to prevent. Let us further elaborate with an example. Consider a pen drive that contains songs. Most people typically do not listen to all the songs in a round robin fashion. Instead, they most of the time listen to their favorite songs. This means that a few blocks that contain their favorite songs are accessed most often and these blocks will ultimately develop faults. Hence, to maximize the lifetime of the flash device, we need to ensure that all the blocks are accessed with roughly the same frequency. This is the best case scenario, and is known as *wear leveling*.

The basic idea of wear leveling is that we define a logical address and a physical address for a flash device. A *physical address* corresponds to the address of a block within the flash device. The logical address is used by the processor and operating system to address data in the flash drive. Every flash device contains a circuit that maps logical addresses to physical addresses. Now, we need to ensure that accesses to blocks are uniformly distributed. Most flash devices have an access counter associated with each block. This counter is incremented once every P/E cycle. Once the access count for a block exceeds the access counts of other blocks by a predefined threshold, it is time to swap the contents of the frequently accessed block with another less frequently accessed block. Flash devices use a separate temporary block for implementing the swap. First, the contents of block 1 are copied to it. Subsequently, block 1 is erased, and the contents of block 2 are copied to block 1. The last step is to erase block 2, and copy the contents of the temporary block to it. Optionally, at the end, we can erase the contents of

the temporary block. By doing such periodic swaps, flash devices ensure that no single block wears out faster than others. The logical to physical block mapping needs to be updated to reflect the change.

Definition 92
A technique to ensure that no single block wears out faster than other blocks is known as wear leveling. *Most flash devices implement wear leveling by swapping the contents of a block that is frequently accessed with a block that is less frequently accessed.*

Read Disturbance

Another reliability issue in flash memories is known as *read disturbance*. If we read the contents of one page continuously, then the neighboring transistors in each NAND cell start getting programmed. This is because the control gate voltage of the neighboring transistors needs to be greater than V_T^+ such that they can pass current. Note that in this case, the voltage of the gate is not as high as the voltage that is required to program a transistor, and it also lasts for a shorter duration. Nonetheless, a few electrons do accumulate in the floating gate. After thousands of read accesses to just one transistor, the neighboring transistors start accumulating negative charge in their floating gates, and ultimately get programmed to store a 0 bit.

To mitigate this problem, we can start out with having a read counter with each page or block. If the read counter exceeds a certain threshold, then the flash controller needs to move the contents of the block to another location. Before copying the data, the new block needs to be erased. Subsequently, we transfer the contents of the old block to the new block. In the new block, all the transistors that are not programmed start out with a negligible amount of negative charge in their floating gates. As the number of read accesses to the new block increases, transistors start getting programmed. Before we reach a threshold, we need to migrate the block again.

10.5.2 Ferroelectric RAM (FeRAM)

Basic Physics

Ferroelectric RAM (or FeRAM) is a strong competitor of flash memory. It was commercialized way back in 1998, and it has been available ever since. It is in principle similar to a DRAM cell. Each FeRAM cell has one transistor and one capacitor. However, the capacitor is special, instead of using a normal dielectric material, it uses a ferroelectric dielectric. A ferroelectric dielectric is made of a ferroelectric material such as lead zirconate titanate ($PbZrO_3 + PbTiO_3$), popularly known as PZT, barium titanate ($BaTiO_3$), or strontium bismuth tantalate $SrBi_2Ta_2O_9$ (SBT).

These materials exhibit the phenomenon of ferroelectricity. Its explanation is as follows. In general, when we apply an electric field across a dielectric, the electrons get displaced towards the positive pole and the positively charged ions get displaced towards the negative pole. As a result, there is a net electrical dipole moment. This means that the center of the positively charged elements and the center of the negatively charged elements do not coincide. There is a separation between them, which results in an electric field. The product of the charge (positive or negative) with the distance is known as the *dipole moment*. *Polarization* is defined as the dipole moment per unit volume.

For most materials, the degree of polarization that is induced by an externally applied electric field is linearly proportional to it. Additionally, when the electric field is removed the polarization becomes zero. However, ferroelectric materials are an exception to this rule. As shown in Figure 10.30, there is a certain degree of hysteresis in the relationship between the applied electric field and the degree of induced polarization. Particularly, it is possible that even when the electric field is zero, the material can have an inherent polarization. Here, the degree of polarization depends on the history of how the

electric field across the medium has varied in the past, we say that there is a certain degree of hysteresis to it, which is clearly visible in Figure 10.30.

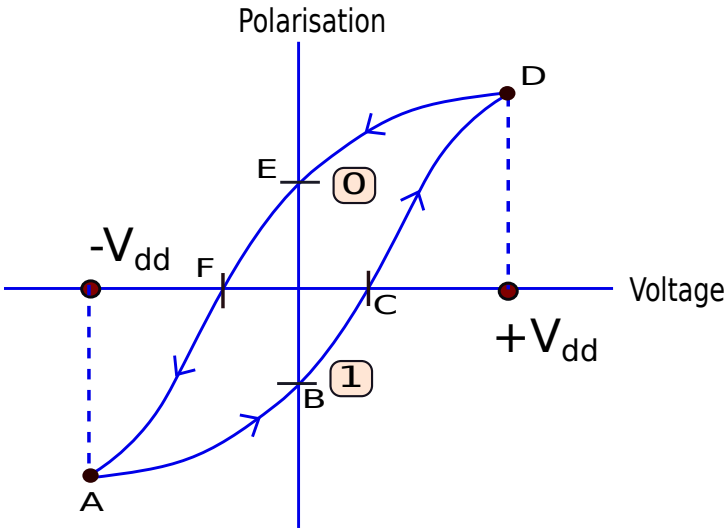

Figure 10.30: Hysteresis loop of polarization vs voltage in a ferroelectric capacitor

Consider a point of time when we are at point A (in Figure 10.30). The applied voltage is $-V_{dd}$. Then as we increase the voltage and the voltage becomes zero, we arrive at point B. Note that at this point, the applied voltage is zero, still there is an inherent polarization. Then as we increase the voltage towards $+V_{dd}$, we arrive at point C where the degree of polarization is zero, and finally when the voltage is $+V_{dd}$, we arrive at point D. Now as we decrease the voltage we do not follow the same path, rather we follow a different path. This is a characteristic of all systems that exhibit some degree of hysteresis. For example, as we reduce the voltage to 0, we start arriving at point E. At this point, there is a certain amount of inherent polarization, which the device seems to remember. Finally, with a further decrease in the voltage across the capacitor, we first arrive at point F, and then when the voltage reaches $-V_{dd}$, we arrive at point A (the same point at which we started).

The key point to note here is that there is a notion of a state associated with the dielectric material. When the electric field is zero there are two states B and E, where it can either have a positive polarization or a negative polarization: these can correspond to different logical states (0 and 1). FeRAMs use this property to store data: a negative polarization is a logical 1 (point B) and a positive polarization is a logical 0 (point E).

FeRAM Cell

Now that the basic physics has been established, the next step is to create a functioning device out of an FeRAM cell. Figure 10.31 shows the design of an FeRAM cell. Akin to a DRAM cell, each FeRAM cell has a bit line and a word line. The bit line is connected to one of the terminals of the access transistor, which is controlled by the word line. The other terminal of the access transistor is connected to the ferroelectric capacitor, which is made by sandwiching a layer of ferroelectric material with two metal electrodes typically made of platinum or iridium.

Note that here there is a major difference as compared to DRAM cells. The other terminal of the ferroelectric capacitor is connected to a plate line (PL), instead of being connected to ground. We can independently control the voltages on the bit line and the plate line. Furthermore, note that the convention is that the voltage across the capacitor is considered to be positive if the voltage on the plate

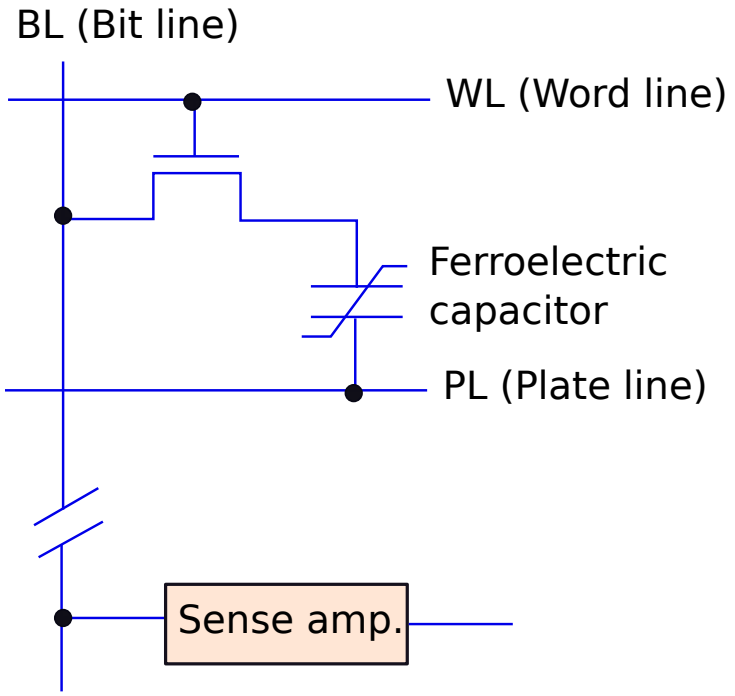

Figure 10.31: An FeRAM Cell

line is greater than the voltage on the bit line. It is necessary to have a separate plate line because we need to create both positive and negative voltages across the capacitor.

Let us now look at the basic read and write operations. We shall describe the write operation first because it is easier.

The Write Operation

To write a value, all that we need to do is set appropriate voltages on the bit line (BL) and the plate line (PL), after setting the word line. For example, when we want to write a 0, we need to ensure that the dielectric is positively polarized. This can be done by increasing the voltage of the plate line to V_{dd}, and setting $BL = 0$ V. To write a logical 1, we need to do the reverse. We need to set $BL = V_{dd}$, and $PL = 0$ V, which also means that as per our convention, the voltage across the two terminals of the capacitor is equal to $-V_{dd}$. After doing this, if we set the word line to 0, the dielectric will move to any one of the stable states: logical 0 (point E) or logical 1 (point B).

The Read Operation

The read operation is slightly more tricky. In this case, we set $PL = V_{dd}$, and set the voltage on the bit line (BL) to 0 V. After this, we enable the word line. There are two choices: either the cell stores a logical 0 or a logical 1. Assume it stores a logical 0, then it will move from point E to D, and this will increase the polarization of the capacitor. This will cause a net current outflow from the cell into the bit line, which will increase its voltage. Let this increase in voltage be ΔV_0.

Next, let us consider the other case where the cell stores a logical 1. In this case the movement will be from B to D. This will lead to a reversal in the polarization. This can happen only if there is a net current flow down the bit line, which will charge a few capacitors along the way and raise the potential

of the bit line to ΔV_1. We expect that $\Delta V_1 > \Delta V_0$ from the shape of the hysteresis curve as shown in Figure 10.30.

The sense amplifier can thus be tuned to sense a voltage that is between ΔV_0 and ΔV_1. Akin to a DRAM array, subsequent stages can buffer the logical values that were read and send them on the bus to the memory controller.

We have a special case if the cell contained a logical 1. In this case, we will move from point B to D. Once the electric field is removed, because of the nature of the hysteresis curve, we will arrive at point E, which actually corresponds to a logical 0. We thus observe that when the cell stores a logical 1, we have a destructive read. However, if the cell stores a logical 0, we move from E to D, and back to E again when the access transistor is disabled. Thus, in this case, the read is not destructive. Sadly, in the former case, when the cell stores a logical 1, it is necessary to write the value back again (similar to DRAMs).

Comparison vis-a-vis DRAM and Flash

Even though FeRAM memories have been around for a long time, they have not gained the kind of popularity that flash memories have gained. There are many reasons for this. Let us analyze some of the important ones.

The first is that flash memories use the regular silicon fabrication process to a large extent. They are not dependent on special fabrication processes or special materials. However, in this case, we need to integrate materials such as PZT or SBT in the fabrication process. This requires us to create new fabrication facilities and also these materials are associated with a lot of contamination issues. Another important factor that prevented the FeRAM technology from scaling is that researchers could not build an analog for trench capacitors or very small stacked capacitors using PZT-based dielectrics, thus preventing their further miniaturization. Moreover, it was observed that as we reduce the size of the dielectric, its ferroelectric properties diminish significantly.

The FeRAM technology however does have its advantages. Unlike DRAM arrays it does not need periodic refreshes, which helps save a lot of power. Furthermore, unlike flash where writing is a very expensive operation, in this case we can write to cells very easily and the energy difference between writes and reads is minimal. FeRAM also has much more endurance than flash memories. An FeRAM cell can support at least up to 10^{12} read/write cycles as compared to just 10^5 cycles for flash memory cells as of 2020.

10.5.3 Magnetoresistive RAM (MRAM)

MRAMs stand for magneto-resistive random access memories.

MRAM Cell

An MRAM cell has three layers: two ferromagnetic layers that are separated by a very thin layer made of an electrical insulator. One of the ferromagnetic layers is known as the *pinned layer* because the direction of its magnetization is fixed. The other layer is known as the free layer because the direction of its magnetization can be changed by applying a magnetic field. This is shown in Figure 10.32. There are thus two states of this device. If the magnetic fields of both the ferromagnetic layers are aligned, then we call this the *parallel* state, otherwise if the fields are in opposite directions, then we call this the *anti-parallel* state. Figure 10.32 shows an avatar of the device where the magnetic field lines are in the same plane as the thin film separating the ferromagnetic layers. These devices are increasingly giving way to devices where the direction of magnetization is perpendicular to the plane of the film. For the sake of simplicity, we shall describe the former approach. Note that the method of operating the cell is the same for both the approaches.

Since the insulating layer (often made from MgO) is very thin, typically 1-2 nanometers thick, a quantum mechanical effect called tunneling magnetoresistance (TMR) is seen. It is possible for electrons

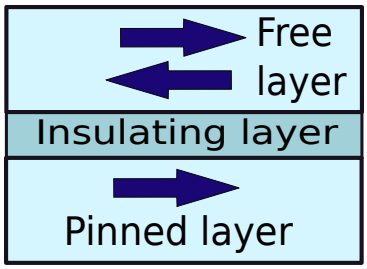

Figure 10.32: Structure of an MRAM cell

to jump from one ferromagnetic layer to the other layer, even though this is forbidden by the rules of classical physics. This means that if there is a voltage difference between the two ferromagnetic layers, a current can flow through the MRAM cell. Specifically, the current through the insulator will flow because of the TMR effect, which acts as a resistive element. The greatness of this device is that the resistance of the cell is a function of the orientation of the magnetic field of the free layer. If the cell is in the parallel state (the pinned layer and the free layer have the same direction of magnetization), then the resistance is low, and if the cell is in an anti-parallel state (opposite directions of magnetization), then the resistance is high. We assume that low resistance means a logical 1 and high resistance means a logical 0. Measuring the resistance is easy (sense the voltage with a fixed current or vice versa) and thus reading the value stored in the cell is fairly straightforward.

However, the main challenge is to create a mechanism such that we can write to the cell efficiently, which means that we need to be able to generate a magnetic field that can set the direction of the magnetic field in the free layer. The traditional approach was to create a magnetic field by passing a very high current; the main problem is that such approaches do not scale with decreasing feature sizes, and it is possible that the value stored in the nearby cells gets perturbed. Hence, a new avatar of such devices has been proposed that also relies on nanoscale effects, albeit quite differently.

STT-MRAMs

These are called spin-transfer torque (STT) devices, or STT-MRAMs. Recall from high school physics that electrons are associated with an angular momentum, which is known as the *spin*. In quantum mechanical terms, the spin can take two values: $+1/2$ and $-1/2$. Furthermore, if any charged particle like an electron has an associated angular momentum, then it also has an associated magnetic moment. In general, if we consider electric current as a stream of electrons flowing along the wire, half of them will have a positive spin ($+1/2$) and half of them will have a negative spin ($-1/2$). However, if we pass an electric current through a magnetized medium such as the pinned layer of an MRAM cell, we can produce spin-polarized current, where a majority of electrons have the same type of spin. Furthermore, when this current passes through the insulating layer and reaches the free layer, it is possible for it to transfer some of its spin (or angular momentum) to the electrons in that layer. This can flip the direction of its magnetic field, and we can thus program the memory cell. The process of transferring this angular momentum is also known as applying a *torque* to the electrons in the free layer. Hence, the name of this device is a spin-transfer torque device. In terms of both latency and power, this technique is far superior to previous approaches that use large magnetic fields to set the state of an MRAM cell. Let us delve into the details.

The Write Operation

Figure 10.33 shows the design of an MRAM cell. The free layer of an MRAM cell is connected to the bit line, and the pinned layer is connected to the *select line* via an access transistor. Let us look at

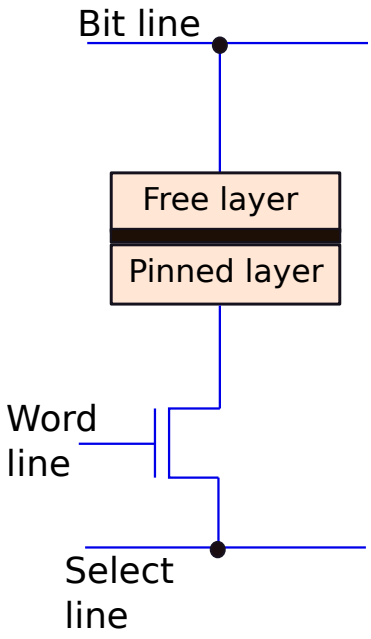

Figure 10.33: Design of an MRAM cell [Kawahara et al., 2012]

the process of programming a cell. Recall that when the cell is in the parallel (P) state (directions of magnetization are the same), we store a logical 1; when the cell is in an anti-parallel (AP) state (high resistance), we store a logical 0.

To switch from the AP to the P state, electrons need to flow from the pinned layer to the free layer. This is because as they pass through the pinned layer their spins get polarized (most of them in one direction). Subsequently, as they tunnel through the MgO layer, and enter the free layer, they transfer some of the spin torque to magnetize the free layer in the direction of the pinned layer. To achieve this, we need to have the bit line at a higher potential as compared to the select line such that current flows from the bit line to the select line.

Now to move from the P to the AP state, we pass electrons in the reverse direction – from the free layer to the pinned layer. This is achieved by setting the potential of the select line higher than the bit line. In this case, the electrons pass through the free layer first. The electrons that have the same spin direction as the direction of magnetization in the pinned layer seamlessly pass through. However, a fraction of electrons that do not have the same spin direction, bounce back. They get reflected from the MgO-pinned layer boundary. Gradually, more and more such electrons accumulate in the free layer. They transfer their torque to electrons in the free layer, and the direction of the magnetization in the free layer gets reversed. Thus, the free layer's magnetization changes its direction, and the cell enters the AP state.

Pros and Cons

The STT-MRAM cell has several advantages. The leakage current, which is the current that flows through the cell even when it is inactive is almost zero. As we shall see in Chapter 11, this is not the case in conventional SRAM and DRAM memories. For them, the leakage power is a major component of the overall power consumption. Additionally, the current requirements to read or write an STT-MRAM cell are low.

A major disadvantage of this device is that it is not as fast as a traditional DRAM cell. The reason

is that a DRAM cell relies on fast electrical switching, whereas in this case, there is an interaction of magnetic and quantum mechanical effects. For example, to write a value it is necessary to wait till the magnetic field in the free layer reverses and reaches a certain strength. Similarly, to read a value it is necessary to sense the resistance of the cell, where the main issue is that the difference in the resistances of the two states might be as low as 20%. This requires a sophisticated sensing circuit. In addition, thermal stability of such memory cells is an issue, particularly, when there are large variations in die temperature.

Notwithstanding such concerns, STT-MRAMs as of today are considered fairly mature technology; they have relatively fast read and write times and a very high endurance. There are challenges in large-scale production (as of 2020), however it is expected that in the coming years many of these issues will be solved.

10.5.4 Phase Change Memory (PCM)

Basic Physics

Another popular memory technology is Phase Change Memory (known as *PCM*). It encodes information in the phase of a material, which is typically a chalcogenide glass. A *chalcogenide glass* is an amorphous solid that is made of one or more chalcogen elements such as sulfur, selenium, or tellurium. One of the most popular materials for making phase change memory is $GeSbTe$ (germanium-antimony-tellurium) abbreviated as GST.

Unlike the memory cells that we have seen up till now, the PCM memory cell changes its state based on its temperature. It has two states: amorphous and crystalline. In the amorphous state, it has a high resistance (logical 0), and in the crystalline state it has a low resistance (logical 1). The reason that a material made of GST has a low resistance in the crystalline state is because there is a lot of order in the structure and there are a lot of free electrons to carry current. The situation in the amorphous state is the reverse, hence it has a high resistance.

Given that the PCM cell has two states, we need a method to switch between the states. All previous technologies have relied on the injection of electric current to either deposit charge, or induce magnetism in a material. A PCM cell also relies on current injection, however the current is required to change the temperature such that the material melts and then either transforms to the amorphous state or to the crystalline state. Whenever we apply a short and high amplitude current pulse, the material melts quickly and then rapidly transitions to the amorphous state. This sets the value stored in the cell to a logical 0, and *resets it*. On the other hand, if we apply a low amplitude and long current pulse, then the material crystallizes. The cell stores a logical 1. This current is known as the *set current*, which *sets* the value stored in the cell.

Let us quickly deduce the properties of a PCM cell from its basic physics. Given that it stores data in the phase of the material, it can hold its data for a very long time. Commercially available PCM devices as of 2020 can hold their data for at least 10 years. In addition, it allows us to directly overwrite the value of a cell, quite unlike flash memory where we erase data at the level of blocks. Finally, unlike DRAM and FeRAM cells, reads are not destructive. The structure of a basic cell is shown in Figure 10.34.

Several improvements have been proposed to the basic PCM cell. For example, it is possible to store multiple bits in a single cell. Between a fully crystalline and a fully amorphous state, we can have two more partially crystalline states, and thus we can realize a total of four states, which are sufficient to encode two Boolean bits. This further increases the density of storage.

Read and Write Operations

A PCM cell is typically a 1T1R (1 transistor, 1 resistor device) as shown in Figure 10.35. The access transistor (controlled by the word line) is connected to one end of the cell, and the other end of the cell is connected to the bit line.

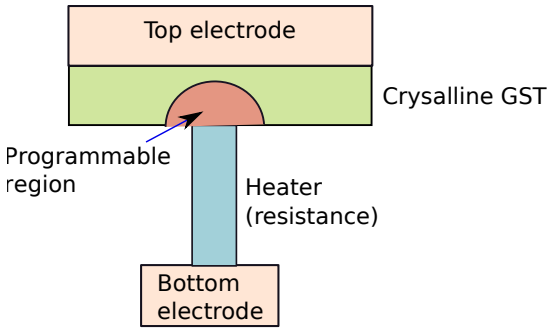

Figure 10.34: Design of a PCM cell

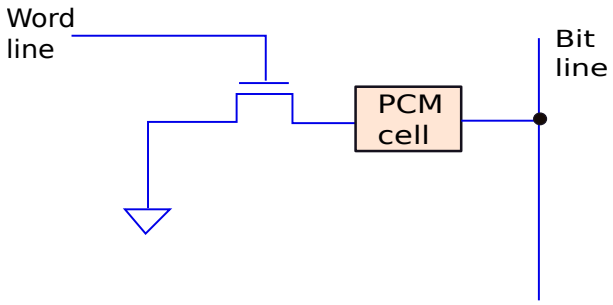

Figure 10.35: A PCM cell connected to the word line and bit line

The read process is simple. We precharge the bit line, and then enable the access transistor. If it is in the low resistance state, then the bit line discharges quickly. This can be sensed using sense amplifiers. Conversely, if it is in the high resistance state, then the voltage after a certain period of time is much higher.

The write operation is dependent on whether we are setting (writing a 1), or resetting the cell (writing a 0). If we are writing a logical 1, then we need to move the state of the chalcogenide material to the low-resistive crystalline state using a low current for a relatively long time. However, if we are resetting the state of the cell (setting to a logical 0), then we need to provide a short high-amplitude current pulse. Finally, note that unlike DRAMs, reads in PCM are not destructive, and there is no need for restoring the state.

Dealing with Slow Writes

In PCM, writes are slow. It takes time to change the state of the chalcogenide material. Particularly, if we are changing the state from the amorphous to the crystalline state, then we need to supply a write current for a long duration of time. During this time the bank cannot be used for servicing any other request. Often writes are not on the critical path, however reads are very often on the critical path.

At the level of the memory controller, we can reduce the priority of writes, implement a DRAM based write cache, and discard ineffectual writes. An ineffectual write operation has no effect. For example, if we are successively writing two values to the same location, then the first write operation is ineffectual.

There are two more sophisticated methods of dealing with slow write operations: write cancellation and write pausing. Assume that a write operation is in progress, and a high-priority read arrives. We can immediately cancel the write in the middle of the operation by deasserting the write enable signal of the PCM device. This will stop the write midway, and the state of the cell will be in a non-deterministic

state. This is not necessarily a problem because we always know what we were writing, and this value can be stored in a write buffer till the write is finally completed at a later point of time. This allows us to immediately service the read request.

Another method in this space is write pausing. In modern PCM cells, the time of a write operation becomes more and more non-deterministic over time because of the varying rates of crystallization of the devices based on how much they have been used. Moreover, if we are using a PCM cell that can store multiple logic levels, this process is even more complicated. Hence, the conventional approach to deal with this problem is that we perform a write iteratively. For one iteration, we apply the write current, and then *verify* if the value has been correctly written. If it has not been correctly written, then we start the next iteration, and keep doing this till the value is correctly written. Let us say that when we have completed an iteration, a high-priority read arrives. Then we can *pause* the next iteration of the write, service the read, and then come back and complete the rest of the iterations for the write operation. This allows us to service requests with minimal delay.

Reliability and Endurance

High write currents and thermal cycling in the chalcogenide material reduce the reliability of the PCM device significantly. Similar to flash memory, the standard approach for dealing with such problems is *wear leveling*. This means that we ensure that all the banks in the PCM array are equally accessed. We need to create a mapping between the CPU generated addresses, and the internal PCM addresses, and keep updating this mapping to ensure that all the PCM blocks are roughly equally accessed. Note that we care about writes more than reads, because writes are more intense operations in PCM memory.

An important approach in the space is start-gap wear leveling. In this technique, we periodically move each block L to another block L', where there is a one-to-one mapping between them. It can be proven that if we do this, all the locations are roughly equally accessed. A simple function can be $L' = L + 1$. It is possible to work out more complicated functions that are based on cryptographic primitives and provide better guarantees.

10.5.5 Resistive RAM (ReRAM)

Basic Physics

ReRAMs or resistive RAMs rely on the *resistive switching phenomenon*. This phenomenon is observed in several metal oxides (such as NiO or TiO_x), where the resistance is a function of the history of the voltage applied across the material. To create a nonvolatile memory, we need to have two physical states that are stable. For such materials we can have a high resistance state (HRS) and a low resistance state (LRS).

Similar to the other nonvolatile memory cells, a ReRAM cell also contains a metal oxide layer sandwiched between two electrodes (see Figure 10.36). The metal oxide layer is typically a transition metal oxide or titanium nitride, and the electrodes are made of Pt (Platinum), Ir (Iridium), or Ag (silver). Depending upon the history of the voltage that is applied between the electrodes, the resistance varies. The I-V curve of such devices typically shows a certain degree of hysteresis, this is why we observe two distinct physical states (explained later). The HRS state corresponds to a logical 0, and the LRS state corresponds to a logical 1. The process of changing the state from HRS to LRS ($0 \rightarrow 1$) is known as the *set* process. The reverse process ($1 \rightarrow 0$) is known as the *reset* process.

Reading such a cell is easy; we just need to apply a small voltage and measure the current. Let us now discuss the two main types of ReRAM cells: Redox ReRAMs, and CBRAMs.

Redox ReRAMs

The most important charge transport mechanism in such ReRAMs is the *filamentary conduction mechanism*. As per this mechanism, when we apply a large voltage across the electrodes, tiny conducting

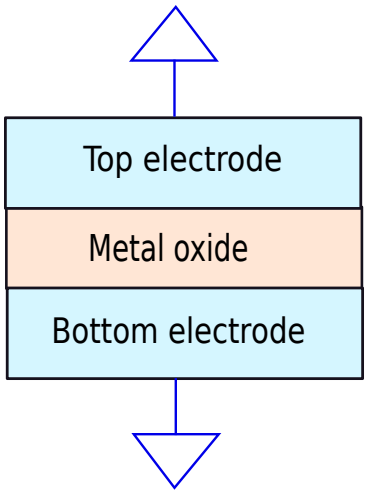

Figure 10.36: Basic ReRAM cell

filaments form between the two electrodes; these filaments can carry current, and thus the cell enters the low resistance state. If we can somehow destroy these filaments, the cell will enter the high resistance state. Let us elaborate.

In a Redox ReRAM, the mechanism is as follows. Let us assume that the material in the middle metal oxide layer is of the form VO_2. Here, V is a transition metal. Similar to electrons and holes, here also we have two kinds of charge carriers: O^{2-} ions and oxygen vacancies (V_o). An oxygen vacancy is similar to a hole in device physics, and is a positively charged quantity. Let us now look at the different phases of a Redox ReRAM.

At the beginning, the density of oxygen vacancies is low. Then each cell needs to go through the forming phase. In this case, we apply a large potential across the electrodes. Dielectric breakdown takes place and the negatively charged O^{2-} ions move towards the anode. Meanwhile, the oxygen vacancies stay back in the metal oxide layer. Near the anode we have an excess of negatively charged oxygen ions. If the material of the anode electrode reacts with oxygen ions then an oxide layer forms on top of the anode. These ions are thus effectively removed from the metal oxide layer.

The oxygen vacancies in the metal oxide layers align themselves along the electric field and form a *conductive filament*. This is a conducting path that can carry current, and thus the cell enters the low resistance state (LRS). This is shown in Figure 10.37. This is the *set* process. Typically, the oxide layer at the anode is amorphous in nature and has large-sized grains. The filaments form at the grain boundaries.

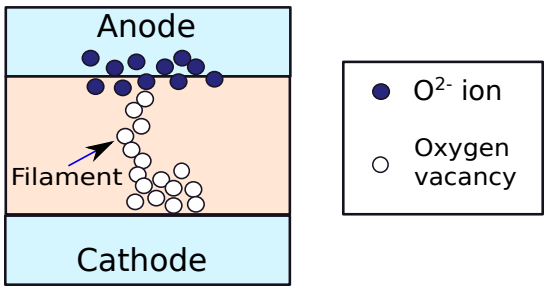

Figure 10.37: Filament in a Redox ReRAM cell

Let us now look at the *reset* process. In this case, we need to break the filaments that have been created such that the cell enters the high resistance state. There are two mechanisms in this space. We can either have unipolar switching or bipolar switching. In unipolar switching only the magnitude of the voltage is important, whereas in bipolar switching the sign of the voltage (positive or negative) is also important. In the case of unipolar switching, we send a high reset current through the conducting filament. This causes localized heating (Joule heating). The O^{2-} ions that are trapped at the anode get displaced, and they combine with the oxygen vacancies in the filament effectively rupturing it.

In the case of bipolar switching we have a smaller reset current. We apply a negative voltage at the anode, which pushes the O^{2-} ions towards the metal oxide layer. They combine with oxygen vacancies and rupture the filament. The cell thus enters the high resistance state (logical 0). The next time that we want to set the value of the cell, we apply a positive voltage at the anode again. This attracts the negatively charged oxygen ions leaving oxygen vacancies in the metal oxide layer. The filament forms again. Given that we have a series of Redox (oxidation-reduction) reactions, this cell is known as a *Redox ReRAM*.

Let us quickly summarize what we have learned in Point 20.

Important Point 20

- *In a Redox ReRAM, to enter the low resistance state, we apply a positive voltage to the cell. The negatively charged oxygen ions migrate towards the anode and get deposited over there. In some cases, they might also react with the material in the anode electrode and form an oxide layer. In the metal oxide layer, the remaining oxygen vacancies align themselves along the electric field and form a conductive filament.*
- *To rupture this filament there are two mechanisms: unipolar switching and bipolar switching.*
- *In unipolar switching we apply a large positive voltage to the anode. Because of the large current flow and resultant Joule heating, some oxygen ions get dislodged and move towards the conducting filament. They combine with oxygen vacancies and break the filaments. The cell thus enters the high resistance state.*
- *In bipolar switching, we apply a negative voltage at the anode; this sends back oxygen ions back to the metal oxide layer. There they get combined with oxygen vacancies in the filament. The filament thus gets ruptured.*

CBRAMs

A *conductive bridging RAM* or a CBRAM is another type of ReRAM cell that also relies on the filamentary switching mechanism. The basic structure of this cell is the same as that of the Redox ReRAM cell. We have two electrodes and a thin electrolyte layer in between.

However, in this case, the electrodes play a very significant role and provide the material for the filament. One of the electrodes is called an electrochemically active electrode and is made of Ag (silver), Cu (copper), or Ni (nickel). The other electrode is called the inert electrode and is made of Pt (platinum), or Ir (Iridium). In between both the electrodes, we have a thin layer of an electrolyte: Ge_xS_y, SiO_2, TiO_2, Ta_2O_5, or ZrO_2.

The process for programming the cell is as follows. If we apply a highly positive voltage to the active electrode (typically Ag), it dissolves into the electrolyte. The positively charged silver ions(Ag^+) drift into the thin electrolyte and get pushed towards the inert electrode by the electric field. Some of them finally reach the inert electrode (cathode), absorb an electron ($Ag^+ + e^-$) and get deposited on

the surface of the inert electrode. Gradually a channel of Ag atoms forms between the anode and the cathode (see Figure 10.38). Once this is fully formed, this becomes a conducting filament, which can carry current. The cell subsequently enters the low resistance state. Even after turning off the voltage source, the filaments remain. This state of the cell thus represents a logical 1.

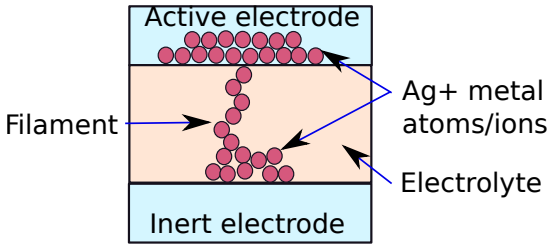

Figure 10.38: Filament in a CBRAM cell

To reset the cell, we need to apply a negative voltage to the active electrode (bipolar switching). Positively charged silver ions (Ag^+) migrate towards the active electrode, absorb an electron, and get deposited on the active electrode. This resets the state of the cell. Subsequently, the filament gets ruptured and thus there is no conductive path between the two electrodes; the cell enters the high resistance state (logical 0).

Applications in Neural Networks

Resistive devices have an interesting property – they can work as a multiplier. In a modern ReRAM, we just don't have one high resistance state, and one low resistance state, we can create several intermediate states with different levels of resistance. If the conductance of a ReRAM device is G, then we have $VG = I$ from the Ohm's law. This means that for a variable voltage, this device works as a multiplier.

Moreover, if we have an array of ReRAM cells, then we can compute a dot product by adding the currents (according to Kirchhoff's law). We shall see in Chapter 14 that this is the most performance-sensitive operation in deep neural networks. We can directly implement such operations in ReRAM based hardware and achieve significant speedups.

10.5.6 3D and Embedded Memory Technologies

Typically, the memory system proves to be the bottleneck for many high-bandwidth applications. Hence, there is a need to adopt futuristic memory technologies. A few of the technologies that stand out in this regard are 3D memory technologies such as High Bandwidth Memory (HBM) and Hybrid Memory Cubes (HMC). They use DRAM devices; however, instead of a single 2D layer, they are composed of multiple 2D layers stacked over each other.

The underpinnings of all these memory technologies are the same. The core idea is that we have a 3D stack of DRAM memory arrays. The memory arrays are connected to each other via TSVs (trans-silicon vias) or microbumps that form high-bandwidth connections across the layers. Microbumps are small metallic structures that are used to connect to the layer below. Furthermore, we divide each layer into a set of blocks, and we consider a column of blocks to be one unit. It is known as a *vault*. There are several ways of integrating such 3D memory stacks. The most common method is a configuration where the silicon die and the 3D memory are placed side by side on the motherboard. They are connected via a high-bandwidth connection known as an *interposer*, which typically consists of several 128-bit wide links to connect the 3D DRAM memory and the chip.

The reason for the high bandwidth is two-fold. The first is that we can read many bits in parallel from each vault. The second is that because of the proximity between the CPU chip and the 3D memory chip, we can afford a very wide bus.

Another technology that allows for high-bandwidth connections is embedded DRAM (eDRAM). Here, the DRAM module is integrated into the same die as the chip or is present in a multi-chip module configuration (within the same package). There are process challenges because we typically use different processes to fabricate logic and to fabricate DRAM. Hence, 2.5D integration, where we fabricate separate dies for the processor and the DRAM, respectively, and place them in the same package is considered to be a more feasible and practical solution. Here, of course, there is a need to create a high-bandwidth network within the package, something similar to an interposer.

10.6 Roofline Model

In this chapter we have introduced many techniques to optimize the memory system. They need to work in conjunction with the techniques that we proposed to optimize the design of the OOO pipeline. Let us now look at modeling and understanding the performance of such systems. We have discussed basic performance equations in Section 2.2 and Section 7.1.6. In this section let us present a simple method for modeling the performance of such systems using simple diagrams. We shall introduce the *Roofline* model [Williams et al., 2009] in this section.

10.6.1 Overview

It is possible to propose many models that relate the DRAM throughput with the performance. Many such models have been proposed in the literature that extend the performance equation and incorporate the effect of different kinds of off-chip memory technologies. The main drawbacks of all of these models is that they are dependent on the workload and use too many constants. This makes the models hard to use, and they are also not very intuitive. The quest for such models ended with the Roofline model that proposes a simple technique to find out the limitations of a workload – is it memory bound or compute bound or both?

Using this model it is possible to understand how the peak off-chip memory bandwidth and peak performance interact. It is also possible to find out how much a given set of workloads can be optimized till they reach the limits imposed on them by the system. Furthermore, with this model, it is possible to study the effects of different performance enhancing optimizations in the pipeline and the memory system.

At the outset, let us define three terms.

Operational Intensity This term captures the crux of the model – it is defined as the average number of floating point operations a processor can perform for every single byte read from off-chip memory. In other words, it represents the processing power of the CPU and the efficiency of the caches. An increased operational intensity means that the CPU can very effectively make use of the information that it reads from main memory. We typically measure this using performance counters that give us an estimate of the number of floating point operations and the number of off-chip memory accesses.

Memory Bandwidth For a given system (processor + off-chip memory), this quantity represents the observed off-chip memory bandwidth. This includes the effect of caching, memory controller optimizations, and prefetching. It is measured with the help of performance counters.

Performance The performance in this case is defined as the number of arithmetic operations performed per second. Typically, in the high-performance computing world, it is measured in the unit of FLOPS (floating point operations per second). It is possible to use other measures as well such as the number of integer operations per second; however, this is not very common and is not relevant in a high-performance computing context.

Hence, we shall go with FLOPS. Note that we shall use the term "FLOP" (floating point operation) to indicate a single floating point operation, the term "FLOPs" as its plural, and the term "FLOPS"

to denote the number of floating point operations per second. The performance of a program is measured using dedicated performance counters to compute the number of floating point operations per second (FLOPS).

It is very easy to relate these three quantities – operational intensity, memory bandwidth, and the performance. The relation is shown in Equation 10.1. This follows from the definition of these quantities.

$$\underbrace{\frac{FLOPs}{second}}_{Performance} = \underbrace{\frac{bytes}{second}}_{Memory\ bandwidth} \times \underbrace{\frac{FLOPs}{byte}}_{Operational\ intensity} \tag{10.1}$$

We thus observe that the performance at any point of time is a product of the memory bandwidth and the operational intensity. The performance and the operational intensity can be varied by changing the benchmark and by making architectural optimizations. Hence, let us plot a graph where the performance is on the y-axis and the operational intensity is on the x-axis. In this graph, let us consider all the points that have a constant memory bandwidth B.

The equation for this line is $y = Bx$ (from Equation 10.1). This is a line with constant slope B and passes through the origin. For different memory bandwidths we shall have lines with different slopes as shown in Figure 10.39. We can make improvements to Figure 10.39 to make it look more intuitive. Let us plot the same data in the log-log scale.

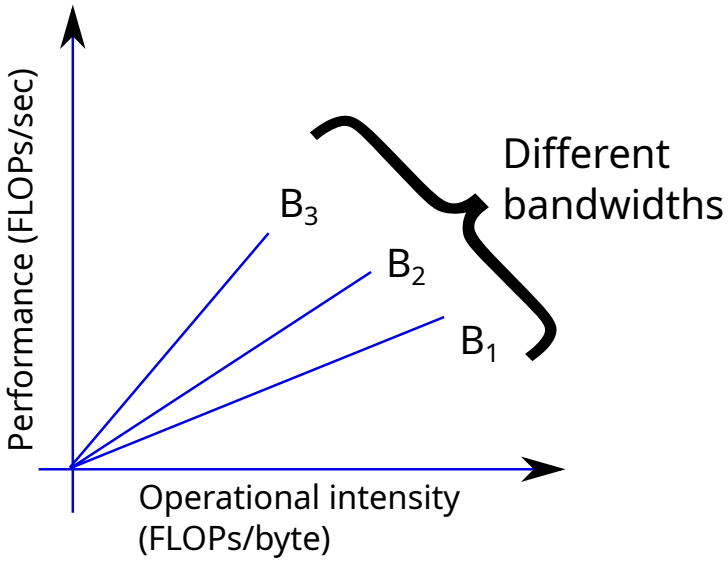

Figure 10.39: Different memory bandwidths

We have:

$$\begin{aligned} & y = Bx \\ \Rightarrow & log(y) = log(x) + log(B) \end{aligned} \tag{10.2}$$

In this case, all the constant-bandwidth lines have a slope of 45°. They are all parallel lines. This is shown in Figure 10.40. Points P and Q require a memory bandwidth that is less than B_1, and points R and S require a memory bandwidth that is more than B_3. Finally, note that for representing a line corresponding to another memory bandwidth, B', we just need to draw a line at 45°. If $B' < B$, the new line for bandwidth B' will be below the line for B. Otherwise, it will be above it.

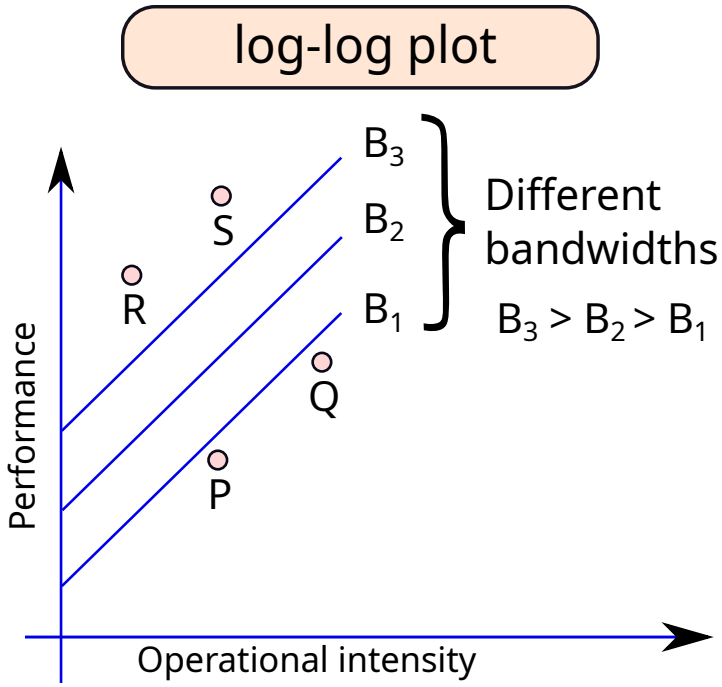

Figure 10.40: Log-log plot of performance vs operational intensity

10.6.2 Adding Ceilings

Now that we have explained the justification for the log-log plot, and also explained how to draw lines that contain all the points that need the same memory bandwidth, let us proceed to add computation ceilings.

Consider the peak performance (measured in terms of FLOPS). There are two ways to measure the peak performance in a system. First, we compute the theoretical maximum by looking at the processor's data sheets, or we write dedicated micro-benchmarks to stress the processor to the maximum limit. We can then find the peak performance using dedicated performance counters.

The peak performance can be represented as a horizontal line in the log-log plot. It is not possible to exceed the peak performance unless we change the configuration of the system, i.e., enable new architectural optimizations or increase the frequency.

On similar lines, we can define the peak memory bandwidth. It is a 45° line in the log-log plot. If we consider both of these constraints (also known as ceilings), then we have a trapezoidal region in which any workload can lie. It will always use less memory bandwidth than the maximum, and its computational throughput will be limited by the peak performance. This is shown in Figure 10.41. Since these ceilings establish limits on the performance and memory bandwidth, they are also known as *Rooflines*. Such a diagram is known as a *Roofline diagram*.

We can have several ceilings corresponding to different architectural and compiler configurations. If we consider the memory bandwidth, we can have different 45° lines corresponding to different memory system optimizations. Figure 10.42 shows three different lines for different configurations: theoretical-maximum, with-prefetching, and without-prefetching. All of these are memory bandwidth ceilings.

Similarly, we can add computational ceilings. These are parallel horizontal lines on the log-log plot. Examples of the computational ceilings are as follows. (1) The peak performance for a default configuration without SIMD instructions such as Intel SSE, (2) performance with SIMD instructions, and (3) the peak theoretical computational capacity. These are shown in Figure 10.42 . We thus observe

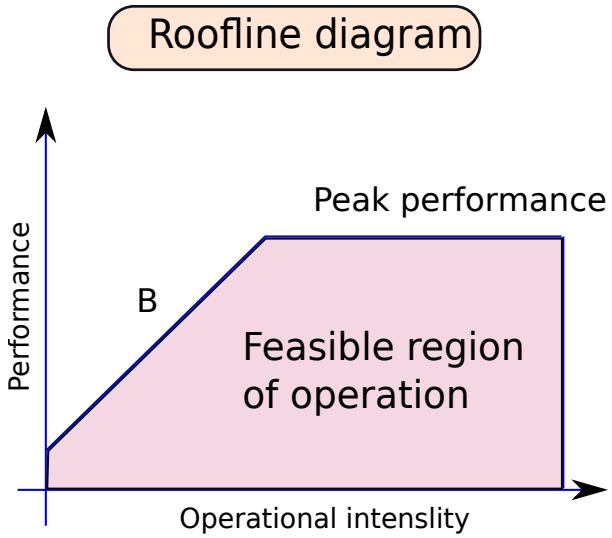

Figure 10.41: Performance vs operational intensity (with Rooflines)

different feasible regions of operation for different kinds of ceilings.

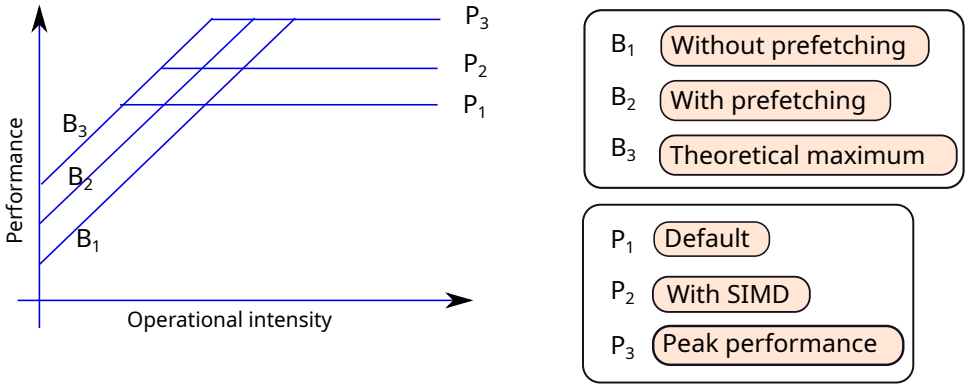

Figure 10.42: Roofline diagram with many ceilings

10.6.3 Uses of the Roofline Model

The Roofline model is a very simple tool to reason about the bottlenecks in an application. Let us consider Figure 10.43. In this figure we show several points: P, Q, and R. Point P is close to the memory bandwidth ceiling. In this case, we cannot get additional performance without increasing the memory bandwidth. Hence, any additional investment should be made in increasing the available memory bandwidth. In comparison, point Q is close to the performance ceiling. This means that it has saturated all the computational power, and memory bandwidth is not an issue because this design is not using all the bandwidth that it can use. In this case, we need to enable newer architectural optimizations for increasing the computational power. Finally, let us consider point R, which is far away from both the memory bandwidth and performance ceilings. In this case, the design can be modified to use far more memory bandwidth, and much more computational capacity as compared to the current

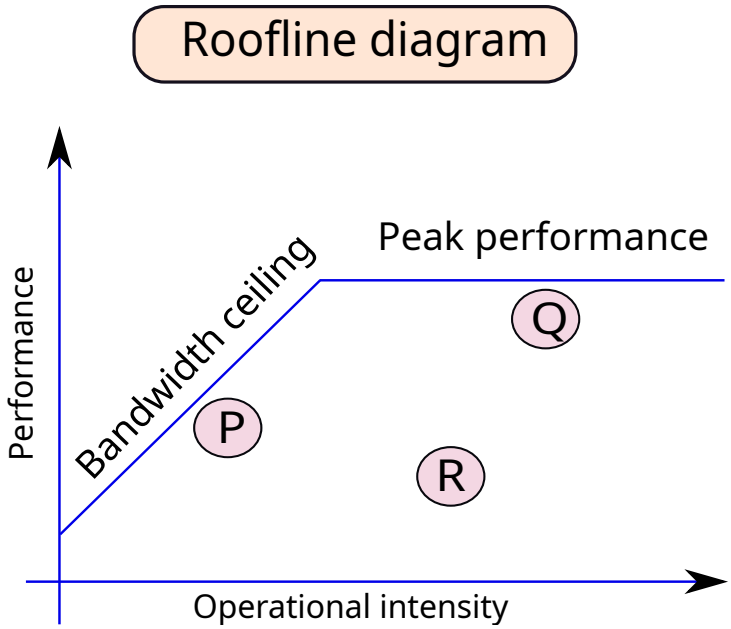

Figure 10.43: Three different operating points in the Roofline diagram

implementation.

We thus see from these three examples that given any workload or architectural configuration, we can easily find out the bottlenecks by finding the proximity of the point to different ceilings. We can use these models in numerous ways. For example, it can tell us that given a certain value of the operational intensity, what is the minimum memory bandwidth that is needed to sustain a given performance.

Additionally, we can also define a lower threshold for the performance, which means that we are guaranteed a certain level of performance. Now between the upper and lower ceilings, we can reason about the bandwidth that is required to sustain a given level of performance. Researchers have proposed numerous extensions to this model to include different kinds of additional effects, notably energy [Choi et al., 2013].

10.7 Summary and Further Reading

10.7.1 Summary

Summary 9

1. *A typical DRAM cell is very simple; it has a single transistor and a single capacitor.*
2. *Two kinds of capacitors are used to create modern DRAM cells: a trench capacitor and a stacked capacitor.*
3. *Akin to SRAM cells, we can create an array of DRAM cells. The major difference is that in this case, the sense amplifiers appear before the column multiplexer/demultiplexer. The sense amplifiers are also designed differently. They are designed to also buffer data for the entire row (page) of cells, and drive the voltages of the bit lines.*
4. *It is typically necessary to split each bit line into multiple segments. There are two architectures in this space: open bit line array architecture and folded bit line array architecture. The latter design is more tolerant to noise, at the cost of lower storage density.*
5. *DRAM reads are* destructive *in the sense that once a row is read, the contents of the cells that store a logical 1 are destroyed. Hence, it is necessary to restore their values. Since the contents of the row are stored in the array of sense amplifiers (row buffer), if subsequent reads are to the same row, then their values can be supplied from the row buffer.*
6. *Each DRAM cell can maintain its state for at the most 32 to 64 ms. The charge stored across the capacitor gradually leaks out. Hence, it is necessary to periodically refresh (read and then write) every single cell in the DRAM array.*
7. *The earliest versions of DRAM used asynchronous transfer. The row address was synchronized with the $\overline{RAS}$ strobe signal and the column address was synchronized with the $\overline{CAS}$ strobe signal. Even though pure asynchronous transfer is a generic solution; however, it has low performance and scalability.*
8. *Given that we often fetch a sequence of bytes from DRAM memory, a more efficient method of transfer is the fast page mode (FPM). After sending a row address, the controller sends a series of column addresses for that row. We read the sequence of bits corresponding to the column addresses. A shortcoming of this method is that we wait to receive the data and then only we send the next column address.*
9. *The next generation of DRAM technology was called extended data out (EDO). This overcomes the shortcoming of the FPM technology, by overlapping the transmission of data and column addresses. To provide the timing for the data signals, an additional data strobe signal $\overline{DQS}$ is used*
10. *EDO was succeeded by the burst extended data out (BEDO) technology. In this case, for a single column address, we read multiple bits – essentially prefetch the next few bits. This reduces the need to send additional column addresses.*
11. *All of these technologies have been superseded by synchronous DRAM technologies. The DRAM devices have a clock that is synchronized with the clock of the memory controller prior to a message transmission.*

12. *The command and address buses are connected to every single device; this increases the capacitive loading on them. In contrast, the data buses are only connected to a subset of DRAM devices. Hence, they are faster. To leverage this, the double data rate (DDR) memory was proposed where we transfer data at both the edges of the clock.*

13. *There are multiple DDR generations: DDR1, DDR2, DDR3, and DDR4. Over the generations, there has been an increase in the bus frequency and storage density, whereas the supply voltage has reduced.*

14. *The topology of a DDR4 memory is as follows.*

    (a) *The CPU is connected to memory modules via a set of channels. Each channel is a set of copper wires that are used to transmit memory addresses, data, and commands. Each channel has its dedicated memory controller that is co-located with the cores and cache banks.*

    (b) *Several DIMMs (dual inline memory modules) are connected to each channel.*

    (c) *Each DIMM is divided into ranks, where each rank is a set of DRAM devices (chips). All the devices in each rank operate in lockstep. They perceive the same amount of clock skew and signal delay.*

    (d) *Each device has multiple banks, where each bank can operate independently of the others. In modern DRAM devices, these banks are organized into bank groups.*

    (e) *When sending a command, we specify the rank, the bank group id, and the bank id. All the banks across the devices in the rank, with the specified bank group id and bank id, get activated (one bank per device).*

    (f) *Each bank consists of multiple memory arrays that are always accessed together. Each array has a set of rows and columns.*

    (g) *We can either read 1 bit (single column) at once, or read multiple bits. If the latter is the case, and we read n contiguous columns, then the prefetch length or prefetch width is said to be n.*

15. *The DDR4 protocol is associated with different timing parameters. Some major parameters are as follows.*

Parameter	*Mnemonic*
Row cycle time	tRC
ACT to PRE command period	$tRAS$
PRE to ACT command period	tRP
ACT to internal read or write	$tRCD$
CAS latency	CL
CAS write latency	CWL
Column to column delay (same bank group)	$tCCD_L$
Column to column delay (different bank group)	$tCCD_S$
Row to row delay (same bank group, 1Kb page size)	$tRRD_L$
Row to row delay (different bank group, 1Kb page size)	$tRRD_S$
Four-bank activation window (1Kb page size)	$tFAW$
Write to read latency (same bank group)	$tWTR_L$
Write to read latency (different bank group)	$tWTR_S$
IntREAD to PRE command period	$tRTP$
Write recovery time	tWR

16. *The memory controller schedules and orchestrates all the memory accesses. It has the following components:*

(a) *A high-level transaction scheduler that reorders read and write operations. It also has a write cache that services later reads.*

(b) *An address mapping engine that maps the physical address to internal DRAM addresses.*

(c) *A DRAM command generator. There are two broad strategies. Either we can keep a row open after it has been accessed once such that later accesses to the row can read or write to the row quickly using the row buffer. This is known as the open page access policy. The other option is to immediately close the row and precharge it. This is the closed page access policy.*

(d) *A command scheduler that chooses DRAM commands from multiple bank queues. The commands are scheduled based on fairness criteria, priorities (e.g. refresh), and the timing requirements of the DRAM devices.*

17. *Nonvolatile memories that maintain their state even after the system is powered off are becoming increasingly popular. It is a storage technology that has replaced hard disk drives in almost all mobile and hand held devices. The main types of nonvolatile memories are as follows:*

Flash memory *Such a memory uses a novel transistor that traps charge in an additional gate called the* floating gate. *The presence and absence of charge in the floating gate represents its two permanent states. Flash memory is easy to manufacture and reads are fast. Sadly, writes are slow because to write a page, it is necessary to first erase an entire set of pages (block). Moreover, it has a relatively low shelf life because it can tolerate roughly* 10^5 *to* 10^6 *program-erase cycles.*

FeRAM *FeRAM uses a ferroelectric dielectric in the capacitor, as opposed to using a normal dielectric. This dielectric has two polarization states when no voltage is applied across the capacitor. To switch between these two states, we need to change the direction of the potential across the parallel plates of the capacitor. Similar to a DRAM, the reads are destructive. However, both reads and writes are very fast and the endurance is much more than flash memory.*

STT-MRAM *In an STT-MRAM cell we have two magnetic layers: pinned layer (fixed direction of magnetization) and the free layer (direction of magnetization can change). They are separated by a thin film made of* MgO*. This cell has two permanent states based on the directions of magnetization of the pinned and free layers: parallel (low resistance) and anti-parallel (high resistance). Reading is very fast because we just need to sense the resistance of the cell. However, writing is a slower process because electrons need to transfer their spin torque to the electrons in the free layer and appropriately change their direction of magnetization. Unlike flash memory, we can access individual words and the leakage current is negligible.*

PCM *PCM (phase change memory) relies on the state of a chalcogenide material: amorphous or crystalline. If we apply a large current quickly, then because of Joule heating, the material melts and enters the amorphous state. This state has a high resistance. However, if we apply a relatively lower current for a longer time, then the material crystallizes and this state is associated with a lower resistance. PCM memory has fast reads and slow writes. Two methods to deal with slow writes are write cancellation and write pausing, where the process of writing a value is terminated midway to give way to scheduling read operations. Endurance is an issue, hence there is a need to ensure that all the locations are equally accessed.*

ReRAM *Resistive RAM (ReRAM) is a family of technologies where the instantaneous resistance of the cell depends upon the history of voltages applied to the terminals of the cell.*

In Redox ReRAMs, a conductive channel forms in an oxide layer sandwiched between two electrodes when a positive voltage is applied to the anode. This conductive channel is a filament that is made of oxygen vacancies after oxygen ions migrate to the anode. This conductive channel can be ruptured by either applying a large voltage (unipolar switching), or reversing the voltage across the cell (bipolar switching). On similar lines, a CBRAM also forms a conductive filament of Ag^+ ions that migrate towards the cathode. Off late, ReRAMs are increasingly being used to implement neural networks because they basically work as multipliers.

18. *The Roofline model is used to study the limits of performance in a system. It is a log-log plot with the operational intensity (FLOPs/byte) on the x-axis and the performance (FLOPs/sec) on the y-axis. A 45° line represents a set of points that have a fixed memory bandwidth, and a line parallel to the y-axis represents constant performance. Using these two lines – memory bandwidth and performance – we can define a feasible region of operation, and study the effect of different architectural optimizations.*

10.7.2 Further Reading

The references that we have used for the topics on DRAMs are as follows: the book by Jacob, Ng, and Wang [Jacob et al., 2007], the design of the DRAMsim simulator [Wang et al., 2005], and the JEDEC standards [JEDEC Solid State Technology Association, 2003, JEDEC Solid State Technology Association, 2008a, JEDEC Solid State Technology Association, 2008b, JEDEC Solid State Technology Association, 2020]. We have used the same symbol names as the JEDEC standards. Readers can consult them for a deeper understanding of technologies related to DRAMs particularly the DDR4 protocol that we have described in great detail.

The following references [Yu and Chen, 2016, Chen, 2016, Feng et al., 2010] give an overview of NVM technologies and future directions. For understanding the physics and the operation of FeRAMs, readers can refer to the FeRAM guide book [Fujitsu Semiconductor Limited, 2010] from Fujitsu. A thorough description of spin transfer torque mechanisms is provided by Khvalkovskiy et al. [Khvalkovskiy et al., 2013], Kawahara et al. [Kawahara et al., 2012], and Apalkov et al. [Apalkov et al., 2013]. For phase change memories, we would recommend the seminal paper by Lee et al. [Lee et al., 2009] and the e-book by Qureshi et al. [Qureshi et al., 2011]. Finally, for ReRAMs the following references [Wouters, 2009, Akinaga and Shima, 2012, Yu, 2016, Wang et al., 2018] will prove to be useful.

Exercises

Ex. 1 — Why is the sense amplifier placed after the column mux/demux in SRAMs? In comparison, why is it placed before, i.e., between the array and the column mux/demux in DRAMs?

Ex. 2 — Compare the open and folded bit line array architectures.

Ex. 3 — Calculate the refresh rate for a DRAM cell with a capacitor of capacitance 1 fF and a transistor whose leakage current is 1 pA . Assume that the voltage across a fully charged capacitor is 1.5 V and the cell needs refreshing before the voltage drops below 0.75 V.

Ex. 4 — Define a DRAM *rank*. Why is it required?

* **Ex. 5** — Why do we typically avoid multi-ported DRAM arrays? Furthermore, why do we typically access the DRAM array at the level of single columns, where a column usually stores a single bit.

Ex. 6 — Why is it typically necessary to use a strobe in high-speed DRAM protocols?

Ex. 7 — What is the advantage of using active low signals?

Ex. 8 — Prove that $tRC = tRAS + tRP$.

Ex. 9 — Why is the $ACT \rightarrow ACT$ delay less for a different bank group?

Ex. 10 — Why is a preamble and postamble required?

Ex. 11 — Prove the formula for the read to write delay. Modify the formula to include the parity latency as well.

Ex. 12 — Can the algorithm followed by the DRAM memory controller cause any memory consistency issues?

Ex. 13 — Create an addressing scheme for an FB-DIMM DRAM system. Make your own assumptions.

Ex. 14 — Why is NAND flash called NAND flash? How is it superior to NOR flash?

Ex. 15 — Design a memory controller for a 3D memory.

Ex. 16 — Can we use the Roofline model to compare a CPU and a GPU? Assume that they use the same off-chip DRAM-based memory system. What insights will we get?

Ex. 17 — How can we model software prefetching, hardware prefetching, and NUCA caches using the Roofline model?

Design Problems

Ex. 18 — Design the sense amplifier circuit using any popular circuit simulator.

Ex. 19 — Understand the working of the memory controller in DRAMSim2 or the Tejas architectural simulator.

Ex. 20 — Design a DDR4 DRAM memory controller using Logisim, Verilog, or VHDL.