
Chapter 17

Computer Memories

To best understand how the dynamic RAM circuitry of our computer works, let us make a short detour to look at memories in general. (Construction will continue in Chapter 18.)

17-1. Memory Basics

Let us begin with a look at what is inside a simple memory IC, shown in Fig. 17-1. The heart of such an IC is the *memory array* which contains the actual *memory cells* (the cells store the actual memory data.). The array shown in Fig. 17-1 consists of just four horizontal wires called *rows* and four vertical wires called *columns*, whereas a real memory chip will often contain hundreds of rows and columns. At the intersection of each row and column is a cell. Since we have four rows and four columns, the array shown has room for exactly 16 (4 times 4) cells. (Notice that the row and column wires, though they are shown as crossing, do not actually connect to each other. Instead, there is a cell at each intersection, and that cell has one connection to the row wire and another connection to the column wire.)

To accommodate 16 cells, we could have used one row and 16 columns, or 2 rows and 8 columns, or several other combinations, but in practice memory arrays generally have the same number of rows and columns because that makes the rest of the circuitry simpler. In other words, memory arrays generally tend to look like a square rather than a rectangle. Furthermore, the number of rows and columns is always a power of 2. For example, a 16K memory chip would have 128 rows and 128 columns, for a total of 128x128 or 16,384 cells. The next larger common memory chip would have 256 rows and 256 columns, for a total of 256x256 or 65,536 cells. This explains why 16K and 64K memory chips are common, but 32K chips are not - their array would be a rectangle rather than a square.

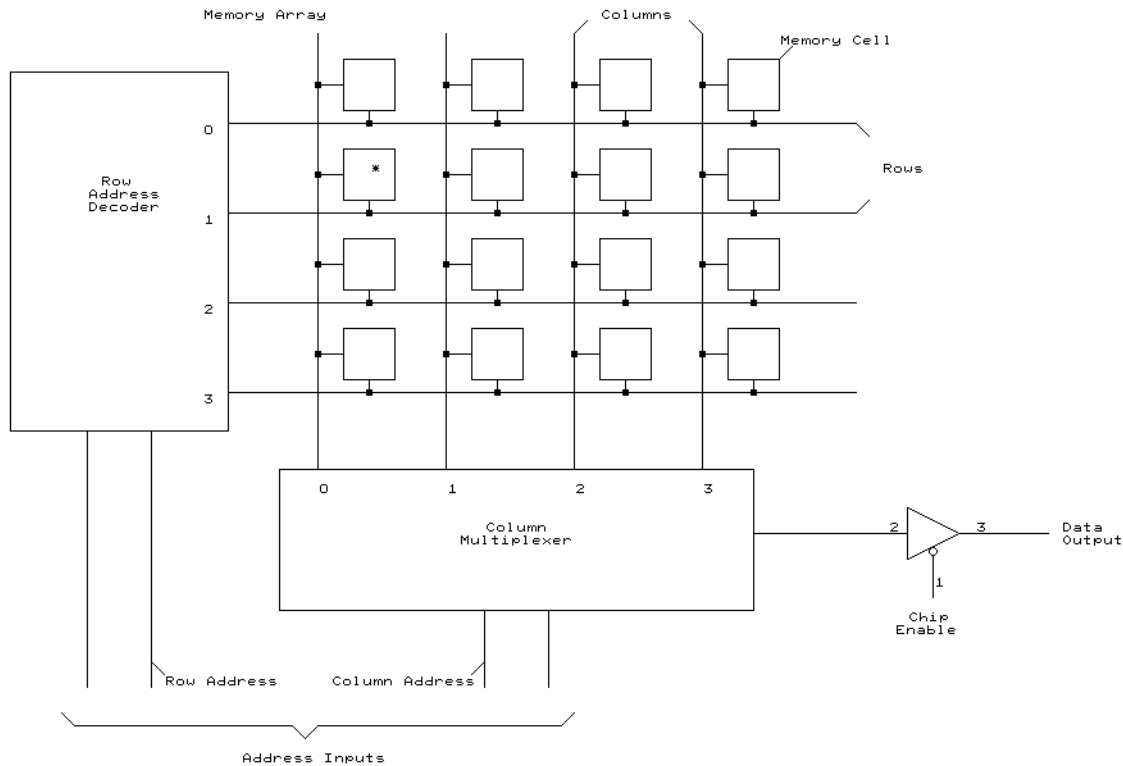


Fig. 17-1. A simple memory IC.

Each cell in the array stores one bit, so the circuit of Fig. 17-1 can store 16 bits. This particular circuit is configured to have 16 locations, each of which stores one bit; such an IC would be called a 16x1 memory, where the first number gives the number of locations while the second gives the number of bits in each location. Small memory ICs often have more than one bit per location, whereas large memory ICs almost always have just one bit in each location, but with many thousands of locations.

Each of the locations in the memory (that is, each cell) has an *address*; each time we want to read from or write into a cell (though Fig. 17-1 doesn't show the circuitry needed to write into a cell), we must specify the address of the specific cell to read or write by giving the IC a binary address on the address input lines. Since this circuit has 16 cells or locations, it requires a 4-bit address to specify the exact cell we want to access. The general rule is that x address bits are needed to specify 2^x addresses, so 4 address bits specify 2^4 or 16 locations in our simple circuit.

The four-bit address is split into two parts - a two-bit row address and a two-bit column address. Since there are four rows, we need two bits to choose one of them (again, because 2^2 is 4); since there are four columns, we need two bits to choose a column. Keeping in mind that most memory ICs have square arrays - the same number of rows and columns - that means that they will need the same number of row address bits as column address bits. This means that the *total* number of address bits is almost always an

even number. For example, a 16Kx1 IC with 128 rows and 128 columns has 7 row address bits and 7 column address bits (since 2^7 is 128), for a total of 14 address bits (and 2^{14} is 16,384). Similarly, a 64Kx1 IC has 256 rows and columns, 8 row and column address bits (2^8 is 256) and a total of 16 address bits (2^{16} is 65,536 or 64K).

Let's suppose that we want to read out the bit in location 4 of the circuit in Fig. 17-1, represented by the starred cell in the diagram. To do so, we send the binary address 0100 (a four) to the address inputs. The left two bits, 01, become the row address, while the right two bits, 00, become the column address. The row address is sent to a *row address decoder*, which has two inputs (for the row address) and four outputs (labelled 0, 1, 2, and 3) which correspond to rows 0, 1, 2, and 3. A decoder normally has a number of outputs, all of which are off except for one - the one specified by the binary input. In this case, the binary input (the row address) is 01, so the decoder turns on its 1 output and turns off the 0, 2, and 3 outputs.

Of the 16 cells in the array, the 12 cells which are in rows 0, 2, and 3 receive no signal from the decoder, so they do nothing. But the four cells in row 1 all receive a signal from the number 1 output of the decoder, so they all get enabled. In turn, each of these four cells sends its bit down a column wire to the *column multiplexer*. In other words, although we only want the contents of *one* cell - the starred one - all four cells along the same row send their contents down to the multiplexer.

The multiplexer's job is now to select the *one* desired bit from the four it receives and send it out the output. To do so, it acts like a SP4T switch - a single-pole switch with four positions, which selects one of the four inputs and sends it out the output. The precise input selected depends on the binary column address - in our case, the column address is 00 so it selects the signal entering on the input labelled 0 and sends it out to a tri-state buffer. If the chip enable input is on (it has to be low, since the tri-state buffer has an active-low enable input as shown by the bubble), then that bit goes out the data output.

As mentioned earlier, the circuit of Fig. 17-1 is very simplified. In an actual memory IC, the chip enable signal might also go to the decoder or multiplexer to prevent their working unless the chip is selected, there would be circuitry to write into cells (along with a R/W input), the array would be much larger, and there would be more components there that we haven't yet discussed.

The next question, though, is this - what is in a cell? That depends on the type of memory IC we are discussing. In a ROM or PROM (a Programmable ROM), the cell might consist of just a diode, or a diode in series with a fuse. Such a cell can store either a 0 or 1 bit, depending on whether the diode is connected or not (for example, if the series fuse is blown out). In an EPROM, the cell consists essentially of a FET transistor which is biased on or off by a charge stored in an insulating region.

In RAMs, there are two main kinds of cells: in a static RAM (or SRAM), the cell consists of a flip-flop which stores a 0 or 1, depending on whether it is set or reset, plus some additional components which connect the flip-flop to the row and column wires. Since this involves several transistors (often six or more in each cell), the static RAM cell is quite complex. A DRAM cell, on the other hand, consists of just one MOSFET transistor and

a tiny capacitor, which stores a 0 or 1 depending on whether it is charged or not. Since the DRAM cell is so much simpler than a SRAM cell, DRAM ICs generally contain many more cells than SRAM ICs. On the other hand, DRAM memories require more external support circuitry than SRAM, and are somewhat slower. Thus smaller memories are usually made of static RAM chips, whereas larger memories are generally dynamic RAMs (except in those cases where absolute top speed is a necessity and cost is no object.) In the SK68K computer, for example, there is a small amount of static memory (consisting of just two ICs) which allows us to get the system up and running quite quickly. But the main memory, 1 megabyte worth, is strictly dynamic to keep the total cost down. Even though the DRAM needs extra support circuitry to make it work, the circuitry is shared by the entire 1 megabyte of RAM so it is worth using, whereas it would not be worth using if only a few K of memory were needed.

17-2. Dynamic Memory (DRAM)

Fig. 17-2 shows a typical DRAM cell, consisting of a storage capacitor which holds the actual bit, in series with a MOSFET transistor. In normal operation, the decoder output is off and so the MOSFET transistor is biased off. This isolates the storage capacitor from the rest of the circuit so it can hold a bit. But when the row holding the cell is selected by the decoder, the MOSFET is biased on and the capacitor is connected to the column wire through the transistor. At this point, the cell can be read out (since the capacitor voltage appears on the column wire) or it can be written into (by sending a signal up the column wire, and through the MOSFET transistor into the capacitor.)

If this were all there was, there would be two major problems: First, since the capacitor is *very, very* small, it discharges very quickly. Just reading the cell (by turning on the MOSFET) puts enough of a load on the capacitor that it discharges almost instantaneously, but even when the MOSFET is biased off, the capacitor will typically hold its charge only a few seconds, and under some conditions, only a few milliseconds. Moreover, whenever a row wire is turned on by the decoder to select a cell on that row, **all of the MOSFETs on that row are turned on!** In other words, reading one cell selects all the cells on that row, with the result that all the capacitors in that row immediately discharge. Thus something has to be done to prevent all

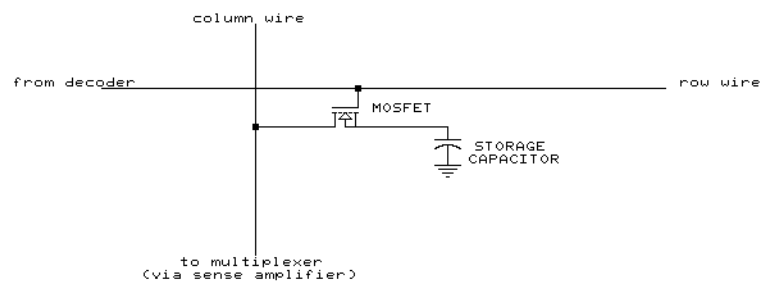


Fig. 17-2. A dynamic RAM cell.

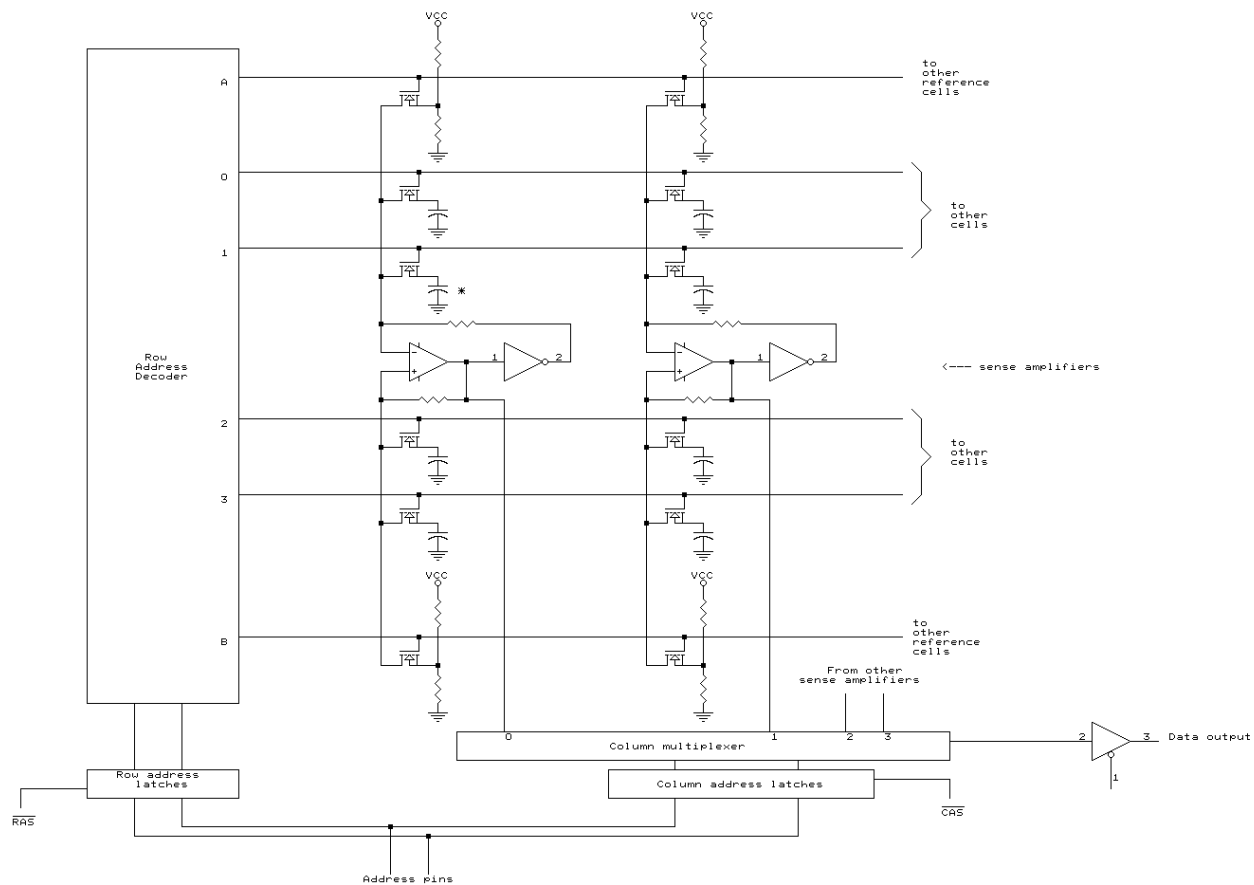


Fig. 17-3. DRAM IC organization.

the cells from forgetting their data. The overall dynamic RAM system therefore has additional circuitry which will (1) rewrite all the data back into all the cells in a row whenever any cell in that row is read or written, and (2) rewrite the data into all the cells of the entire memory at intervals of a few milliseconds. The first job - that of rewriting cells when a row is accessed - is handled internally by each DRAM IC; the second job - that of rewriting all of memory every few milliseconds - is called *refreshing* and is handled by external *refresh circuits*.

Fig. 17-3 shows how the circuitry inside a typical DRAM integrated circuit handles rewriting data into the cell capacitors whenever a row is selected. Though DRAMs typically contain thousands of cells, this diagram still shows only a small 16x1 chip with an array of four rows and four columns, though only the left two columns are actually shown in the diagram.

Fig. 17-3 has several new components we have not seen before. Although the 16x1 DRAM circuit needs four address bits, the diagram shows only two address pins, connected to both the row address decoder and the column multiplexer through two sets of flip-flops called the *row address latches* and the *column address latches*. The four-bit address is sent to the IC two bits at a time; the first two bits are stored in the row address latches by

a pulse on the $\overline{\text{RAS}}$ or *row address strobe*, and then the second two bits are stored in the column latches by a pulse on the $\overline{\text{CAS}}$ or *column address strobe*. This technique is used in DRAM ICs to save input pins. For example, a 256Kx1 DRAM would normally need 18 address lines, whereas splitting the address into two parts allows the 18 bits to be input through 9 address pins and two address strobe pins. This allows 11 pins to do the work of 18, so the DRAM can be packaged in a smaller case. There is, of course, a disadvantage - external circuits have to be added to split the address into two parts, and the process takes a bit longer than it would otherwise.

As before, the two-bit row address is again sent to the row address decoder, which outputs a pulse on one of the four row lines labelled 0, 1, 2, or 3, depending on the value of the row address. Note, however, that the array is now split into two parts by a row of *sense amplifiers* running across the middle of the diagram. These sense amplifiers are essentially op-amps or comparators, and have an inverting input (labelled with a -) as well as a non-inverting input (labelled with a +).

But now there are two more rows, labelled A and B, each of which contains a row of *reference cells*. These cells are similar to regular cells, except that they use a two-resistor voltage divider instead of a capacitor and therefore always output a constant voltage whose value, chosen by the ratio of the two resistors, is half-way between a 0 and a 1. The decoder is now modified so that, in addition to selecting one of the rows 0 through 3, it also selects a reference row at the same time, but it always makes sure that the reference row is on the opposite side of the sense amplifiers.

For example, let's again assume we want to read out the contents of the starred capacitor, which is again location 4 or 0100. The row address (01) entering the decoder selects row 1; at the same time, the decoder enables the row of reference cells connected to output B. Note that the reference row is on the opposite side of the sense amplifiers - row B gets selected along with rows 0 or 1, whereas row A gets selected along with rows 2 or 3.

When row 1 is selected, all of the MOSFET transistors connected to that row are turned on, and all four capacitors on that row (only two are actually shown) send their voltage to the top input to a sense amplifier. At the same time, the bottom input of each sense amplifier gets a reference voltage from a reference cell at the bottom. Since the reference is a voltage between 0 and 1, each sense amplifier can compare the capacitor voltage against the reference and decide whether the capacitor held a 0 or 1.

All of this must happen very quickly, because the capacitor almost immediately discharges. By then, however, the bit stored in the capacitor has already arrived at the output of the sense amplifier, which sends it down to the multiplexer and the output.

But each of the sense amplifiers has a sort of positive feedback circuit connected from output back to the two inputs. The actual circuit is somewhat different and more complicated than what is shown in Fig. 17-3, but the main idea is this: as soon as the bit arrives at the output of the sense amplifier, it is immediately sent back to the inputs through the two resistors. Since the top sense amplifier input is inverting, the output is fed back through an inverter so it comes back in the same polarity, but larger. This signal then recharges (refreshes) the capacitor. If the original capacitor voltage was lower than the reference voltage, this circuit pushes it back

down toward ground; if it was higher than the reference, this circuit pushes it up toward the positive supply voltage.

Even though we only wanted to read out the bit stored in the starred capacitor, actually every capacitor in row 1 was read out and refreshed by its own sense amplifier. This is an important concept - reading out any cell automatically refreshes all the cells located in the same row.

If such a DRAM IC is connected to a microprocessor, each time the processor reads (or writes - Fig 17-3 does not show any of the circuitry for writing into a cell, but this can be accomplished by feeding bits backward through the multiplexer) it refreshes an entire row of each DRAM IC. If the computer were to use data from every row, then the entire IC would be refreshed automatically and we wouldn't have to do any more. In general, though, we cannot trust that to happen, since the computer could easily get stuck in a loop where it only accesses one or two rows, with the result that all the rest of the memory would be forgotten. We therefore have to add external refresh circuitry to make sure that every row of each DRAM IC is properly refreshed.

17-3. DRAM Refreshing

Most current DRAM ICs require that they be completely refreshed once every 2 milliseconds, so the refresh circuits have to ensure that every row of the memory is accessed at least once every 2 milliseconds. To minimize the effort required, IC makers build larger DRAM chips a bit differently from the smaller chips. In smaller DRAMs, up to 16Kx1, the array is essentially square as we have discussed, and has 128 or fewer rows. In larger memories, the array wiring is split to make several smaller arrays out of the one large array, with each smaller array having only 128 rows and all arrays being refreshed at the same time. As a result, instead of a 256Kx1 DRAM having 512 rows and thus needing 512 reads for refreshing, it still only needs 128 reads. This makes refreshing faster.

There are two basically different ways of refreshing DRAM:

(1) The cheapest, requiring very little actual hardware, is to build an oscillator which interrupts the CPU once every 2 milliseconds., and forces it to stop the current program and execute an interrupt routine. The interrupt routine, in turn, does a read from every row and then returns to the main program. This is essentially a software approach, but it has the disadvantage that it wastes a significant portion of the computer's time and slows down every program. For instance, if a read of one row (counting the time to fetch and perform the read instruction) requires four microseconds (which is not unusual for an average 8-bit microprocessor), then 512 microseconds (plus interrupt processing time) would be taken up out of every 2 milliseconds for refresh. In other words, more than a quarter of the processor's time would be used up just on refresh.

(2) The second approach is to build a counter which counts out the rows from 0 through 127, and send its output as a refresh address to the memory. The counter must go through the complete count at least once every 2 milliseconds, and every one of those counts must be sent to the memory. The trick here is to do all this without slowing down the processor or

affecting any programs. As usual, designers use many different ways, some better than others:

(a) One approach is to periodically halt the processor, and send the 128 counts to the DRAMs instead, either individually or as a burst. If the computer has a direct memory access (DMA) circuit, then this is easily handled by the same circuitry. IBM PCs and clones use this approach, but it obviously slows down programs and so is not the best.

(b) The best - and most expensive - approach is to split the memory into two halves. Whenever the CPU is accessing one half, refresh the other half. For example, in an 8-bit computer, all the even locations could be in one half of the memory, and all the odd locations in the other half. Since most of a computer's time is spent accessing consecutive locations, it mostly alternates from one half to the other, so each half of memory is unused roughly 50% of the time. This leaves plenty of time to do memory refresh without in any way slowing down the processor.

(c) A middle-of-the-road approach is to try to detect clock cycles when the CPU is doing internal operations instead of using the memory, and squeeze refresh accesses into these unused slots. With a processor which uses many cycles for internal operations, this can sneak in refreshes without slowing down the processor at all, but this approach doesn't work quite as well with processors which use the address bus fairly heavily. In that case, the refresh circuits may be able to sandwich many - or even most - of their memory accesses between CPU memory accesses, but there may still be occasional conflicts when both need to access memory at the same time. In that case the refresh circuits must get priority to avoid losing data in memory, so there has to be a fairly complex circuit which times refreshes and arbitrates between CPU and memory accesses. This is the approach used in the SK68K computer, and it slows down CPU operation an average of one or two percent.