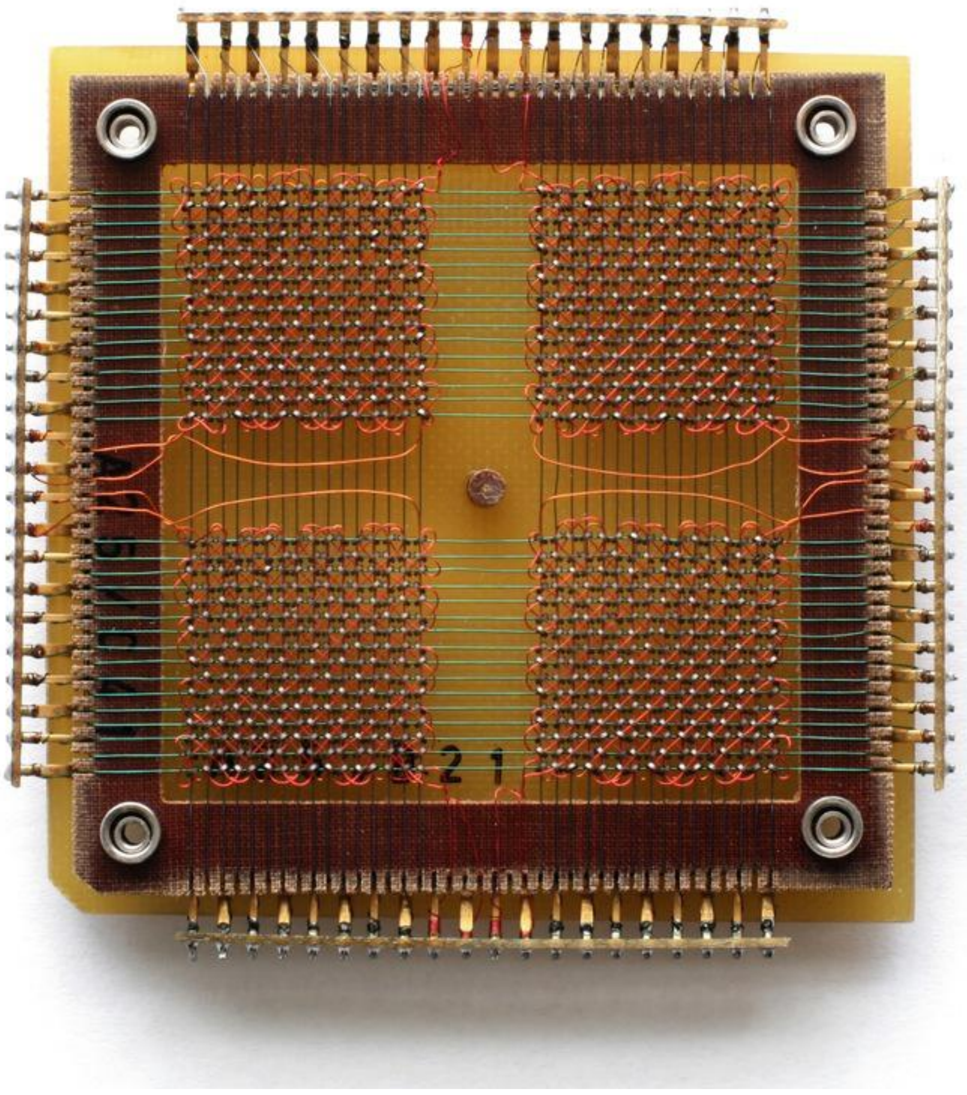
Digital Circuits 5: Memories

Created by Dave Astels



https://learn.adafruit.com/digital-circuits-5-memories

Last updated on 2022-12-01 03:12:14 PM EST

Table of Contents

Overview

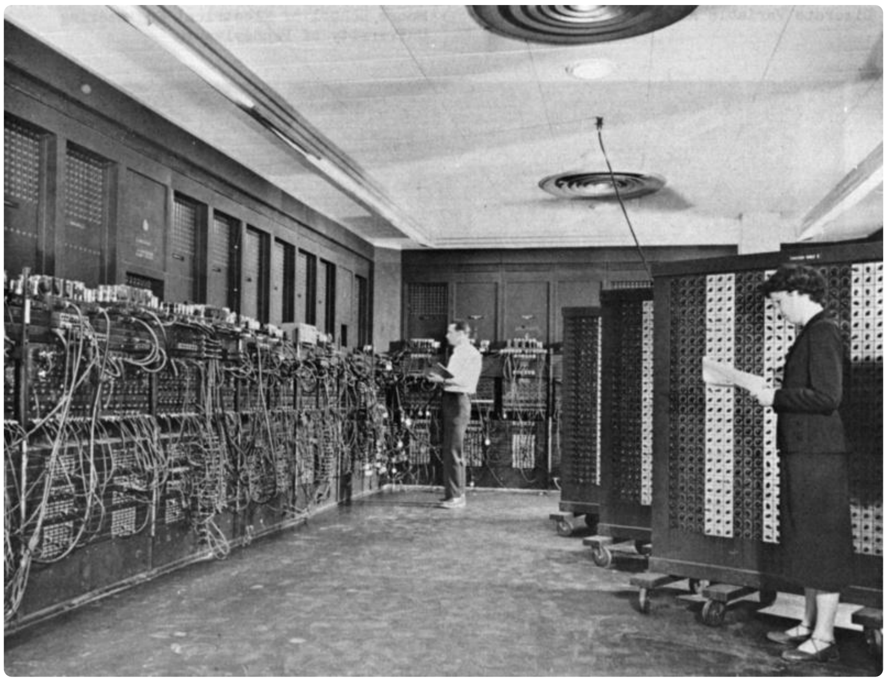
Memory for computers has been around since there have been computers. As with the rest of computer circuitry, memory has come a long way since the beginning.

In this part of the Digital Circuits mega-guide, we'll be look at computer memory, its history, how it works, and how to use it.

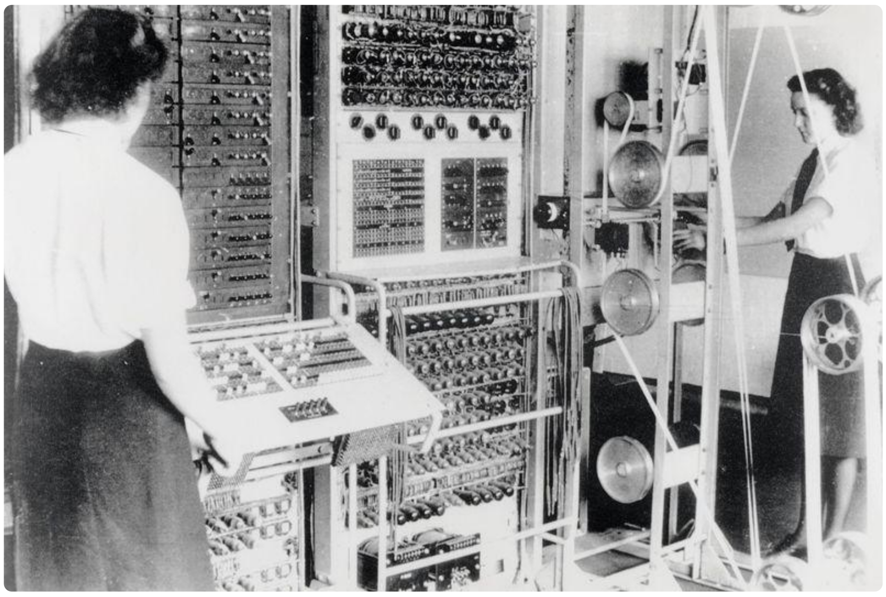
History

Vacuum Tube Memory

Back in the 1940s the first digital computer, the ENIAC, used a very small amount of memory made from vacuum tubes (). Actually, the entire computer was made of tubes. It could manage calculations with 20 10-digit numbers.

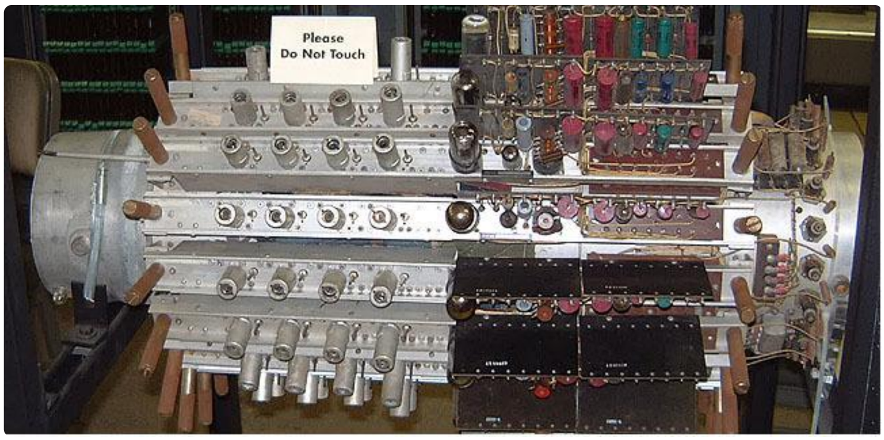


Vacuum tubes have several drawbacks (but keep in mind it was revolutionary at the time), primarily the amount of power required, and the reliability (or lack of it). Colossus () was a significant vacuum tube computer, released in 1943. It was used at Bletchley Park in the British WWII intelligence effort. When operating it consumed 8.5 kW (15 for the Mk2), contained 1600 tubes (2400 in the Mk2). Typically a tube would fail every couple days, which needed to be found and replaced (taking about 15 minutes).



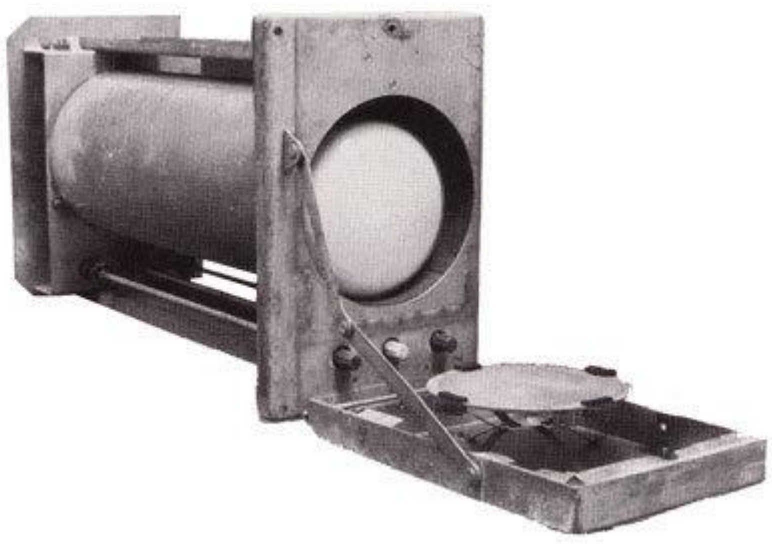
Mercury Delay Line Memory

Later, but still in the 40s, Presper Eckert developed acoustic delay line memory (). This consisted of a glass tube filled with mercury, with a crystal transducer at each end. At one end you would vibrate the transducer using a sound wave based on the serialized data to be stored. That would case the wave to ripple down the mercury in the tube, to be received by the transducer on the other end. That transducer would convert the wave in the mercury back into the electrical signal corresponding to the data. You had to have that loop back to be replayed into the tube to continuously refresh the stored data.



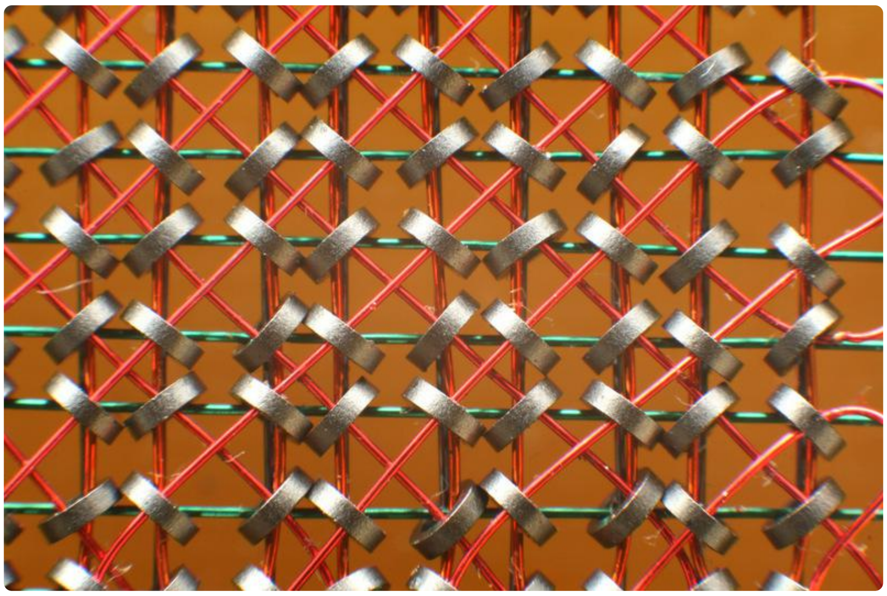
Cathode Ray Tube Storage

Remember when TVs and monitors weren't flat? I remember having a 21" screen that was bigger front to back that it was side to side. But I digress. Somewhat. The next evolution of storage used tubes very much like those CRT screens, the data being stored in the image on its face. The charge that made the phosphor glow was read by other circuits. The data was stored for a fraction of a second so, like the delay line memory, had to be continuously refreshed. The photo below is a Williams Tube (), one of the designs that was in use.



Magnetic Core Memory

In the late 40s several researchers, notably Jay Forrester of MIT, developed magnetic core memory (). This memory was made up of tiny magnetic rings that were called cores (hence the name of the technology). Wires are threaded through the cores allowing them to be magnetized in either a clockwise or counter clockwise direction. The direction of magnetism represents one bit of data: a 0 or a 1. Interestingly, reading a bit always results in it being a 0 afterward. So if a 1 was read, it would have to be rewritten to maintain the information. Wikipedia () gives a far more detailed description of the technology. Magnetic core did have the great advantage that the information that was written into it stayed there, unlike previous technologies that required continuous refreshing.



Magnetic core was the standard memory technology from the 50s through into the 70s. It only lost popularity when semiconductor memory was developed (initially by Intel). Semiconductor memory was much smaller and cheaper. Even to today the technology has had a lasting impact. Where do you think the term "core dump" comes from?

It's interesting to look back at the early computers and their technologies. Remember those photos of ENIAC and Colossus the next time you grumble about soldering an SMT MCU :)

Read Only Memory

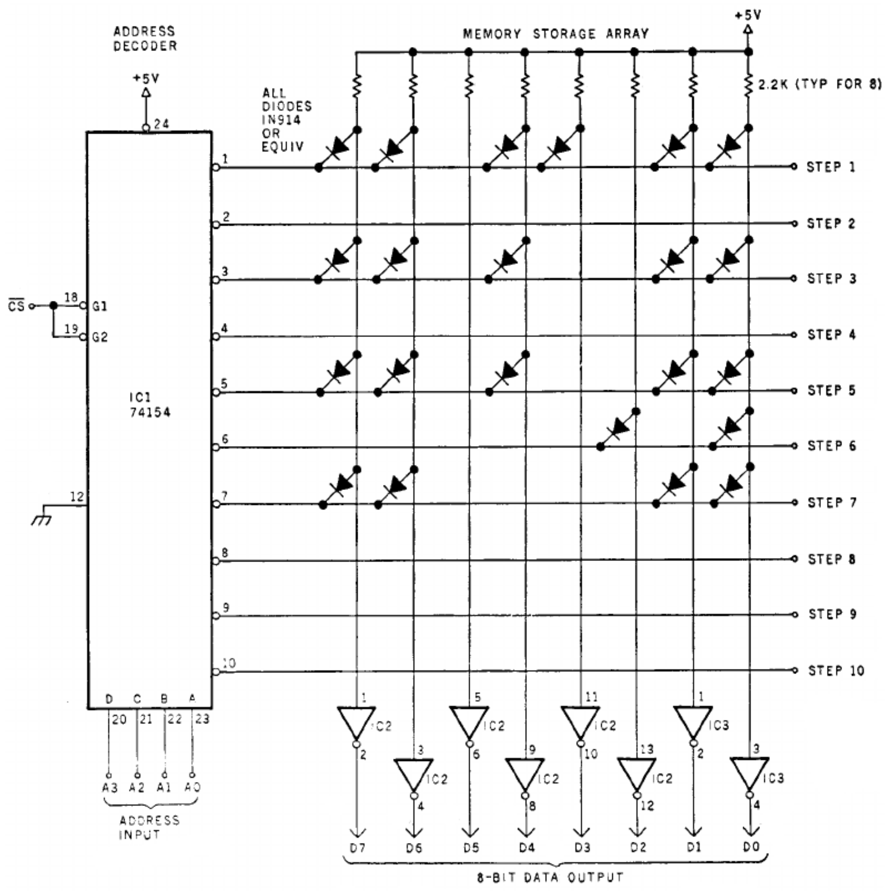
Read-only memory (ROM) is just that: memory that can only be read from, not written to. Some forms are can be written, but only by using special tools.

Mask ROM

This goes all the way back to the dawn of semiconductor technology. This is basically one big lookup table that is designed into the chip. As such there is no programmability once the chips are made. To change the contents, the circuit is changed and new chips are manufactured. This has several downsides:

1. It is only financial feasible to do in large quantities.
2. Any updates involve a product recall/update/return.
3. Time from design to part is long due to the manufacturing timeline.
4. Unusable for experimenting or small batches.

The circuit involved has several input lines, typically decoded from a binary encoded address, and whatever number of output lines are required, The input lines are connected to the output lines, or not, by a diode to encode the data at each address. The following is a small, simple implementation of ROM. It's not a mask ROM, but is an implementation of the same approach. The 74154 is an 8-to-16 decoder, much like the 3-to-8 decoder we saw in an early part of this guide (the 74138). Only the first few (of 16) addresses are shown, for brevity. The output corresponding to the address A (bottom left) goes low, while the rest remain high. The pullups at the top ensure that outputs are normally high. When an output of the '154 goes low (its address is on the A inputs and the /CS input is low) any output line connected to it through a diode goes low. In this way we can read the byte at any of the 16 locations. Note the inverters on the output lines, this negates the sense of the bits, resulting in a 1 wherever there is a diode. For example, at address 0 (the '154 output labelled "1") the data is 11011011.



Programmable ROM

Invented in 1956, this is also known as PROM. It is a ROM technology that can be written to once, and only once. It's basically like the above diode ROM, but with a fuse at each intersection. During programming, an address is selected and the desired data is placed on the data lines. Then a higher than normal voltage pulse is applied (generally through a special purpose pin) to blow the fuses where the data (at the selected address) should be a 0. Once a fuse is blown, that's it... there's no going back.

Of historical note, Steve Wozniak used PROMs in his inspired floppy disk controller for the Apple][() to, along with some clever programming, avoid most of the hardware a typical disk controller at the time contained.

Erasable Programmable ROM

One disadvantage of PROMs is that you only get to program them once. The practical effect of this is that if you need to change what's programmed onto a one, you throw it away and program a new one. So, like mask ROMs, they're still not the greatest for iterating on a design. You can replace them quickly as long as you have them on hand, but each time means paying for a new chip.

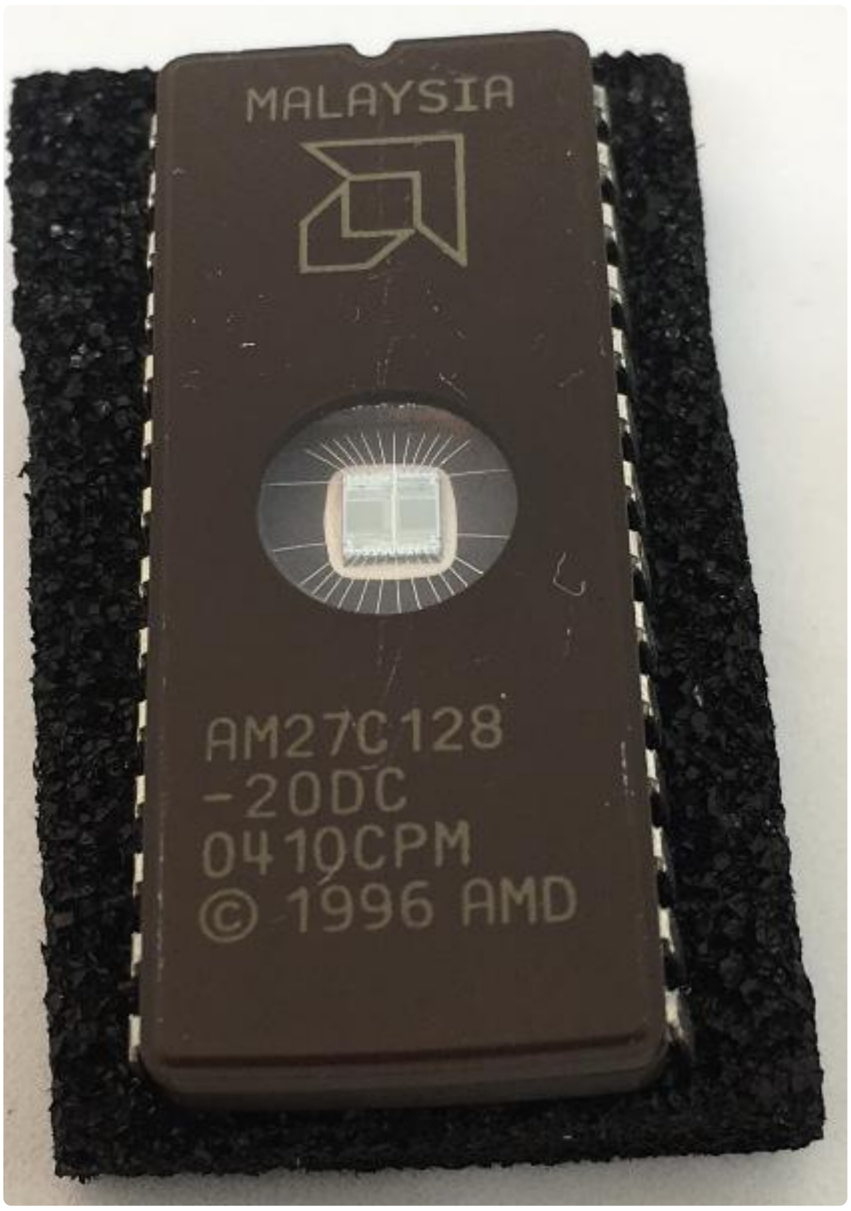
Enter the Eraseable Programmable ROM, aka the EPROM. This was a huge jump forward in that it could be erased and reprogrammed. The EPROM as developed at Intel in 1971, by Dov Frohman. Interestingly development of the EPROM got its start with the investigation of faulty ICs. The engineers found that some transistor gate connections accumulated a charge that changed their properties.

EPROMs are programmed in a way very similar to PROMs: select an address, supply the data to be stored there, and pulse a write line with a high voltage. The difference is that instead of blowing a fuse, programming a bit on an EPROM causes electrons (i.e. a charge) to build up on the gate of selected (via the address and data supplied) transistors. When the write pulse is done, those electrons are trapped, storing the programmed data for decades if protected from light. The data can be read any number of times without effecting the stored charge.

Now the bit about being erasable. It turns out that exposing the chip to UV light in the right conditions will cause the charges stored during programming to dissipate. I remember using EPROM erasers that were basically little tanning beds for chips. To make this workable, EPROMS have a clear window over the actual chip. Once programmed that window needs to be covered with a UV opaque sticker.

EPROMs can be erased and programmed several thousand times before wear and tear makes the chip unreliable.

This was an incredibly useful and popular technology for quite some time, and anyone that was doing computer design/hacking of any sort had a EPROM eraser and programmer. An note on that last bit: EPROMS had to be removed from their circuit before being erased (for physical reasons) and programmed (for electrical reasons). I'm toying with the idea of building an EPROM eraser and programmer for a future project guide.



Read Mostly Memory

This category of memory can be written to, but under specific situations and with certain limitations. As such, it's usually treated like ROM, but has the distinct advantage that its contents can be updated without having to remove it from the circuit.

Electrically Erasable Programmable ROM

EEPROM is a lot like EPROM, except that it can be erased electrically rather than using UV light. That means it can be erased and reprogrammed in-circuit. Another advantage is that usually a single word/byte can be erased and reprogrammed rather than the entire chip as with the EPROM.

Many of today's microcontrollers include a small amount of EEPROM to store various small bits of information, such as configuration options. For example, the ATmega328 (used in many maker boards including the Arduino UNO, Pro Mini, the Adafruit Feather 328P, Metro 328, and Metro Mini 328) contains 1K of EEPROM.

Flash

These days we don't often use EPROMs unless we are working with retro CPUs. Instead we're more likely to use flash memory.

Flash is technically a type of EEPROM, but with some specific characteristics that lend its use to situations where the amount of data being stored is relatively large, and not written often:

- Whereas EEPROM can erase individual bytes, flash erases regions at a time.
- Flash has a limited number of erase/write cycles before memory regions wear out and lose integrity. Over time that number has increased to the point where it is generally not an issue any longer. Additionally, the control hardware balances region use to spread writes across regions so that all wear relatively evenly.
- Access times are relatively slow compared to RAM or ROM. However for MCU applications, clock speeds are relatively low, so this tends not to be a limiting factor.

Since most of us are working with microcontrollers, we should be familiar with flash since that's the memory on the controller where our code is stored. Also SD cards and USB memory sticks are flash based. Adafruit's CircuitPython blends the two uses, storing the bootloader and python runtime in flash as usual, but also providing access to a filesystem via USB for loading Python code and data files onto the device.

Flash is a primary memory technology and has improved (as things do) drastically over time. The current state of flash make it feasible to use it as the sole secondary storage technology, i.e. Solid State Drive (SSD). Flash is perfect for this since it's much faster than disk (especially the NVMe/M.2 format SSDs) and more reliable since there are no moving parts.

If you have a Raspberry Pi (or the like), you use a microSD card as the "disk"; it's basically an inefficient SSD. In that context, the computer is slow enough that the slow SD card isn't a huge problem.

Read/Write Memory

This is usually called RAM. Which is odd, because RAM is an acronym for Random Access Memory. Ok, not so odd when we think about the history. A lot of the early memory technologies (all of them? I'm not sure... I'm not THAT old) were NOT random access. They were sequential access. You couldn't just read/write any word in memory. Instead you had to wait for it to "come around again" and read it or write it when it did. Whether that was in a delay tube, or on the surface of a rotating magnetic drum/tape/disk. Even today with a hard disk you have to wait for the platter to spin around to where the data you want is located. That's why 7200 rpm spinning hard drives are more responsive that 5400 rpm ones.

Interestingly Mask ROMs, EPROMs, etc. are all random access. You provide an address, and read the value stored there. In fact all semiconductor memory is random access.

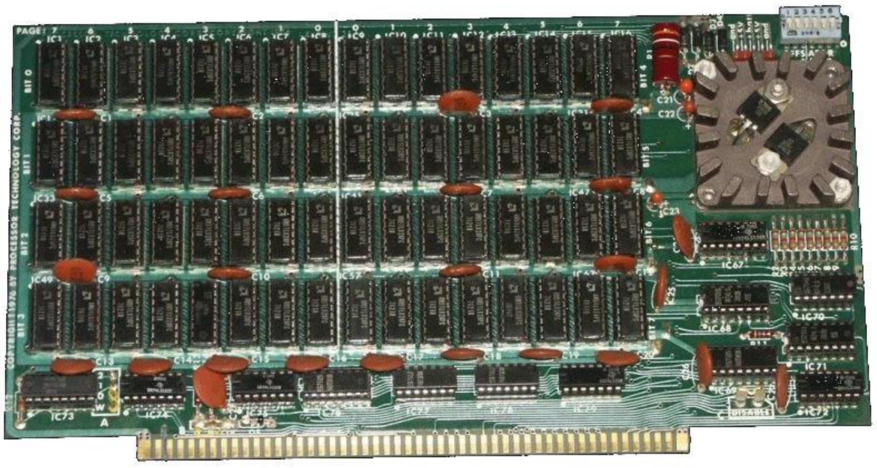
So what does RAM really mean for us? It's memory that we can write as well as read. Flash starts to blur that differentiation, but flash is still relatively slow. RAM is fast. The drawback is that it isn't persistent. When the power goes away, so does the data. That's why a system will have some RAM for working storage (data being manipulated), and some ROM/flash for more permanent storage (the OS, firmware, or applications). In a larger system like a server, workstation, or laptop, there is often a disk of some sort.

We looked at magnetic core in the history section of this guide. That was an early form of RAM. Data could be written and read "in place", i.e. without removing it from the system that was using it. Semiconductor RAM replaced core as the standard memory technology very quickly: is was smaller, cheaper, and production was automated rather than largely manual.

There are two types of RAM that we will consider: static and dynamic.

Static RAM

As implied by the name, static RAM () just sits there remembering what you put in it until you remove the power. It uses a flip-flop to store each bit. That means that each memory cell (i.e. each bit) is fairly large, typically 4 or 6 transistors, so there doesn't tend to be too many on a chip. The big advantage is that nothing extra has to be done for them to hold their contents, which means simpler support circuitry, which means that static ram is easy to work with. If you only need a small amount of read-write storage, then it's much easier and overall efficient to use a bit of static RAM. Back in the day, I made heavy use of the 6116 2Kx8 static RAM chip. That chip stored 2K bytes. Those chips were in 26 pin, wide (.6in) dual-inline (DIP) packages. Today, you can get the same chip in a narrow 24-pin DIP. Think about that, 2K of RAM in a chip that's almost as big as a through-hole ATmega328 (a 28 pin narrow DIP). The '328 contains 2K of ram in addition to the CPU, Flash, and all the IO, timers, etc.



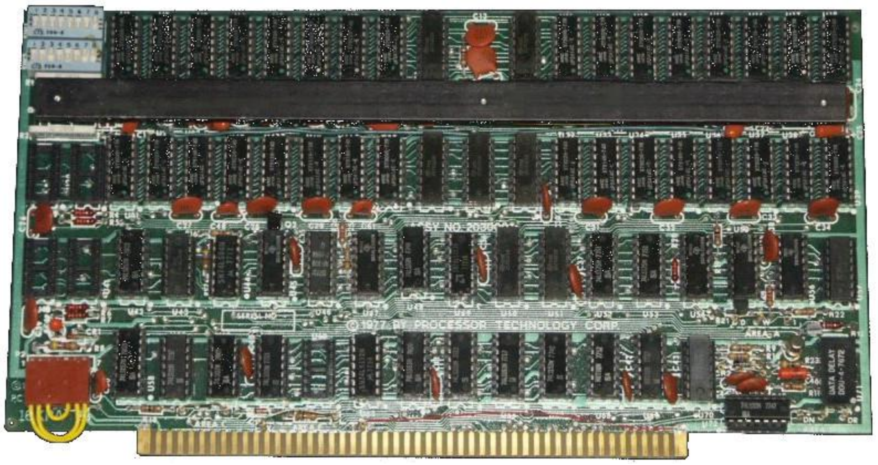
Dynamic RAM

Something dynamic is always changing, and dynamic RAM () is no different. Where static RAM uses a flip-flop to store bits, dynamic RAM uses 1 capacitor and 1 transistor for each bit. This is drastically smaller that a bit in static RAM, so dynamic RAM chips tend to have many more bits on them. This makes them cheaper and denser than static RAM (per bit). As computers became more powerful, with bigger address spaces, having more RAM was desirable. Since dynamic RAM was denser than static RAM, it became the standard memory technology for all computers larger than embedded 8-bit systems. For example, my current workstation has 16Gb sticks of dynamic RAM, each one the size of about 8 state of the art static RAM ICs which have a capacity of 2M each (so that's 16G of dynamic RAM in the same space as 16M of static RAM... that's 3 orders of magnitude difference in capacity).

The downside of dynamic RAM is the same thing that gives it it's major advantage: it uses just a capacitor and transistor to store each bit. Capacitors have the habit of discharging over time, and each dymanic RAM cell uses a very small one (meaning it holds very little charge, which dissipates quickly). In order to retain the data, it has to be refreshed frequently. Thankfully, entire sections of the data array* can be refreshed at the same time. Special, additional circuitry is required to do this, but there have been some clever designs to minimize that. The Apple][(all praise Woz the great) made use of it's video hardware to do RAM refresh for free, as a by-product of fetching data to be displayed**. The Z80 CPU (a contemporary of the Apple]['s

6502) had built-in dynamic RAM refresh support that took care of much of the hardware needed.

RAM technology has advance a lot since them, but the concepts are more or less the same.



Have a look at those two memory boards. The static RAM board is almost entirely RAM chips (the regular block of 4x16 identical chips) with a small amount of support circuitry. Compare that with the dynamic RAM board: the top two rows are mostly RAM chips (except for the 5 in the middle). The other two rows are support circuitry.

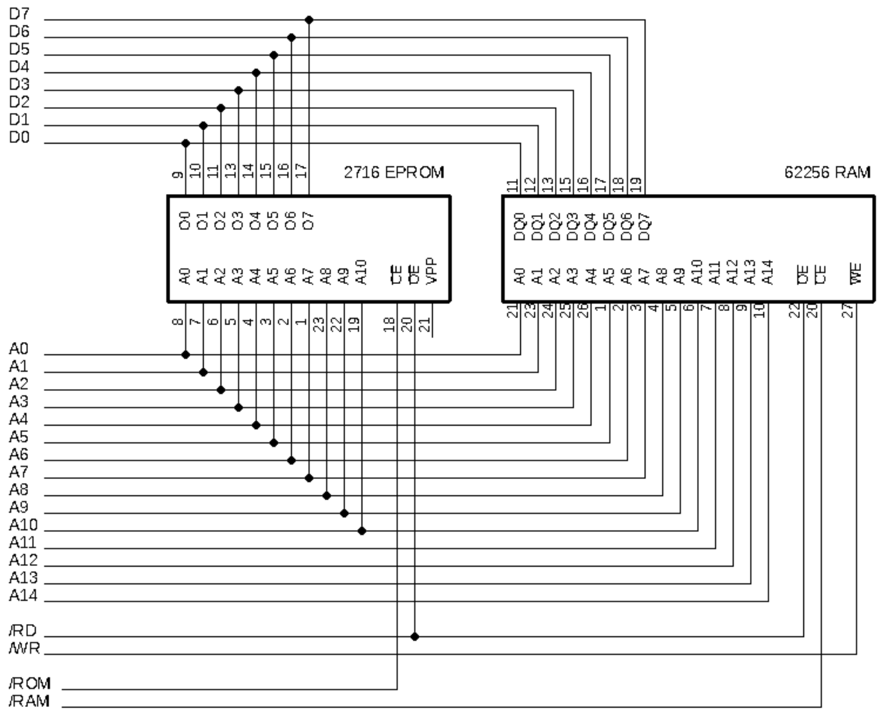
For comparison, below is one stick containing 16G of dynamic RAM. It's smaller than 14cm x 4cm and costs about $825 USD for 4 of them on Amazon today.

* The bits in a dynamic RAM chip are typically arranged in a more or less square array. An entire row of the array can be refreshed at once. For example, the 4116 used in many 8-bit era computers contains 16K bits arranged in a 128x128 array.

** If you want a master class in digital design, study the Apple][. "The Apple][Circuit Description" by Winston Gayler (published by Howard W. Sams & Co) is the go to walk through and analysis of the Apple][hardware.

Using memories

Modern* memory chips are incredibly easy to use (apart from dynamic RAM refresh requirements). Below is a typical schematic for using a 2Kx8 EPROM and a 32Kx8 static RAM. Notice the address bus that is common to both, as well as the data bus.The CPU places the address of the data it want to read/write on the address bus. For a read, it will take the resulting data from the data bus after the selected chip provides it. For a write, the CPU will place the data to be written on the data bus and ask the appropriate chip to write int into the selected location. Read (/RD) signal goes to the output enable (/OE) signal of each chip. That asks the chip to put the addressed data on its data pins if the chip select (/CS) is also active. The write signal (/WR) only goes to the RAM chip since the EPROM can not write in-circuit so has no write enable (/WE) input. It does connect to the RAM chip, though, asking it to take whatever is on its data pins and place it in the addressed location.



That's all there really is to it. The only real work is generating those chip select signals (/ROM and /RAM in this case) and for that we generally want to use some sort of 1-of-n decoder (e.g. a 74138).

* since the 80s if not before that

Next in the Series

What have we covered so far? We looked at the basic ideas of digital circuits, built some tools, explored various types of combinational and sequential circuits, and looked at memory technologies.

In the next tutorial in this series, it will all be put together in a simple project that will be used later in the series.

Series Index