On Memory Architecture

Shaan Fulton

July 4, 2025

We know how to represent information as bits. We also know that we can represent both data and instructions as bits (this is von Neumann architecture, see `the-stored-program`).

But how does this actually work in practice? Do we just have a massive list of one's and zero's to represent our instructions and variables? Yes, we do. Memory (and storage, for that matter) looks like this:

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09
AF	34	C1	00	B7	5D	4E	13	88	FA

Memory is byte addressable. That means every byte (256 combinations, 8 bits, 2 hexadecimal digits) has an address.

"Why not make things bit addressable instead?"

We could. But memory addresses can only be so long. Our CPU gets data through a memory bus (more in this below). These memory buses are only so long. For example, if we have a 128-bit memory address but our CPU memory bus is only 8-bits, it will take 16 operations just to get our address processed by the CPU so that we might be able to visit it. This is terribly inefficient.

If we made things bit addressable, a reasonable 32-bit memory address would allow for $4,294,967,296$ combinations. This means we would be able to address $4,294,967,296$ bits, or $536,870,912$ bytes, which is about 500 MB.

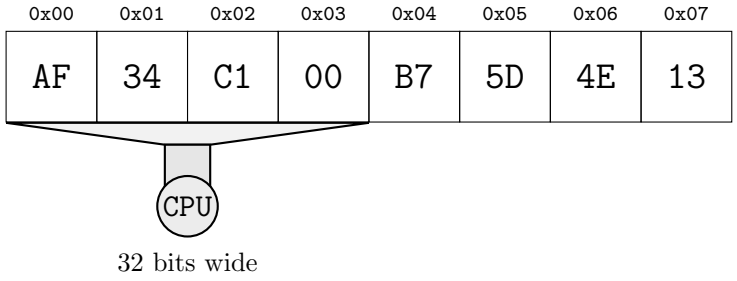
That sucks. That means the maximum RAM we can have in our system is 500 MB. Conversely, if we make our system byte addressable, we up that to 4 GB. Much better.

The Memory Bus

So we have this massive array of cells, each containing one byte, with a little address fixed to it. We'll assume we're working on a 32-bit address bus (32-bit addresses, a 32-bit system, etc.). But how do we actually use this array?

Our CPU has to perform its minimal operation set so that we can compose and execute all sorts of programs. But it needs to get its instructions and data from memory somehow. The CPU uses a CPU bus to accomplish this.

Think of the CPU bus like a fixed width vacuum. In our system, it can suck up 32 bits of memory at a time.

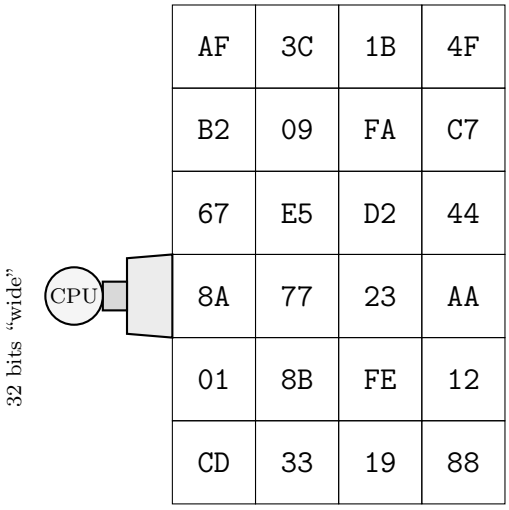


32 bits wide

Memory Alignment

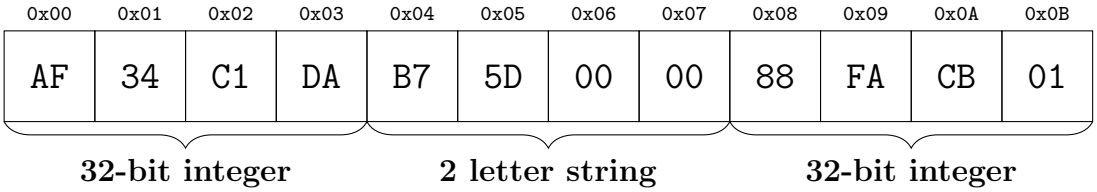
Because the CPU is sucking up 32 bits at a time, or 4 bytes, that means it can read a whole integer at once (most integers are represented with 32 bits)! That's very convenient. We can easily pull in an integer to do operations on it and then spit it back out all in single swoops. It can also read 4 characters at once (ASCII characters are just 8 bits or 1 byte)!

But there's a little caveat to all this. The CPU memory bus isn't really like a free-floating vacuum that can align itself to any address. The actual hardware more closely resembles a set of drawers where each drawer contains 4 bytes (imagine four folders, or dividers). And so in reality, the CPU memory bus can't read addresses 2 to 5 or 3 to 6 in one go, because these would be folders between two seperate drawers (drawers containing 0 to 3 and 4 to 7 respectively).



To accomodate this architecture, we actually make sure that our data is stored in "alignment" with these drawers.

We could totally ignore this and have an integer, for example, stretching between two drawers. But then we would need to make two attempts at getting that integer out of memory. Too slow! And so our memory ends up looking a little like this, with padding wherever we do not fill up an entire 4 byte drawer (yes, it ends up wasting space, but it is faster):

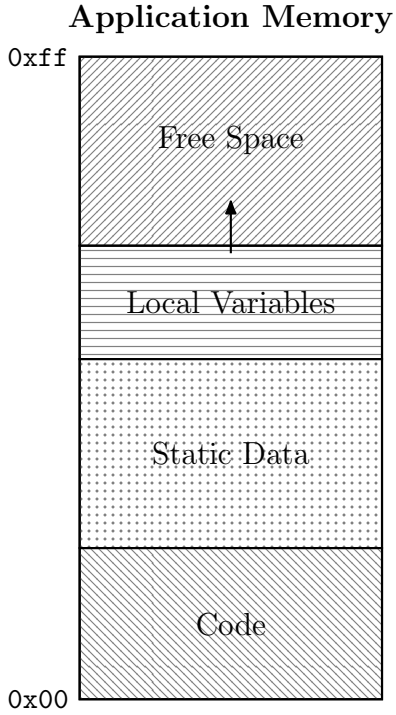


Memory Locations

We are very close to understanding how memory works on a computer. But there are some additional details. For instance: do we really just store **everything** in one massive line, sort of just stacking as we go?

We could. Say we start up a program and grant it a section of memory. Maybe we'll dump all the instructions into memory first, then we'll dump global variables that won't need more space as our application progresses, and lastly we'll dump everything else that comes up as we go through the program (so think local variables and arrays that grow and shrink).

Our memory might look something like this:



So our little application is running! We pile on the local variables from a function that works on a massive string. And then some more from an advanced physics simulation function. And then more from another massive string manipulator. The ones from this massive string manipulator we want to hold on to, we'll need it later. But as we return to our physics simulation, and return its data, we determine we don't need that space anymore. So we could just store the fact that there's free space in that area now.

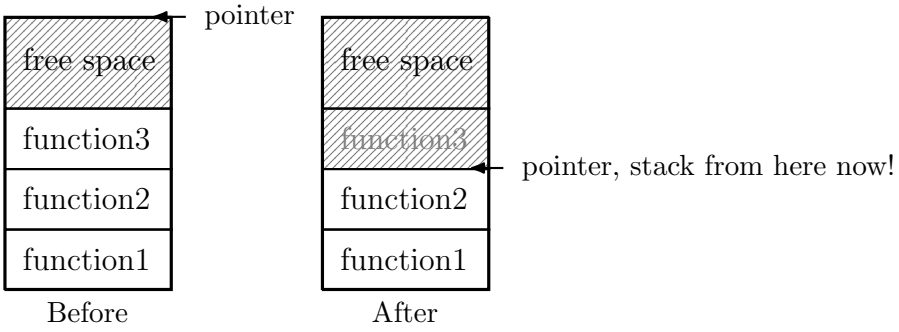
But wait. Now we have to keep track of all these little gaps in our data where there's free space. And then whenever we add more things to memory we have to literally iterate over all those gaps to find a spot that works for our piece of data. And my goodness, imagine if we find a spot for a local

frame in between other memory areas, and then need to extend that local frame! Now we have to move all that local frame's data to the top of our stack so it can extend freely! This is extremely innefficient, and seeing why is critical to understanding why we store memory the way we actually do.

The key idea here is this:

> **"Local function frame memory is different from long term allocated memory."**

The former is quite easy to deal with. We can stack as we get more data, and then simply wipe the stack and move ourselves back down to the last frame and continue stacking from there. That's because once we're done with a local frame we really don't need that data anymore, and we also only go to one local frame at a time, so we just move our little pointer around up and down as we add and remove local frames.



The latter is a bit more involved. We need to store a list of where all the free spaces are as we free up data, and shift things around as we extend them. For instance, if we create an array of 20 integers, and later one want to store another 10 integers on the end of that, it would require us to do a bunch of shifting if there's no space on either side. We would have to copy all 20 integers and move them to a larger spot... which means searching through our free list for that larger spot... it's quite involved.

We need this sort of storage. But it's clearly something we'll handle differently from how we handle local function frame storage.

And so: because this long term allocated and reallocated data is more complex to store than short term one time function frame data, we actually create two expanding memory locations...

The Heap and The Stack

The first of these is the stack. The stack is for our local function frames. The second is the heap. The heap is for this longer term allocated and re-allocated memory. It's data that has to exist between function frames, but is still constantly growing and shifting, unlike static data.

> **Brief tangent in pedagogy**: Most courses teach the heap and stack by simply explaining what they are and what data they store. This is brute force memorization. The insight of the approach we've taken here is that we're arriving upon it as if inventing it, thus we understand precisely the reason for and beauty of its existence.

And so here it is. Modern memory location architecture:

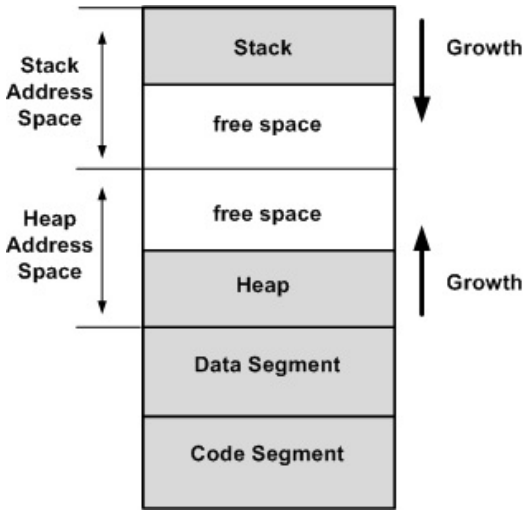


Figure 1: The Memory Locations

Location	What is stored
Stack	Local variables, function calls
Heap	Dynamically allocated memory
Data Segment	Global/static variables
Code Segment	Program instructions