

## Re: memory reading and writing

**Source:** <http://coding.derkeiler.com/Archive/Assembler/alt.lang.asm/2004-07/0703.html>

---

**From:** Beth (*BethStone21\_at\_hotmail.NOSPICEDHAM.com*)

**Date:** 07/16/04

Date: Fri, 16 Jul 2004 05:04:19 GMT

Ludas Matyas wrote:

> *How do I know how much is the Memory bandwidth/ frequency (in MHz*  
> *33/66/100/133 etc.) on my PC?*

Ummm, typically there's a small sticker or writing on the memory board itself which says...also, the initial "POST" screen that the BIOS shows when you first switch on your machine often states the memory bandwidth...and, if the memory was purchased and came in a box, then it might specify this on the side of the box ;)

But, yeah, I know: You mean "how do I determine this \_programmatically\_"...

Unfortunately, I'm more "software" than "hardware" on these nitty-gritty matters, so, umm: Goodness knows! I've got a funny feeling it's probably "chipset-specific" or something, anyway...

I defer this question about how to work it out using a program to someone else more qualified who actually knows the answer...

> *If I have a PC133 SDRAM does it mean its bandwidth is 133 MHz\*64-bit=1064 MB/sec = 1.0 GB/sec*  
>  
> *Why is reading memory faster than writing it?*

Good questions; This is "hardware stuff" again...I only bother with the "software" side of things usually...I'm not really a great fan of "gadgets"...so long as the hardware works, I don't usually care how it works...at least, in terms of "transistors" and "asserted pins" and that kind of "hardware" thing...I start where "software" starts...try asking wolfgang or one of the more "techie hardware" people around here...

> *I have a CPU of 800 MHz (Pentium III).The frequency ratio is 6x Because I*

> *have a 133MHz RAM.*  
>  
> *if I have a code like this:*  
>  
> *mov dword ptr[edi], eax*  
>  
> *; Then I have 6-1 cycle time here to do something before a*  
*memory data can*  
> *be read.*  
>  
> *mov dword ptr[esi], ebx*  
>  
> *Or am I misunderstanding something?*

You're not accounting for the "cache memory"...

Indeed, due to the large discrepancy between the speed of the CPU and the speed of the memory bus, plus the fact that you can't go for too long without needing to access memory, "cache memory" is installed to "bridge the gap" between the two different speeds...

There's usually two levels of "cache memory" – L1 and L2 – between the CPU core and the actual system RAM...the L1 cache memory is on-chip and near register-like in access speeds BUT, being on-chip and expensive memory, there's a limited amount of it...next, there's the L2 cache, which is off-chip and not quite so fast BUT there's more of it...finally, there's the actual system RAM itself: far off-chip, runs at the bus speed mentioned BUT, well, it's possible to have MBs and GBs of the stuff ;)...

Basically, fast memory is more expensive...but, using the "cache memory" scheme, then you can often access memory without any delays at all as if you had lots of the expensive fast RAM when you don't (well, unless you get a "cache miss")...

The idea is quite simple but really rather clever...when memory is read from system RAM, a whole "chunk" of memory is actually moved across in one go (not just the specific memory location you want to read but a larger "chunk" including a bunch of the bytes following this in memory as well :)...this is then copied into the "caches" on the way to the CPU...

What's so special about this? Ah, well, if you're then going to access the next byte after that with the next instruction, then as we read a whole "chunk" of memory from system RAM and copied it into the faster access "cache memory", then there's a copy of this RAM in the L1 cache which can be accessed at near register-like speed...so, we can read a whole series of bytes at speeds as if they were all stored registers for the rest of that "chunk"...

## alt.lang.asm: Re: memory reading and writing

When writing, the CPU writes – again, at fast speed – to the L1 cache copy...so the CPU actually directly deals with just the L1 cache which is superfast...although, as noted, this cache memory is superfast but it's limited in size...at some point, the memory you want to read / write won't be in this cache...in which case, the L1 cache looks in the L2 cache to see if there's a copy of the memory there it can use...the L2 cache is not quite so fast but it's larger in size than the L1 cache...and when the memory to be accessed isn't in the L1 or L2 caches, then it's fetched from the actual system RAM itself (bus speed but you can have lots and lots of it :)...the writes are "flushed" back to system RAM in a "lazy" fashion (not written immediately but as a "chunk" :)...

Thus, when successive memory accesses are close together – and typically they are...a principle called "locality of reference": most related data is clustered together and is read / written in a sequential order byte by byte, which is what the "cache memory" scheme works on optimising – then the program can mostly operate at the CPU's speed via the L1 cache memory...

But as this superfast memory is limited, it can only hold so many "chunks" of copied RAM inside it...so when it "misses" – the memory to be accessed isn't already stored in the cache – it "falls through" from the L1 cache to the L2 cache or from the L2 cache to system RAM itself...the L2 cache isn't quite so speedy but it's bigger so has more copied "chunks" inside it (and, thus, more chance of the memory you need being there, if it was a "miss" in the L1 cache :)...ultimately, if the memory isn't stored in either cache, then it passes through to system RAM (runs at the quoted bus speed you were mentioning but, this is your actual RAM and you can have MBs of it, if you like :)...

[ And, in a sense – though it works by a different mechanism and is controlled by the OS rather than the hardware – with "virtual memory", this can even be considered to be extended one level further in that "virtual memory" uses a "swap file" on the hard disk...and the PC can appear to have more RAM than it really does physically have by using the disk to store "pages" of memory which get swapped in and out of physical RAM as needed...the hard disk usually have even larger capacity but much slower than any memory (because, well, hard disks have moving parts...in terms of the speeds computer operate at, anything with "moving parts" is intolerably slow by comparison :)... ]

Hence, IF the memory to be accessed is in the "L1 cache" then there will be no visible delay at all...the problem with this scheme comes about from "cache misses" – where the memory to be accessed isn't already in the cache – because then it must "fall through" to the other cache or system RAM...and these

## alt.lang.asm: Re: memory reading and writing

accesses are slower...although, once you make that first access to this region of memory, another "chunk" is loaded into the caches and any subsequent memory accesses to that location (or to neighbouring memory locations :) will again happen at L1 cache speeds...so, you get "penalised" on the first access but, while that memory remains in the cache, subsequent accesses to memory in the same "chunk" will already be loaded into the cache...

When access speed really is an issue, then an optimisation to consider is, of course, to make the data access "cache consistent"...that is, you try to minimise "cache misses"...keep all your memory accesses inside one "chunk"...then the CPU operates at the rated "clock speed"...

Without the cache memory – and some systems allow the caches to be disabled so that you can see this directly – then all the memory accesses would, indeed, have to always "fall through" to actual system RAM each and every time...as you can't write too many instructions without needing to access memory, then this would actually, indeed, drop the speed of a program to `_bus speed_`, NOT the CPU speed, however fast the CPU is (because it would constantly need to wait for the memory to be passed back and forth along the memory bus to the system RAM :))...

When I mentioned this before, someone tested out the theory and disabled their caches and then tried to boot up Windows...it took absolutely forever to load...but this makes sense because, without the caches stopping the CPU constantly `_waiting_` for data to be delivered across the bus to system RAM, then the software will be running at, say, the 133MHz of the bus (on average; Not a precise science...because, yes, non-memory instructions would continue at CPU speed ;)...yup, doesn't matter too much whether it says 3GHz on the CPU itself, it's constantly having to `_wait_` for memory, which only runs at 133Mhz in this example...

This is the purpose of introducing the "cache memory", of course...a high-speed bus and superfast memory is `_expensive_` (and needs to be `_close_`, right up to actually being `_on-chip_`, that physical space is also an issue)...but using the caches, the system can (again, on average; not a precise science) operate at CPU speed most of the time while the main system RAM is, indeed, the non-expensive, not-so-fast, sitting over a memory bus from the CPU type of memory that you can have MBs or GBs installed, no problem...the use of the "caches" smooths out the CPU vs. bus speed discrepancy so that it `_mostly_` doesn't make a difference, without costing you the Earth in buying the expensive superfast kind of memory to do so...

## alt.lang.asm: Re: memory reading and writing

Of course, it's not perfect...when you get a "cache miss", the system does drop in speed because it has to resort to going to the slower access memory...but, hey, with a well-written "cache consistent" program that has "locality of reference" to exploit the caches to the full then this happens less frequently and it's saved you a lot of money in buying your PC and its memory ;)...

> *Pushing onto and popping from stack is it also memory reading/writing?*

Yes; Essentially, the "stack" is actually just the (E)SP register and how it gets used, when you get down to it...this register contains the memory address of the "top of stack"...when you PUSH something onto the stack, it gets written to that memory location and ESP is decremented...when you POP something, ESP is incremented and the value pulled from that memory location...

[ Yes, you read correctly...the "stack" is actually "upside-down" in memory...or, at least, it's "upside-down" from the natural perspective you'd take of, say, a "stack of plates"...the stack expands *downwards*, moving to (numerically) *lower* memory addresses as things are added...and when things are removed, it moves back to (numerically) higher memory addresses...although, you only need to know this when you're accessing the "stack" in a more direct fashion than only PUSH and POP (for example, you can "reserve" a whole chunk of your stack space – say, as "scratch memory" for local variables inside a procedure – with "SUB ESP, 64"...this is fine because, as noted, the "stack" is really just an abstraction based on the ESP "top of stack" register ;) ]

But, yes, the "stack" *IS* in memory and, thus, any reading / writing to the stack – such as with PUSH and POP – is accessing memory...

Although, from the discussion of the "cache memory" above, it should be noted that as the stack is used for all manner of things all the time – passing parameters, local variables, storing return addresses (CPU uses the stack for this automatically, in fact, whenever you "CALL" / "RET" implicitly ;), etc. – then there's a very high chance that the memory for the stack is already in the "cache"...and, thus, though the stack is in memory, it's also usually in the L1 cache memory too, which is superfast and operates at register-like speeds, anyway...

[ (Side note, not related to rest of thread...ignore if you don't know what I'm talking about, it's referring to things said in other earlier posts) Having said this, some people might

query the fuss I therefore make about not using the stack for this and not using the stack for that...surely, if it's all in the L1 cache and we have register-like speeds, what am I worrying about? Ah, it's a subtle and simple point: the stack `_MAY_` be in the cache and `_MAY_` be accessed with register-like speeds...on the other hand, registers are `_ALWAYS_` accessible at register-like speeds, by definition...no "may" about it...`_AND_` the stack is, indeed, `_STILL_` in the cache too for other purposes as well...the stack is still there so we're not giving it up...indeed, our "local variables", when we have some, are likely to go there and the return address is already there because the CPU handles it... ]

> *Task switches*

Oh, boy, how many weeks have you got?

Well, let's strip it down to the basic question: Yes, the CPU stores a task's "context" (all of its registers and so forth ;) into an area of memory (called a TSS: "Task State Segment" :)...so, yes, there's a memory access for a task switch (well, two: got to write the current task's "context" and then read up the next task's "context" :)...and which task runs next is actually decided by the `_scheduler_` (a small piece of OS software that is attached to a periodic timer, whose job is to simply make the "task switches" happen, including working out which task to run next and that kind of thing :)...and, well, the scheduler has its own "tables" relating to which tasks are running, what "states" they tasks are in (they might be ready to execute – "active" – or suspended, waiting on some condition to happen, like a disk I/O command to complete – "blocked" :), what the "priorities" of the tasks are and that kind of thing which is the information it uses to base its design on which task runs next...these "tables" are almost certainly in memory too...

Actually, an awful lot is in memory...because, in protected mode, there's the GDT "global descriptor table" and IDT "interrupt descriptor table" and then there's the "page tables" used for implementing "paging" (typically, this is how most OSes – like Windows and Linux – implement their "virtual memory" schemes :)...all this stuff is also in memory...though, there are actually special "caches" on-chip for some of this stuff and the stuff about L1 and L2 caches all still applies here...

Ah, this answer feels "incomplete" to my standards...but, well, some questions you've asked I don't really know the answers to...and the ones where I've got an answer, it's possible to write entire novels on the subjects...but, simply, this'll do for now: it's early in the morning here and I'm tired :)...

Beth :)