

Re: What do I do with Art Of Assembly?

Source: <http://coding.derkeiler.com/Archive/Assembler/alt.lang.asm/2004-01/0640.html>

From: Beth (*BethStone21_at_hotmail.NOSPICEDHAM.com*)

Date: 01/17/04

Date: Sat, 17 Jan 2004 01:08:59 -0000

Herbert Kleebauer wrote:

> *Beth wrote:*

>> *Truth is, the stuff about HLA above is merely _only_ to correct the*

>> *"myths" circulating about it...I am not really telling you to use HLA*

>> *or not use HLA...I'm just posting to make sure that if you've heard*

>> *"rumours" about this and that, to make sure that _facts_ replace those*

>> *"rumours" instead...I've given most of the assemblers a go and they*

>> *all have their own "charm" and "character"...and, in the end, if you*

>> *learn about ASM and the tool lets you put instructions in a sequence*

>> *and spit it out into an EXE file then, with ASM coding, it's _your_*

>> *knowledge that's most relevent...tools are tools...they just assist*

>> *you in doing that...it really _ISNT_ as big an issue as people often*

>> *make out...but, well, it's one of those "religious" things – people*

>> *with their own likes and dislikes – and gives people something to talk*

>> *about (when not simply totally off–topic ;) while we wait for some*

>> *actual code to be posted or for a proper, reasonable question (just*

>> *like yours here, in fact ;) to appear on the group...*

>

> *Please give us your definition of an assembler (in a few*

> *sentences and not in hundred of pages). Then we can first*

> *discuss this definition an then, using this definition, prove*

> *whether HLA is an assembler or not.*

Okay, here's a "first shot" at that:

alt.lang.asm: Re: What do I do with Art Of Assembly?

Assembly language is an augmented human-readable notation for formatting data more easily for `_direct_` presentation to the CPU...

Note what I have and haven't said in the above...

It's an "augmented" notation because we're NOT only concern with "machine instructions"...this you should know better than most, in fact, as your favourite `_directive_` is "DB"...that's NOT a "machine instruction", it's an assembler directive...other examples would be "BITS 16/32" in NASM or would be implied in RosAsm's "[Variable: D\$?]" (merely a syntactical variant of "DD" there)...there are, of course, further "augmentations" in various assemblers to the "machine instructions" such as macros, section declarations, MASM's "HLLisms" like "invoke" and others...but those first examples I've given: "DB", "BITS 32" and "[Variable: D\$?]" have been selected `_specifically_` to show that `_NO_` x86 assembler is able to live without at least `_SOME_` "assembler directives"...even DEBUG understands "DB" and "DW" in its assembly mode...NASM would be `_completely_` unable to work out the encodings between 16-bit and 32-bit mode without the "BITS" directive (because the correct encoding to be used entirely depends on the "mode" the CPU is in...this `_could_` be set via a command line option but that would stop mixed 16-bit / 32-bit coding, such as a DOS program that goes into protected mode)...

Further, note that I've said "formatting data"...this is because, of course, CPU instructions are `_also_` "data"...they are just "numbers" in memory locations...what turns them into "code" is that the CPU fetches them as part of the code stream and interprets them, according to its encoding rules, as particular instructions...this is another important distinction for things like placing "DB" directives into a code stream to "emulate" a machine instruction that's not natively supported by the assembler...

And this distinction is further of importance because it calls into question what is specifically meant by "instruction", anyway...ask wolfgang and he'll tell you that his special hex-encoder-assembler (much lower-level than most of what we use ;) has the ability to specify whether a particular "MOV" instruction is a "load" or "store" version of that instruction...because, indeed, on the x86, it's possible to encode even this most basic (and other most basic) instructions in `_more_` than one way...we can also look to other commonly used instructions and question them on this facet..."MOVS" is NOT a real machine instruction...there is a "MOVSB" encoding and then there's a "MOVSW/D" encoding (the actual size - 16 or 32 bit - being selected by the segment "mode", the presence or absence of prefixes and so forth)...this is why, on some assemblers, the "MOVS" instruction requires a "dummy" operand to specify the desired size (the opcode itself has NO operands at all, this is part of the "directive" nature of "MOVS" in many assemblers)...

Hence, "MOV" is a `_directive_`..."MOVS" is a `_directive_`...and then, seeing as we've had a discussion about the "JMP" / "Jxx" instructions and the issue of "jump size", then we can also see that this is a `_directive_` too on a lot of assemblers...again, there is a short encoding, a near encoding and a far encoding (for the unconditional "JMP")...furthermore, "JMP", like CALL, is capable of `_direct_` or `_indirect_` operands (e.g. there's an encoding that uses an immediate in the instruction and an encoding which jumps indirectly via a memory address)...

Hence, in total, there are – wait for it – `_5_` different encodings of "JMP" (ignoring prefixes), which is a total of `_9_` different actual forms of "JMP" as 4 of the encodings may take a size prefix to select 16/32 bit, depending on the current CPU "bitage")...does your assembler provide 9 different "JMP" mnemonics? If not, then where's our precious "1:1" requirement? Note, also, that any x86 assembler is going to find "1:1" incredibly difficult because of Intel's own encodings...some opcodes "double up" and may be 16 or 32 bit, depending on `_both_` the prefix and the current CPU mode...worse, the meaning of the prefix reverses (toggles) in the different "modes" ...so, that, simply, due to Intel's own encodings, it is `_impossible_` to determine, from looking at the opcodes alone, what we actually have...it's NOT possible...you may "presume", of course, from knowing that it's Windows code (got to be 32-bit due to the OS rules and the prefixes must, therefore, "flip" to 16-bit mode) or by taking a 16-bit or 32-bit "perspective" on the matter and seeing which is "most likely"...but, strictly, no x86 assembler will be able to make the determination without the use of a "BITS"-like directive or to supply "ambiguous" mnemonics, leaving the "bitage" determination to a human being...

You could once have claimed "1:1" and so forth on a different architecture...but Intel's messy encodings actually mean that this `_cannot_` – when we're pedantic to the facts – be true for any x86 assembler, even when you use "ambiguous" mnemonics (and I've never seen a `_single_` assembler which does use "ambiguous" mnemonics like: "mov (e)ax, (e)bx" :)...not that it matters because, note, that if we've got a constant, then the interpretation is `_crucial_`...is it "mov ax, 1234h; int 21h" or "mov eax, 123421CDh"? Get it wrong and you've messed up your disassembly in a major way...not only are the instructions nothing alike but you've "swallowed" up another instruction if you wrongly take a 32-bit interpretation...

Not even DEBUG can lay claim to being able to operate without directives in its assembly mode – it has "DB" and "DW" – and even if DEBUG was expanded to properly include 32-bit instructions and MMX/SSE and so forth (apparently, someone has done something like this :), it would still not be most people's "assembler of choice", exactly because it's limited...remembering that it doesn't use "labels" and you have to work out all those addresses yourself...

alt.lang.asm: Re: What do I do with Art Of Assembly?

Hence, we've got a pretty damning set of examples: "MOV" (also, "ADD", "ADC", "SUB", "SBB", "OR", "AND", etc. by extension of the same principle :), "MOVS" (also, "LODS", "SCAS", "STOS", "INS", "OUTS" by extension of the same principle ;), "JMP" (also, "JC", "JNC", "JO", "JNO", etc. by extension of the same principle...worse, throw in the "synonyms" too..."JE" is the exact same instruction as "JZ", it's just a convenient synonym...which makes that most certainly a "directive")...and there's MUCH MORE than merely this...but, seeing as we've just covered a massive amount of the basic x86 instruction set, I think the point's been made, hasn't it?

If your definition of x86 assembly includes "1:1" or "no directives" then your definition describes an impossibility...the Intel encodings and suggested mnemonics ("Intel syntax") themselves mean it's not possible...as I was explaining this point to Frank before, it's actually amusing because AT&T syntax better fulfills that (but still can't manage it)...

Plus, our "directives" are pretty powerful, you know...they select the encoding based on the operand in a large amount of cases..."MOV" alone can cover a number of encodings..."MOV al," cuts things down...and "MOV eax," or "MOV ax," are dependent on the "bitage"...as well as the fact that most of these encodings are "reverseable"...and movements to segment register (CS, DS, etc.) or system registers (CR0, etc.) are encoded completely differently and have a much more limited application (can't write an immediate directly into a segment register with "MOV", for example)...yet, all these things are being covered by a single "MOV" mnemonic!! Sounds suspiciously "HLL", in fact...

Your notion of "purity" is actually a myth...the CPU itself and its encodings do not permit it themselves...

But, you know, as interesting as that is, that ISN'T my defence of HLA at all...that's just pointing out, before we even start, that you're trying to argue for something that itself is non-existent...in short, you've failed before you've started because this "purity" only exists in people's imaginations...if you really relate any assembly – that you may deem "pure" – to the actual encodings, there's an awful lot of jiggery-pokery involved..."isn't there?", she says to all those who've played with the encodings and might have attempted an assembler of some kind...

What, in fact, does HLA do? How does it differ from what you call "pure ASM" but I prefer to call "traditional ASM" (in answering this question, we'll begin to see why I make that distinction and stick by it)? Let's compare differences, shall we?

Data declarations, under any assembler, are directives...there is no "Intel syntax" for them nor "standards" nor any agreement...the best we have is "DB", "DW", "DD" and so forth as a loose "de facto"

Re: What do I do with Art Of Assembly?

alt.lang.asm: Re: What do I do with Art Of Assembly?

standard for this...but then again, MASM lets you call these "BYTE", "WORD", "DWORD"...RosAsm calls them "B\$", "W\$", "D\$"...NASM has a different set of names for uninitialised and initialised versions ("DB" vs. "RESB", "DW" vs. "RESW", etc. :)...some assemblers support structures, some don't (plus "unions" too)...and, note, though "structure support" usually includes some "dot notation" thing, it's quite possible to simply allow `_data_` to be declared in structures (for the naming and "sizeof" benefits) but insist that instructions are written in a more "pure" way ("`[ebx + SecondMember]`"), not that there `_is_` any "standard" or Intel ever specify this element of assembly language at all in their documentation...

None of this relates to "machine instructions"...so, all of these variant implementations are as good as one another...and, fundamentally, "data is data is data"...a byte is always a byte, a dword is always a dword...it matters not what programming language you use, this is always the case and data declarations are semantically equivalent...whereas, we couldn't be too sure what the ASM for C's "VarA++" might equate to (more than one possibility there), we `_know_` 100% that "char VarA = 3;" `_is_` equivalent to "VarA db 3"...whereas HLLs may "insert" things and we don't know what's happening "behind the scenes" with the `_CODE_`, there is no such problem with `_data` declarations...

Hence, HLA recognises this and uses 100% HLL-inspired data declarations (but also includes ASM-inspired alignment options too, for precise location of data...note, NO such thing under supposedly "real" RosAsm...it takes care of all the data and its alignment and so forth in totally 100% "HLL" way...for example, in one "title", I finish with a word data declaration and at the start of the "next" title, I have another word data declaration...these "titles" are all listed equally so when I say "next", I'm referring to the location of the "tab" in the control, goodness knows if that actually relates to its location in the source file...further, this data all gets thrown into the data section...there is NO method supplied by this "real" assembler in order to choose where – code or data – nor is a single thing mentioned about "alignment" or "padding" in the documentation...

So, are my two word-sized data declarations aligned to the byte, word, dword, paragraph or page? Don't know...because they are less than 32-bits, does RosAsm "pad" these out to dword-sizes for "alignment" reasons? Don't know...

Hence, we don't actually know with this supposedly "nothing behind the scenes real assembler" whether it's actually "safe" and "correct" to grab `_both_` word-sized data declarations in one dword-sized operation (that is, "mov eax, FirstWord" which "picks up" the second word at the same time because – we Hope though RosAsm does not explain nor guarantee – that the second word is right next door in the data section...and we have no choice at all about it being in a data section, as RosAsm – in a way that's 100% "HLL" to me – deals with

alt.lang.asm: Re: What do I do with Art Of Assembly?

data "behind the scenes"...you can't control what it does, you don't really know what it's doing (alignment? Padding? No explanation of that in the documentation)...

Well, sorry, if that's "pure ASM" in demonstration then HLA is far better...I can put the data wherever I like and I can align it...I `_know_` – both because it's clear from the source and because Randy has detailed all the issues in his comprehensive reference manual and AoA – whether it's "safe" to grab two WORD-sized pieces of data with a single 32-bit instruction...better yet, I can use the "union" declarations to `_specifically write out_` that this data can be accessed in more than one way, giving me the ability to assign meaningful symbolic names and make it totally understandable to anyone else reading that this data gets used as one 32-bit `_and_` two 16-bit values...and there's not an ounce of difference between this "union" and using "DW" twice, then an "ORG" to "backtrack" and declaring a "DD" over the top of them...in a sense, all "union" does is give a much easier to read notation for getting the assembler to do exactly that...needless to be said, there is NOT the slightest difference whatsoever in what data actually ends up in the final file, regardless of syntax..."data is data is data"...

HLA's use of a full and comprehensive "HLL-like" data declarations is the thing that I found absolutely brilliant...formatting out things like "tables of table to pointers to tables to procedures", which `_are_` very useful because many of the best algorithms can use very complex data structures...it's a doddle...you can create a "template" structure with "data", "previous" and "next" pointers and then declare your "doubly linked list" (a common enough requirement for many good algorithms, right? :) with complete ease...doing the same using "DB" alone is a total fudging nightmare in comparison...

What about HLA's "procedure" assistance? You know, don't ask me too much about it, as I never use it...I switch them off with a couple of directives at the start of the source code and then do everything myself...because, you see, it's entirely true...you can simply `_IGNORE_` these things...they DO NOT in any way "get in your way"...data typing? Same thing, in fact...HLA can do it but if you declare things as "BYTE", "WORD" and "DWORD" then the only thing HLA checks is `_size_`, just the same as any other assembler...those silly "IF" statements? Again, wouldn't know...never use them...but I did once have a go for some example code I posted up here and the HLA "IF" is much like the MASM one, except that, in fact, it's `_more limited_` and guarantees that it `_WON'T_` play around with registers, something MASM does NOT guarantee...

The macro stuff, of course, is all `_compile-time_` pre-processing...it all goes on during compilation and is all done by run-time...these are merely just methods of helping a programmer to write their code by wrapping up "common" things in simple macros...HLA macros are more powerful – that is, they provide more facilities for writing them –

Re: What do I do with Art Of Assembly?

alt.lang.asm: Re: What do I do with Art Of Assembly?

but they don't differ (other than syntax) to the essence of macros on any other assembler that has such things...

The machine instructions themselves? Well, the operands are reversed and put into brackets...that's about all we can say there...nothing else is any different about those at all...oh, fair enough, SSE instructions aren't currently supported because Randy didn't have a machine to test that out on at the time...you can, of course, "DB" your own versions and use a macro so you'd never know the difference...many assemblers also don't have SSE, simply because those instructions turned up after the assembler was last updated...and nothing stops them being included, Randy just didn't have them originally for practical reasons (no SSE-capable machine, can't test it so best to leave them out until Randy can get an SSE-capable machine...in the meantime, if it's needed then use "DB", just as you would do with any other assembler for non-native instructions)...

You're fighting a phantom; Because you know what's really "HLL" here? This attitude...things are ONLY "behind the scenes" in your mind because you've not even downloaded HLA and given it a go to see what it is...RosAsm's data declarations are pure HLL because only HLLs give me no choice or explanation of what's happening (heck, C and C++ are lower-level in some regards)... "Intel syntax" is HLL...these notions of "purity" demonstrate that you've not really studied the encodings sufficiently and considered the ramifications of them to realise that "pure 1:1" is a complete nonsense on Intel CPUs...

I'm a low-level, do-everything-yourself, explicit coder and I use HLA...and there's NO conflict between the two whatsoever...I know that's totally NOT what many people want to hear but it's fact and it would be wrong of me to lie otherwise just to support some "phantom world" in your own imagination...to massage your ego that you're a "real" coder, with all the macho superiority nonsense that comes along with that...unfortunately for you, as you'd like it were I "clueless newbie" about this, I was not born yesterday about these things and do know what I'm talking about...I've hand-rolled my own PE files and it's not particularly difficult or complicated...you might think using "DB" to do this makes you some "macho" kind of guy but it's not that impressive...so, you read the PE file format documents? Big bloody deal...

Because, let's cut to the chase here, this has very little to really do with ASM – "pure" or otherwise – it all to do with being the "biggest" in the group...and Randy's sitting at the top in everyone's perceptions so you gain "kudos" as "guru", if you topple him from his position...it's an awkward little thing regards the "guru" status that Randy has read those boring (and they really are boring, aren't they? ;) "decompiler theory" papers and he took the time to understand them...because what's a real pain about Randy getting in the way of this "supreme guru award" is that he takes a professional attitude – he looks these things up and makes sure he understands them –

Re: What do I do with Art Of Assembly?

whereas you want the title but you don't want the work...you'd like to be able to "hack" your way to the top, yeah? "I'm a practical coder and this makes me the best!", right? Except, unlike some of the teachers you may have met, Randy can walk the walk just as well as he can talk the talk...one tends to keep quiet about this for "politeness" reasons but the size and complexity of HLA, the productivity of spitting that out while producing _professional_ quality documentation _and_ he still has time to help people out and fight an argument or two on a few groups (hey, I stick to one group because I couldn't deal with the volume handling CLAX as well...I just "look-in" from time to time now instead :)...he's in a _totally different league_ to most of us...different ballpark, sunshine...

And _THAT_ is what this all really about...whereas other assemblers are rooted in '60s ideas and just try to desparately "extend" that to modern OSes, HLA just said "sod it!" and looked again at assembly with a modern perspective...but Randy _has_ stayed true to the roots and it is a _true assembly language_ at the core...and the "old guard" are just disturbed that they are "stuck" with the old ways and can't comprehend the "new"...

This is simply an actual demonstration of the famous saying: "New ideas never succeed, they simply wait around for the old ideas – and the keepers of the old ideas – to eventually die off"...

Beth :)