

Re: Evolution

Source: <http://coding.derkeiler.com/Archive/Assembler/alt.lang.asm/2003-11/0319.html>

From: Randall Hyde (randall.hyde_at_ustraffic.net)

Date: 11/19/03

Date: 19 Nov 2003 12:38:32 -0800

Herbert Kleebauer <klee@unibwm.de> wrote in message news:<3FBB743E.2146D6AF@unibwm.de>...

>

> *What you describe here is a HLL: you have the freedom to use all from
> OOP, simple HLL constructs (like if for while do ..) down to
> simple processor instructions. And that is surely the best way to
> write applications. Use a C++ compiler to write the main part and
> implement only critical sections in assembler (small parts with
> inline assembler and larger parts by linking to an assembler
> generated object file). But in these optimized assembler parts
> you surely don't need HL constructs (otherwise it wouldn't have
> been necessary to implement this part in assembler).*

in-line assembly often causes more harm than good. It's **only** useful for achieving small things that simply cannot be done in the HLL. It's generally useless for real optimization as it tends to freak out the optimizer in the HLL (i.e., every cycle you save in the in-line code gets lost because the HLL compiler cannot optimize across the in-line assembly sequence).

Linking large assembly blocks with large HLL blocks is a reasonable way to go, but often it's more convenient to work in a single language, particularly in smaller projects. This is where having the HLL-like constructs in an assembler like MASM, TASM, and HLA is really useful – you can decide on a statement by statement basis how you want to write the code: high-level or low-level. You get to see every machine instruction that the system emits. You control the semantics, exactly.

>

> *The use of libraries has nothing to do with the way (low level
> or high level constructs) the source for the main program (which
> links to the library) or the library itself was coded.*

No, but having a decent set of libraries available is one of the main reasons assembly is experiencing a resurgence. Take away the C/C++ libraries from the C/C++ programmer, and the result isn't a whole lot prettier than working in assembly language. Add those same libraries to assembly

language, and assembly is almost as nice as working in C/C++. In the past, the *big* dichotomy between HLL and assembly programming has been the availability of standardized sets of library routines. Today, assembly programmers enjoy this same facility.

> *Macros*
> *are a different thing. The use of macros may simplify the writing*
> *of a program but is also dangerous: often you are not aware how*
> *many code is generated when you use a macro (especially when*
> *the macro is from a macro library which you have written a long*
> *time ago). But if you would have to manually include the code*
> *instead of using a macro, you would see, that sometimes it is*
> *much better to use a subroutine and call this subroutine than*
> *just inserting the same macro again and again. Because macros*
> *are often as general as possible, you maybe can save one*
> *or two instructions (e.g. save and restore a register) if you*
> *don't use this general macro but insert the code directly.*
> *So, either you have to optimize your code for every byte and*
> *nanosecond, then don't use macros or there is no need for this*
> *optimization, then don't use an assembler but a HLL.*

What a great argument.

1. You should be using a high-level language because it's more productive than using assembly.
2. If you're using assembly, you should avoid anything that makes assembly coding more productive.

I realize that you're a C/C++ apologist, but the above is nonsense. If macros and other assembler facilities give me the ability to quickly crank out some section of code, whose performance isn't all that critical (without the bother of having to switch to using multiple programming languages), who cares if there's one extra instruction or not? And if a macro is continuously generating more code than it should in typical invocations, maybe it's time to review that macro and extend it to handle those cases better. Of course, you do need a decent "compile-time language" to pull this off – so you'd best be using a decent macro assembler like MASM, TASM, or HLA if you want to write efficient macros.

>
>
> > *assembler. Effectively I want an assembler that is powerful enough to*
> > *emulate other coding styles rather than restrict what I write to*
> > *someone elses limited vision.*
>
> *Why do you want to emulate other coding styles? Just use*
> *this other coding styles (your C or Basic Compiler) and only*
> *use assembler where this other coding styles are not appropriate.*

Given a choice of learning a single language that lets me emulate several different programming styles, or learning several different languages and combine their output into a single program, I'll choose the one language approach everytime.

I don't know your background, but you sound like a real "C/C++ is the perfect language" kind of guy. Truth is, there are lots of other languages out there that handle their own problem domains quite well. For example, I would never consider using C/C++ for a problem for which SNOBOL4, APL, or Prolog is uniquely suited. OTOH, with the appropriate "macros", it's quite possible to solve problems these languages are great at in assembly language. And this is real important – because many of these languages do not produce object files that you can link with other programs (indeed, many of these languages are *interpreters*).

If all you wanted to do was solve the same problems that C/C++ solves, you'd probably have a good argument. But C/C++ isn't the be-all/end-all programming language. Sometimes you need to solve problems for which C/C++ isn't ideally suited for, and assembly does a much better job on those problems than C/C++ will. Particularly if you have the right "macros" available.

>
> *Size:*
> *Do you think it is a positive marketing argument, when you say your new game fits on 3 floppies instead of 3 DVDs? And you can only reduce the size of the code and not the size of the data by programming in assembler. How many percent of a big Windows application (game, compiler or office packet) do you think is code resp. data? Even if you can reduce the code size by 20% how much would reduce this the overall size?*

Think: keeping things in cache lines.
Think: keeping things in MMU pages.
Think: controlling the alignment of everything.
There are more facets to size than making the smallest possible program.

As for the data size argument – the big issue isn't whether C/C++ *could* use as little data as an assembly program, but whether this is *typically* the case. As you noted earlier, assembly code (macros or otherwise) force the programmer to look at what they're doing. The same applies to data as it does to machine instructions. Assembly programmers are *far* more likely to choose more compact data structures than HLL programmers because they see the effects of those choices being made. Also, a construct like "**p=++i;" often gets replaced by something else when the programmer sees the number of instructions this code explodes into.

>
> *Speed:*

- > *You must be a very good assembly programmer to compete with the*
- > *code generator of a modern compiler. And "good assembly programmer"*
- > *means, that you know the processor architecture very well (which*
- > *instructions can be executed in parallel, how to avoid pipeline*
- > *conflicts or cache misses and so on) and not that you have a good*
- > *grasp of the assembler syntax. And I doubt, that somebody who try to*
- > *learn assembly programming with HLA will become a "good assembly*
- > *programmer" in that sense.*

Absolute nonsense.

Without even trying I still write programs that outrun comparable C/C++ programs. And this is without using any of that knowledge about pipelines and caches. There are two kinds of compilers in this world – the ones the CPU manufacturer creates to perform benchmarks for a particular CPU, and the ones everybody uses to write actual applications. The problem with the compilers provided by the chip manufacturers is that they work real good *on the ONE CPU they were written for*. They don't do so well when you run that code in a different CPU with a different internal architecture. In practice, compilers like VC++, BCC, and GCC do all the standard architecture-independent optimizations real well, but attempts to create code optimized for a single processor don't work well and they certainly don't transport well.

A friend of mine wrote an optimized floating point matrix multiply in assembly a few years ago. It was optimized for the Pentium II, IIRC. He also wrote the same code in hand-tweaked C. The assembly code was much faster (as you can imagine). Over the years he has recompiled the C code with newer compilers (PIII, PIV, etc.) and compared the results of the newly-compiled code against the original assembly code. The assembly code still wins hands-down (yes, the gap is narrowing a bit, but the PII code running on the new processor is still faster than the C code compiled for the new processor).

While in special cases you can achieve some amazing things by considering architecture issues, the truth is that most performance gains in assembly language are achieved by "thinking in assembly" rather than thinking in a HLL.

If an "assembly" programmer writes "C code using MOV instructions", then they're not going to achieve decent performance. This is true whether you're using HLA, MASM, FASM, SpAsm/RosASM, whatever. Conversely, however, the fact that an assembly programmer uses HLA does not mean they don't know how to think in assembly. The fact that learning assembly via HLA makes learning assembly a whole lot easier doesn't change the fact that they're learning

assembly. As you said yourself – the syntax is immaterial. It's the semantics that are important and *all* assemblers share Intel x86 semantics; this is strictly specified by the architecture in use, not the assembler.

- >
- > *So if you want fast code, write all the parts which are not time critical in a real HLL (and not in a "want to be HL" language like HLA or MASM) and use the saved time to optimize the time critical parts by using low level assembler code.*

Except, of course, for the time, knowledge, and capability to link those HLL programs with your code. Once again, this process is fine if all the problems you solve can be easily solved by C/C++. This logic falls apart when you start considering other HLLs.

- >
- > *Yes, the "21st century assembler" is called C++ or Delphi or*

Utter BS.

A C apologist would like you to believe this. But this is no more true today than it was 10 years ago or 20 years ago.

- > *But if you need uttermost optimized code (either size or speed) you still have to do "middle 80s style coding" by using processor instructions and nothing else in your source code.*

Even when those "middle 80s style coding" sequences produce the exact same machine instructions as those using a modern high-level assembler? Again, nonsense. A good assembly programmer understands the code an assembler will generate for a high-level control structure and uses that construct (for readability reasons) whenever s/he'd wind up writing the exact same code anyway.

You're a big fan of in-line assembly in a language like C/C++. It lets you "drop down" into assembly when appropriate. High-level assemblers provide the complementary ability, the ability to "rise up" into a HLL language, as appropriate. In both cases, this facility helps avoid mixed language programming for certain applications. And that's a good thing.

- >>
- > *I think, this is the main reason to learn assembly programming.*
- > *So you have to differentiate between an assembler optimized for learning assembly programming and an assembler for writing applications. To learn assembly programming in the first place you don't even need an assembler, the processor user manual is sufficient (but the other way around is not possible: you can't be a good assembly programmer without reading the processor manual).*

Too bad everyone else in the world isn't the total genius that you appear to be. It's great that you can learn assembly language from square one with nothing more than a processor manual. Most people don't find this very practical. Even those who are capable of this, find that learning assembly language is far more efficient when they've got a decent book to learn from. If this weren't the case, neither Jeff Duntemann nor myself would be making any money selling assembly language books and Webster wouldn't have racked up nearly four million hits by now.

And reading the processor manual is **not** a prerequisite to becoming a good assembly language programmer. Most of the material found in Intel's processor manuals, for example, is found in lots of other places, and presented in an easier to digest form.

> *But it is very boring to only read the manual without practical*
> *testing. But you should only use processor instructions as*
> *outlined in the processor manual and no HL constructs. You should*
> *directly write instructions for the processor and not instructions*
> *for the assembler which are used by the assembler to generate*
> *instructions for the processor.*

Now let's consider the rest of the world. They know a HLL. They want to learn assembly. You're suggesting "hey, just read this processor manual and you'll learn everything you need to know. Oh! That's too hard, need to actually write code to figure this stuff out, well after you read and digested those hundreds of machine instructions, start writing code — it'll come natural to you."

Sorry, the average person doesn't work this way. They like to build their knowledge a step at a time based on what they know already. The **one** thing I've learned by teaching assembly language for many years is that the approach of "if you do it in C this way, this is how you do it in assembly" works **real** well for beginners. Using high-level constructs is great because it allows a beginner to use their existing knowledge to delay learning about conditional jumps, comparisons, and how to simulate the high-level constructs they already know. Instead, they can **focus** on a smaller set of instructions, like MOV, ADD, SUB, MUL, etc., and get comfortable with those instructions before learning how to do low-level constructs.

Not everyone is as brilliant as you are, I'm afraid. They can't read through a processor manual once or twice and walk away as expert assembly language programmers. For everyone else, using high-level control structures is a great way to start when covering all the other prerequisite information someone must learn prior to mastering all the low-level machine instructions. If your way of doing things

is the best way to do it, how come every college and university in the nation isn't simply specifying the Intel Processor manuals as the textbook for their assembly courses?

- >
- > *You also need an operating system which allows you to experiment*
- > *with all the processor instructions, so Windows is inappropriate.*
- > *You need the processor in real mode, so best is to use DOS without*
- > *EMM loaded. DOS isn't obsolete to learn assembly programming, which*
- > *is the generation of a sequence of processor instructions and has*
- > *nothing to do with operating system (the processor knows nothing*
- > *about the running operating system, a call to the operating*
- > *system is nothing but a INT or CALL processor instruction).*

This is utter nonsense.

How many introductory students, for example, need to learn about the LGDT instruction? The x86 architecture has several hundred instructions. *No* assembly course on this planet covers them all. Instead, they focus on those instructions that are common to most programs. The time for learning about instructions like LGDT is when you're writing an Operating System.

As for real mode, gee, many of the instructions don't make sense in real mode (e.g., LGDT), so your comments above are contradictory.

You are absolutely correct that the choice of OS is mostly irrelevant when learning assembly. However, in order to do any I/O at all, the student is going to have to interface with *some* operating system *somehow*. There are two ways to do this: OS API calls or a "wrapper library" that hides the APIs from the student. One need only subscribe to the Linux assembly mailing list, or read some posts around here to find out what the problem with teaching DOS APIs are: people try to write code under Windows or Linux using statements like

```
mov dx, offset somestr
mov ah, 9
int 21h
```

and then post a question asking why they get a seg fault (under Linux). Then they get real annoyed when they discover that they spent all that time learning the DOS API calls and find out that they can't call them anymore.

The second approach, supplying students with a set of standardized library routines is moderately better. Yes, the students still waste time learning the semantics of those library calls (I can remember students emailing me to ask me why UCR stdlib calls weren't working in their

embedded systems). OTOH, the library routines *can* be ported so that the knowledge is transferable. For example, the HLA Standard Library is currently portable across Linux and Windows (with BSD and QNX ports in the works). Just like learning the C standard library, the time spent learning the HLA Standard Library transfers across operating system, thus making the choice of OS less relevant. The HLA Library has even been ported to MASM and FASM. Thus, those who learn HLA and find themselves working with another assembler can still use familiar calls.

The other problem with DOS/real mode is segmentation. Segmentation is a relic at this point. Flat model programming is much easier to learn and understand and the need for segmentation is pretty much gone these days. You cannot write DOS/real mode code without learning about segmentation. What a waste of time. Time that could be put to better use learning other topics of importance to the assembly student.

> *Now we are by assemblers for writing applications. It is hard to write a real application with a real assembler.*

Certainly if you apply all the constraints you've listed. Just as HLLs have evolved, to make programming easier, so have assemblers. To say that programmers must stick to 1960's style or even 1980's style assembly, or else it's not assembly, is foolish. Assembly programmers deserve to use modern tools, just like HLL programmers. The difference is that an assembly programmer has complete control over their program, a HLL programmer does not.

And yes, there *are* applications written in assembly. Unless, of course, you wish to speak in "Betovian" and carefully define "application" in such a way that the programmer cannot write an application in assembly.

> *To make it easier*
> *you can add macro support and/or some simple HL constructs. But*
> *it would be stupid to stop here. Just go on and you will end by*
> *a HLL like C or C++.*

PL/M-86, for example.

Regardless of what people add to an assembler, a product is an assembler by virtue of the fact that it lets the programmer have complete control over the machine instructions that the translator generates. It can add all sorts of things beyond that (e.g., HLL-like control structures). But as long as the programmer can ignore all that and write "pure" code using nothing but machine instructions, and the translator adds nothing else, you've got an assembler for an assembly language.

- > *There is no need and no market for an assembler*
- > *for writing applications (I'm not speaking from embedded systems*
- > *or driver programming, but applications for desktop systems).*

In your opinion, maybe.

Others have a different opinion.

Most of the MASM, FASM, HLA, SpAsm/RosASM, NASM, GoASM, etc., code I see being written these days is application code. Not device drivers. Not embedded systems code.

Indeed, as Rene constantly points out, RosASM and FASM are both written in assembly. Do they not count as applications?

Visit Steve Gibson's site sometime.

- > *I also can't see a "success of MASM in 32 bit Windows". Show me*
- > *the Windows applications written in MASM (which are not only written*
- > *just for fun). If there were a market for such an assembler, you*
- > *can be sure MS wouldn't give it away for free.*

Win32 programming in assembly is slowly building momentum.

The problem, you see, is that Intel's processor manuals plus the MSDN listing of the Win32 APIs are insufficient documentation for most people. They aren't going to write Win32 assembly programs if it's terribly painful just to figure out **how** to write such programs. Today, we have the pioneers and early adopter types writing Win32 assembly code. Slowly, documentation is appearing (e.g., the Iczelion tutorials and http://webster.cs.ucr.edu/Page_win32/0_win32asm.html), and slowly tools are being developed to aid in the development of Win32 assembly. But it is gaining momentum. Largely on the backs of high-level assemblers like MASM and HLA, I might point out.

- >
- > *I fully agree, writing DOS applications is obsolete. But I asked*
- > *about learning assembly programming. And for this purpose there*
- > *is no better operating system than DOS (it is great if you can*
- > *use hardware breakpoints by using the processor debug registers*
- > *in your debugger).*

???

Hardware breakpoints are available in debuggers under Windows and Linux too. If what you're saying is true, a lot of OllyDbg users are going to be quite surprised to find that they haven't really been setting hardware breakpoints. Both Win32, Linux, and most other modern Oses make all of these facilities available to debugger programs. DOS has **zero** advantage in this respect. Of course, one thing **great** about a protected mode OS is that the errant application can't walk all over the debugger and crash it during debugging :-)

> *I never understood why people want to read second level documentation.*

Well, you're just an absolute genius. But you can't expect everyone else to be as smart as you are. Some people prefer to read "second level documentation" because their time is more valuable to them and they learn the material much faster by reading that documentation. But then, this is getting redundant, eh?

> *If you want to learn assembler programming, forget about all the
> assembler books and read the x86 processor manual (or better read it
> twice).*

And three, or four times. And then you still won't be able to print a single character to the display or read a character from the keyboard because *no where* in the processor manual is it going to tell you how to do this.

> *Then select a simple assembler (or better write your own,
> if you do it without macro support and without FPU, MMX, SSE instructions
> then you only need a few lines of C code and you will learn much
> more about the instruction set than just using an existing one)*

Ah yes! Not only must they be brilliant enough to grok the processor manuals, but they should write their own assembler too! "Just a few lines of C code..."

You'll certainly learn a whole lot about the instruction set by doing this. It will not make you an assembly language *programmer* however. Knowing the encoding of all the instructions (about the only assembly-related knowledge you gain by writing an assembler) is fine knowledge to have, but that knowledge doesn't tell you how to put those instructions together to solve some problem.

> *and start writing your first program. All you need for this is the
> documentation how to read from the keyboard and stdin and how to write
> to the screen and stdout (and that is much easier in DOS than Windows).*

Oh, so you *do* need something besides the processor manual? And your assertion that stdin/out I/O is much easier in DOS than Windows is absolutely false. Even if you don't use prepackaged library routines to do this work for you, in Windows you're only talking about calls to the read and write API functions. Not difficult at all.

> *As I see it:
> Randy has to give an assembler course at the university. Nowadays
> students are not interested in learning assembly programming. So
> what can he do? He designs a simple HLL with good inline assembler
> support and calls it "High Level Assembler". Now all are happy,
> the students don't need to learn assembly programming, he gets
> enough students for his course and his employer still thinks the*

alt.lang.asm: Re: Evolution

- > *students learn assembly programming (because of the name HL Assembler).*
- > *That isn't a problem at all, but sometimes there are people who*
- > *really want to learn assembly programming (e.g. in this group).*
- > *And if these people use HLA then in the best case this is just a*
- > *waste of time but in the worst case they really believe, that the*
- > *have learned assembly programming and will become much trouble*
- > *later, when they need assembly programming in there job.*

As you know nothing about my courses, very little about students in general, nothing about my students in particular, and, obviously, almost nothing about HLA, what gives you the impression you have an educated opinion about any of this?

I'll give you the benefit of the doubt and *assume* that you learned assembly *exactly* the way you're proposing it should be taught. I'll also assume that you're the "expert" assembly programmer you seem to imply that you are. What on Earth makes you think that the pedagogical methods that worked fine for you are going to work for everyone else?

If you *know* how assembly language should be taught, then I strongly suggest that you do the same thing I did back in the days when *I* knew how it should be taught – go get a job as a lecturer at the local University and teach the course. I think you'll find out in short order that your pedagogical methods don't work at all. You can bitch and moan about students not wanting to learn and all that other crap, but at the end of the quarter you can only measure your success by how many machine instructions the students know how to use and how adept they are at putting those machine instructions together to solve programming problems. All this other nonsense about OSeS, hardware, real vs protected mode, using HLLs, writing your own assembler, etc., is exactly that: nonsense. The bottom line is whether the students can write machine instructions (and no HLL-like statements, I might add) by the end of the quarter. My experience with HLA suggests that the students learn quite a bit more with HLA under Windows than they do using MASM under DOS. And this is with the HLA students using HL-like control structures to begin with (and getting weaned off of them as the quarter progresses). You can complain all you want about how bad this approach is, but it sure works for me. If you want to prove your way is better, feel free to teach the course yourself sometime. Until then, who do you think someone else is going to believe? You (who has never taught an assembly course) or me (who has taught assembly at the University level for over a decade)? Until you've actually taught the subject and encountered the problems, you're not in a real good position to comment on teaching methods for assembly language programming.

I strongly encourage you to get so annoyed that you decide to teach assembly for yourself. I bet that after a few years, you'd be switching to HLA, too.

- >
- > *No problem if he call his tool SHLL (Simple High Level Language)*
- > *instead of HLA. Then it is clear, that you learn just an other*
- > *HL language (which is simple and therefore can't compete with*
- > *a real HLL) and nobody will have the wrong presumption that he*
- > *now is a good assembly programmer.*

Except you're not learning another high level language. You're learning assembly. The whole point of the high-level control structures in HLA is that the students *already know them*. It would be a complete waste to teach these control structures and their associated semantics to students who don't already know them. The point behind those HL-like control structures in assembly is that students can use them as a crutch to avoid learning CMP/Jcc sequences while they're first learning assembly and then they can replace the IF/WHILE/etc statements with real machine instructions later in the quarter. This lets them start writing *real* applications their first week of the course. Of course, you've made it clear that you don't believe an assembly student really needs to write much code; just read the processor manuals through once or twice and they're in business. In the real world, however, students learn by doing. I've met very few students who could learn *any* kind of programming by reading a book. They had to write code; lots of code; the more code the better. That's where HLA comes in. By leveraging their HLL knowledge during the first few weeks of the course, before they've had time to learn the complete instruction set, HLA allows the students to write more code than they otherwise would. And that's why they wind up being better assembly programmers at the end of the term.

- >
- > *I fully agree, but you shouldn't use the name "assembly programming"*
- > *for this diversity.*
- >
- >> *I suggest that assembler programming is poorer when this diversity is not present.*
- >
- > *No. Programming is poorer when this diversity is not present. Assembly*
- > *programming (the art of generation an optimized sequence of processor*
- > *instructions) is just one method of this diversity.*

So, IOW, everybody has to write assembly code the way *you* write it, or it's not assembly. Gee, you sound just like Rene. And just like Rene, you'll be much happier in life once you accept the fact that there are lots of *assembly* programmers out there who don't write code the same way you do. Any attempt to try and classify them as "not assembly programmers" is foolish. They'll keep right on writing assembly code, and laughing at that

alt.lang.asm: Re: Evolution

poor guy who wish the world would following his example.

Cheers,

Randy Hyde