A-LANG: WAYS TO IMPLEMENT COMPUTER LANGUAGES ON 6502s

David A. Wheeler

(original at http://www.dwheeler.com/6502/a-lang.txt)

Table of Contents

- A-LANG: WAYS TO IMPLEMENT COMPUTER LANGUAGES ON 6502s
- BACKGROUND/THE PROBLEM
- THE SOLUTIONS
- SOLUTION 1: Overloaded static parameter/local addresses - ASM+ and Simple.
- SOLUTION 2: FIXED LOCATION DATA STACK
- COMPARING THE TWO APPROACHES
- Notes on creating other languages:
- REFERENCES:

Here are some of my ideas on how to create efficient and easy-to-use programming languages for 6502-based computers (such as the Apple II, Commodore 64/128, Atari 400/800, and the original Atari game machine).

I haven't tried to implement these ideas at length, since I haven't directly used Apple II's (or any other 6502-based system) for some time (though there was a time I could rebuild an Apple II from memory!). For me, this is simply an intellectual exercise. I did create a partial prototype of some of these ideas, particularly some of the linking techniques.

Still, this was an old problem that a friend of mine (Bob Pew) & I often hashed out. Bob passed away just when I completed writing the first draft of these ideas, and I was about to call him about my new ideas for this old puzzle we'd often chatted about. I miss him; perhaps this little file will serve as a small memorial to him.

Perhaps some reader of this file actually DOES decide to do this. If you do: (a) are you SURE you don't have better things to do? (b) please let me know, I'd love to hear about it.

Who knows; maybe the ideas of an "optimizing assembler" and "compiling linker" can be taken to other systems. It makes some sense, especially for systems where compilers are quickly written & can generate assembly, so that poor compiler code can be made a little better. It's possible these ideas have been created before, especially on far older systems.

These are essentially notes, and are a little rough in places. Sorry about that, but I think you'll still find them useful.

--- David A. Wheeler


BACKGROUND/THE PROBLEM

The 6502 is a weird beast with a very irregular instruction set. Bob Pew & I wanted a language that would be (a) faster to program in (than assembly) but be (b) fast when running. It needed to be pretty efficient. This turns out to be an intellectually challenging problem; below are (what I think are) some solutions.
There are many odd things about the 6502 that make it hard to do this: + It's REALLY an 8-bit processor. It doesn't even have 16-bit data registers or instructions (many other 8-bit chips, like the Z80, at least had these). That means that, where possible, use 8-bit manipulations instead of 16 or 32 bit data types. + Its instruction set is rather irregular and has a few odd addressing schemes (esp. "zero page"). + Its built-in stack (for subroutines) is only 256 bytes long - possibly enough for storing the return address, but probably not big enough for storing data as well.

A key to using the 6502 is to keep as much as possible in the "zero page." The zero page is the first 256 bytes of memory; the 6502 has a large set of special operations that use this area of memory, in many ways like an extended register set. Loads and stores are faster to the zero page, and some operations (such as indirect addressing) REQUIRE use of the zero page.

THE SOLUTIONS

I have developed 2 new (to my knowledge) solutions to this problem: 1. Statically assign parameter and local variables in a special way. To do this, divide the work of compilation & linking in an unusual way (the linker does a ton of "compilation" work) to efficiently use 6502 resources, and in particular assign all local variables and parameters to static addresses. Global analysis is then used to assign addresses so that, in most cases, all copying between locations (e.g., between the addresses for parameters and addresses for local variables) are essentially optimized away. This could probably be best utilized by creating a different language, so it then discusses a potential ASM+ & Simple, a pair of languages that squeeze as much efficiency out of the 6502 as reasonably possible. However, the approach could be used by C, etc., especially if the programmer is willing to accept certain conventions and extensions (e.g., extending C to support pass-by-reference like C++ and then writing 6502 code that uses that extension in most cases). This approach can support recursion - even automatically - but there is a cost where the recursion occurs. 2. Fixed location data stack. This would be easier to implement & would support recursion more easily. This approach wouldn't be as fast or small as the previous approach, but it would support a more traditional approach (linking could just link, compiling could just compile). This approach would be particularly good for FORTH-based languages.

It's hard to argue that these are "always better", but I suspect they're better in at least some circumstances than other approaches. Stack-based approaches that use a zero page two-byte pointer to an arbitrary stack are flexible, but SLOW on a 6502 and create large code sizes. Anyway, if you're implementing these kinds of systems, you might want to think about these ideas.

SOLUTION 1: Overloaded static parameter/local addresses - ASM+ and Simple.

This approach creates overloaded static addresses for parameters and local variables, in a special way that maximizes use of the zero page. It's probably easiest if specialized languages are designed to exploit this property, so I also discuss 2 specialized languages to do so.

The approach below uses the zero page as much as possible, overloading each zero page location with many different variable values (as long as they aren't used simultaneously). In fact, it's difficult to believe that normal humans could overload its usage so well, so this approach has the potential to be in some ways more efficient than humans.

This approach doesn't _require_ specialized language, but it might be easier and more flexible that way, and originally I had in mind the idea of creating a high-level language specifically to be easier to efficiently implement on a 6502 (of which this idea would be part). Two specialized computer languages are discussed here:
ASM+: A macro assembler with 2 peephole optimization stages and a special macro language with common operations (increment integer, etc). It can be used as a "smart" assembly language directly, or as an intermediate compilation language generated by a "Simple" compiler.

Simple: A higher-level language that would be C (and Ada) like, but would pass by reference (like Fortran) instead of by value (like C). Why reference-passing, when the 6502 poorly supports arbitrary pointers and copies? The answer is that, with this new overloading approach, reference passing could often be implemented by a copy that never actually happens (via address overloading). Since the 6502 doesn't have a usable data stack, this approach is more efficient on this platform (in the implementation, the "reference" is implemented not necessarily by copying values, but by having exactly the same address assigned to the parameter and the variable calling it, creating "zero-copy" calls where possible). Recursion could still be done, though recursive calls could be expensive when executed. By using Ada-like parameter passing (noting IN, OUT, and IN OUT) passing data is MUCH more efficient than for C, because most "copies" could be eliminated.

Creating code when everything was implemented would go through the following steps (with product types in square brackets):

[Simple Code as ASCII text, created by a programmer] -> Simple compiler. -> [ASM+ Code as ASCII text] -> Special preprocessor and/or done as part of the "linker" step -> [List of procedures & what they call; also input/output params, globals, and a list of variable sizes which need allocation] ->

(At this step "compilation" is done. Now to link:)

Run "linker" across all ASM+ code - first determine what procedures call what & then do a topological sort. Then determine address locations of all variables (including variables used to pass data between procedures), trying to use zero page as much as possible. Generate a file of all address locations. I've written a quickie "demo" program that handles the topological sort and shows that you really can stick in a lot of variables into zero page if you do this (it also generates a huge batch of EQU's that a simple assembler could use to support this idea). Note that you also want to "prefer" a memory allocation that eliminates copying memory for parameter passing; this is possible by judiciously "overlapping" memory locations from different calls. Pass one of ASM+ assembler. Transforms "early" macro expansions, then does a peephole optimization pass on the assembly-with-some-macros. Pass two of ASM+ assembler. Transform all normal macro expansions into 6502 code, then do another peephole optimization.

[Executable Code]

"Compiling" a Simple or ASM+ file is relatively cheap. Linking on larger systems would get expensive, since there would be some global analysis followed by assembly of everything. This is necessary to maximize zero page usage.

The ASM+ assembler could fit in the restricted 6502 memory. The assembler would need to keep the macros, global definitions, jump addresses of procedures, and addresses for parameters in memory at all times; for each procedure definitions could be kept & then dropped. Code could come in & out via the disk. Thus, this could be self-hosting.

The procedure call system should be by-reference, since this is cheaper on a system without a stack and where "copies" aren't really copied. By-value can be supported too, but the goal is to minimize actual copies. Values are simply written to fixed locations (preferably in the zero page) and the procedures are then JSRed. Return values are handled the same way. Recursive calls can be handled, though expensively: copy the values that might be changed onto a stack and/or malloced area, call the recursive call, then pop the values off the stack. This can be noticed at link time; by simply following the call tree you can see which calls "loop back", and wrap those calls with the commands to save & restore the variables covered by everything memory location possibly used "between".
Implementation order: naturally not everything need be built at once. The ASM+ assembler could start out as a stock macro assembler, and just create macros for it in the stock macro language. The macro language would need the ability to do .IF's during expansion, since it would need to pick from a number of expansions depending on whether or not certain values were constant, in the zero page, etc. Thus only the "linker" would need to be built, which would simply examine for special macros that define procedures, etc, and generate addresses for everything. The peephole optimizers could be built later.

Bob originally wanted an ASM+-like language, and he always liked Fortran-- I think he would have liked a Fortran-like parameter system!

Here's a sample ASM+ syntax:

"I" is used for (2-byte) integers, "C" for (1-byte) characters, "L" is used for (4-byte) longs, "P" for (2-byte) pointers, "R" for records of other sizes.

Results are always the left-hand-side argument (so that later Simple operations will look similar).

I++ A  increment integer A.      INC A ; BNE .1 ; INC A+1 ; .1
C++ A  increment character A. INC A
I--
C--

I:=I A,B  copy B into A.
C:=C A,B  copy B into A.
I:=C A,B  copy B into A.

I+=I A,B
C+=C A,B
I+=C A,B
also -=, *=, /*
I:=I+I A,B,C
also -, *, /, and for C as well.

MEMCOPY

IFI==0 A, BR
IFI==I A, B, BR
also !=, >, >=, <, <=
also for C.

And also some "combined" macros, which expand into optimized 6502 instructions (some for loops, for example):

I---=0 A, BR     Decrement A; if not 0, branch.
I++!=I A, B, BR   Increment A; if not B, branch
I++<=I A, B, BR   Increment A; if less than or equal to B, branch.
IZERO A           Same as I:=I #0.

FOR ... various forms, inc. some which use the Y register for fast loops.
CALL location

To call another procedure, use I:=I etc to copy values into the procedure’s IN and INOUT parameters, then use CALL to call into it. A processor will examine ASM+ files to look for CALL, IN, OUT, INOUT, FUNC, and PROC commands.

The first peephole optimization step would try to fold constants, combine copies together (so that they're all done at once), and combine separate macros into specialized combined macros (like the ones listed above). Example: I--; IFI==0 becomes I--==0 I:=I B, #0 becomes CLEAR I B

The second peephole optimization step would try to optimize combinations of assembly instructions to eliminate extra work. For example: LDA #!XXX; STA ?A; LDA #?B; STA ?C ; LDA #!XXX becomes LDA #?B; STA ?C; LDA #!XXX; STA ?A; so loading sets of variables with constants is faster. For example, if the constants are <256 this would combine the high byte 0s. If the constants were identical (say 5), similar approaches could combine all the loads into pretty reasonable items.

JSR X ; RTS => JMP X

Two peephole steps could be used - one to see "wider" areas of code (collections of copies, etc.), and the second one to cover special-case asm instructions, etc. As a quick-and-dirty implementation, lex/flex or a regular expression engine could be used to implement the peephole optimizations.

Now for the "Simple" language. It is to generate the ASM+ language. It's designed to be simple to parse (probably by recursive descent or LL(1), since they’re easy to build & small). It would be easier to create if, at first, the language were VERY restricted.

The syntax I've put down is very C-like, but with some Ada & Pascal influences. Ideally, with the "right" definition files or simple text preprocessing, it should compile on a stock compiler on other machines.

The syntax should also fit within the target machine limits. For example, old Apple II's can't generate curly brackets ({} or underscores (_). I would substitute the dollar sign ($) for _, and use instead of {} - again, to support self-hosting.

As far as parsing it goes, recursive descent is often easy to understand, but I suspect an LL(1) parser would be better (see Fisher, page 120). Eventually it'd be good if Simple were written in itself, and using an LL(1) parser driver would mean that recursion wouldn't be necessary (which would be an expensive operation in Simple). Also, a table-driven parser would be much smaller (inserting comparisons everywhere would make the compiler large) & according to Fisher they tend to be faster too (that's surprising). The table could be used to at least list what are legal tokens where an error occurs, and for LL(1) it's easy to add an automatic error repairer.

Down the road, it'd be better if there was a way to specify that some parameters get passed in a register, and/or fix certain addresses. This could be used to reduce link-time, -- precompile large modules, and could be used to create really nice interfaces to underlying OSs. While it might be difficult to generate code that used the params in registers, it'd be useful to low-level routines -- allow direct calls to assembly-level procedures.

File = External_Statements*

External_Statements =
Global_Declaration | Subprogram_Definition

Global_Declaration =
Here's a sample Simple program:

```plaintext
private char current_x, current_y;

procedure move$cursor(in char x, in char y)
[
    int b = 0;
    // Move the cursor to the new position x, y.
    x++;  
y += current_y;
    b += 0x1fff;
];
```

Which would generate the following ASM+:

```plaintext
MODULE LOCAL$FILE
PRIVATE-BYTE LOCAL$FILE.CURRENT_X
PRIVATE-BYTE LOCAL$FILE.CURRENT_Y

PROC MOVE$CURSOR
IN X,1 ; second parameter is size
IN Y,1
ALLOC B,2
; so far we've just been allocating space. Here's executable code!
IZERO B
C++ X
C+=C Y, CURRENT_Y
I++ B
; The following should probably deallocate storage in the assembler so
; that entire large programs can be assembled in one pass.
ENDPROC MOVE$CURSOR
ENDMODULE LOCAL$FILE
```

Which in the end would probably generate something like the following:

```plaintext
; Example of Link-generated allocation of storage.
MOVE$CURSOR.X EQU $10
MOVE$CURSOR.Y EQU $11
MOVE$CURSOR.B EQU $12 ; for low byte; also covers $13.
LOCAL$FILE.CURRENT_X EQU $900
LOCAL$FILE.CURRENT_Y EQU $901

; Example of code.

MOVE$CURSOR
LDA #0        ; zero
STA $12
STA $13
INC $10       ; ++ for character.
CLC            ; += for character.
LDA $11
ADC $901
STA $901
```
LDA $12       ; B += 0x1fff
ADC #$ff
STA $12
LDA $13
ADC #$1f
STA $13

There's a lot of allocation operations in the intermediary code, but that makes sense.. the key to using
the 6502 is to allocate scarce zero page resources, and the key to making a faster assembler is to
allocate & dealllocate procedure space so all procedures can be assembled at one time. There needs
to be a keyword that hints "put this in the zero page." The C "register" keyword isn't quite right, since
it's perfectly reasonable to take the address of such a variable. Perhaps the right keyword is "zpage".

Probably recursive calls should be marked as "recurse" to note that they're expensive and that's
okay, though maybe not - the compiler in particular might need to recurse. Ideally, the compiler/linker
combo could figure this all out automatically (this is quite practical, since the linker has to determine
the call tree any for allocation purposes).

Procedures-in-procedure support could be added later. That's easy as long as they don't recurse -
simply reference the address of the object.

Various other words as compiler hints (like "register" in C) would be helpful, esp. for the loops (so
that the X or Y register can be used as a loop register where appropriate). Such keywords probably
include ZPAGE, REG(A,X,Y). Perhaps FOR REGX I := 10 DOWNTO 0 {by 2} DO ... ENDFOR; could
become

LDX #$0a
; if constant start, don't need to check here!
.l
...  
DEX
BPL .l

Probably it should support modules (like C, based on files), header files (maybe add a C-like macro
expander?), etc.

Note one odd thing: in a big program, it might be efficient if the tightest loops are in separate
subroutines called by others; that way, since they're lower in the tree, they're more likely to get zero
pages assigned to them.

CC65, a freeware C compiler for 6502s, has a "-static-locals" option. This option lets locals be in
static locations, and they correctly note that this produces faster code but doesn't permit it to be re-
entrant. However, it doesn't appear that CC65 takes advantage of allowing function parameters to
also be passed statically, nor does it exploit the approach discussed here to allocate so that copying
(e.g., to parameters) is only notional and doesn't actually occur in the code itself. CC65 appears to
be open source software, and is available at: http://www.cc65.org CC65 normally takes a different
approach (based on the older SmallC); it uses the A and X registers as a register, and pushes
parameters on a small stack through multiple JSR's (that are simple to implement as a compiler, but
not really very efficient).

Then a small text editor could be built, both to use these things (eventually the Simple compiler,
linker, and ASM+ assembler should be self-hosted) and to show how to use these things.

It would probably have a buffer module (for storing the text) and a display module (to take what's
in the buffer & display it). If text has been modified, move it to a line buffer internally so that inserts
& deletes don't move everything. Allow cursor motions keys like wordstar (control ESDX), delete previous letter, move around like emacs, find (control-F?), and ESC back to a menu for saving, loading, cataloging, help. Two option sets: one for normal text vs. one for programs, and a general host-based one (shift key mod, etc).

The key here is to have a basic, easy-to-use text editor; one didn't come with the Apples, oddly enough.

**SOLUTION 2: FIXED LOCATION DATA STACK**

This approach depends on fixing the location of the data stack, but spreads the lower and upper bytes to different locations (possibly different pages) instead of having the bytes be adjacent in the memory space. It would work nicely for Forth, C, etc. The trick is that the upper & lower bytes of multibyte values aren't stored contiguously in memory, but are instead "spread" into parallel pages; this trick increases the available space significantly, makes stack movement code faster, and increases safety. IE, you have "all data stack low bytes" contiguous, and "all data stack high bytes" contiguous if you allow 2-bytes of data on your stack.

Though it's the normal 6502 convention, there's no REQUIREMENT to store 16-bit values in the order of "lower 8 bits", followed immediately by "upper 8 bits". The upper & lower bytes could be separated by a standard distance (e.g., 256 bytes) instead.

If you create fixed positions for the array of "lower bytes" and the "upper bytes", as long as there aren't more than 256 elements, access is easy. Simply set the X or Y register to the index, then use the "address,X" or "address,Y" access mode to get or store the lower & upper byte of whatever you want. Of course, the address+X shouldn't cross pages, since this would slow down access on an already slow machine.

Thus, the X or Y register becomes the stack pointer, into a stack that might use 2 pages.

For data stacks (necessary for Forth, C, etc), this works out particularly well. This means that if you're willing to live with a 256-element data stack (not completely unreasonable), you can have a pretty clean & quick interface. Moving up & down the stack involves a single increment or decrement to the X or Y register, the cheapest possible operation. You can also cheaply access operations "inside" the stack by selecting precomputed offsets, again making access MUCH easier.

I'd earlier toyed with a 128-element data stack, but this "spreading" approach is easier, and allows twice as many data elements! Pushing and popping are cheaper (one inc/dec instead of two, without lots of carry checks). This also means that char-vs-int errors aren't deadly (as they are if chars push 1 byte and ints push 2). Just use the X or Y register as the data stack pointer, and have two pages set aside for data (one for low, one for high). Even better, you could have 4 pages, and long ints can be passed around without worrying about incorrectly taking them off the stack as the wrong size! Type conversions between char, int, and long become no-ops, eliminating a set of dangerous mistakes.

If you want the approach but can't live with only 256 data elements, you could check on each stack push & pop, and move (most of the) data elements when the stack got full or empty. This is slower, obviously.

Conversely, if you're willing to accept a far smaller data stack, the results would be ESPECIALLY efficient if you slap the stack into the zero page. If you normally have to deal with 16-bit integers, this won't waste any more space, but it saves all the incrementing/decrementing you have to do to fix up stacks, AND if there's an error in stack manipulations (a serious problem in Forth) you'll have far less damage.
You could even combine techniques: use the zero page for the data stack, with a standard offset. If the stack overflows or underflows you copy to/from elsewhere. Having a zero-page stack, accessed this way, with overflows might give really good performance for a data stack-based system.

One problem is that the stack can't handle many large data elements (no local variables with 200 bytes each); I've programmed on such systems before, and as long as there's a heap allocation system that's not a serious problem. The implementation could even put variables beyond a certain size on the heap automatically.

One design decision: do you keep an X or Y register set nearly all the time as the stack pointer? Almost certainly yes, since this would be heavily used. If so, do you use the X or Y register, and which direction do you grow (up or down)? I'm not certain if the X or Y register is best, nor if growing up or down is best. At first I used X, then someone suggested I use Y, but after examination I think X would be better. There are some accesses using indirect pointers that are useful and require the Y register. Even more importantly, many instructions such as the incrementing and decrementing instructions can only be used directly if X is used as the stack pointer; with Y it's more complicated.

Here's a sample. For the stack, the X-reg points to the top of stack, growing downwards in memory. Thus X starts large & get smaller as more items get put onto it. If it becomes 0, the stack is full. On decrementing X, you can check to see if it's become negative, or to be more paranoid you could declare an error if it's zero (X=0). Set X=$FE (say) as an "empty" value. To look "inside" the stack, simply use addresses with assembly-time increments (i.e. no runtime cost!).

```
; perhaps alias "LOWSTACK+1" as "SECOND$PARAMETER$LOWBYTE",
;               "LOWSTACK+2" as "THIRD$PARAMETER$LOWBYTE", etc.
; LOWSTACK and HIGHSTACK could be in the zero page, giving less data space
; but more speed and smaller code -- or they could be at the beginning
; of two different pages, giving 256 possible data values.
; Note that the assembly language given here can assemble either way,
; so that could be an easy configuraion option.

ADD-INTEGERS
; pop & add top two integers on the stack.
; pushing result onto stack.
CLC
LDA LOWSTACK,X        ; add low bytes.
ADC LOWSTACK+1,X      ; this is how to look at the "one inside".
STA LOWSTACK+1,X      ; replace 2nd parameter (which will be the return value)
LDA HIGHSTACK,X       ; add high bytes.
ADC HIGHSTACK+1,X
STA HIGHSTACK+1,X
INX                   ; pop the stack (once), leaving result on stack.
RTS

DUP-INTEGER
DEX                   ; push the stack, making room for the new value.
; Always make room for a value first by pushing first;
; that way, the system could support
; interrupts which use this stack too.
LDA LOWSTACK+1,X
STA LOWSTACK,X
LDA HIGHSTACK+1,X
STA HIGHSTACK,X
RTS
```

If the stack were always pushed before data was stored on it, interrupt handling could be well-supported. This would make it easily possible to create time-sharing on a 6502 (a frightening thought!).
Obviously, stack-checking code could be inserted too, at some cost in code size and performance.

In comparison, the FIG Forth implementation available at: http://www.6502.org/crossdev/lang/index.htm does use the zero page, but it has to do a little more work to manipulate the stack (more fiddling with the X pointer to move data around). Even more importantly, if the programmer makes a mistake, it's curtains for the data. Doing this can encourage using single-byte data (something the 6502 is FAR more efficient at), since it no longer is as dangerous to use in a Forth system. The FIG Forth implementation is a word-based interpretive system, which is slower than a JSR-based implementation (where the Forth words are compiled into JSR WORD everywhere)... I'm partial to the JSR design due to its speed, but the approach described in this paper will work with either.

Implementing a C compiler this way would produce less efficient code that a "Simple" compiler would, in general, since reading or writing any value requires an indirect use of a register (,X or ,Y) instead of a direct write to memory, and would mainly use non-zero-page (instead of zero page). Thus, it'd take more bytes of code, and it'd be slower to execute. However, recursive code would be much easier to handle - Simple would have to work hard to do it. Also, implementing this scheme is _extremely_ easy, while doing "ASM+" right would take a LOT more work.

If I were building a "language shop", this would be a good way to add a Forth & a C compiler to the "language collection." Programming languages which are often implemented using a stack-based language would fit this model easily too (Java, C#, Python, Perl, PHP, Pascal). The next trick would be to figure out how to make sure each can call the others. Probably not too bad via some special "external language call" interface, at least no worse than other approaches. Forth in particular would work nicely in this scheme.

The two different solutions need not be completely independent. For Simple, using the basic idea implies that integer arrays (up to 256 elements) could be stored as an array of low bytes, then an array of high bytes, making access easy.

In fact, it seems like 2 macro languages could be developed - the ASM+ one above, based on fixed addresses, and a stack-based macro language, and lots of reuse could occur. The underlying assembler & peephole optimization system could be the same, and the same 2nd-stage peephole optimizations might be used for both (though some optimizations would only be useful in one or the other).

Then you could allow something of the form:

```
PROC HELLO ; proc hello(in int c, out char d)
TEMPS 2 ; use this so can set up space for temporaries.
S-IN-I C ; make sure these are in order.
S-OUTI D ; the macros remember order to make declaration easy
TEMP-I E ; int e;
TEMP-I F ; int f;
SZEROI F ; f := 0;
SI+ E, C, F ; E := C + F;
SI:= D, E ; D := E;
SI++ E; ; I++;
ENDPROC ; pop off temporaries, deallocate in assembler.
```

Would generate:

```
HELLO
DEX ; space for E
; stack overflow check if desired.
DEX ; space for F
; stack overflow check if desired.
; Now stack looks like: (top/low mem) F E D C (bottom/higher mem)
LDA #0 ; F := 0
```
STA LSTACK, X
STA HSTACK, X

CLC ; E := C + F
LDA LSTACK+3, X
ADC LSTACK, X
STA LSTACK+1, X
LDA HSTACK+3, X
ADC HSTACK, X
STA HSTACK+1, X
LDA LSTACK+1, X ; D := E
STA LSTACK+2, X
LDA HSTACK+1, X
STA HSTACK+2, X

; Now need to implement E++. Since
; INC LSTACK+1, Y is not a legal addressing mode,
; this is a good argument for using X and not Y.
LDA LSTACK+1, X ; E++
CLC
ADC #1
STA LSTACK+1, X
LDA HSTACK+1, X
ADC #0
STA HSTACK+1, X

1
INX
INX ; pop temporaries. Should params be popped by caller?
RTS

A few notes regarding incrementing - Geoff Weiss noted that if you find that you rarely have to
increment the high byte, and want to get better performace from incrementing 8-bit values and add a
penalty to the 16-bit operation, use this code:

LDA LSTACK+1, X
CLC
ADC #1
STA LSTACK+1, X
BCC .1
LDA HSTACK+1, X
ADC #0
STA HSTACK+1, X

1

Again, note that implementing "E++" can't be done with the following code, since it doesn't use a
legal addressing mode:

INC LSTACK+1, Y ; E++. This addressing mode is not legal.
; Should use X instead of Y.
BNE .1
INC HSTACK+1, Y

COMPARING THE TWO APPROACHES#

Approach #1 takes more work to implement, has the linker do more work, and doesn't handle
recursion as easily, but it has lots of performance advantages. Approach #2 is probably much easier
to implement, but it has its weaknesses.
Let's examine adding two numbers (ignoring any return) to compare them. Here's approach #1, adding A and B and placing them in C (to keep things consistent, I'll only use normal adding and not the adding that speeds low-only-byte adding while making higher-byte adding more expensive and adding code):

```
ADD-INTEGERS
CLC
LDA A$LOW
ADC B$LOW
STA C$LOW
LDA A$HIGH
ADC B$HIGH
STA C$HIGH
```

Here's approach #2:

```
ADD-INTEGERS
; pop & add top two integers on the stack.
; pushing result onto stack.
CLC
LDA LOWSTACK,X       ; add low bytes.
ADC LOWSTACK+1,X     ; this is how to look at the "one inside".
STA LOWSTACK+1,X     ; replace 2nd parameter (which will be the return value)
LDA HIGHSTACK,X      ; add high bytes.
ADC HIGHSTACK+1,X    
STA HIGHSTACK+1,X    
INX                  ; pop the stack (once), leaving result on stack.
```

Pulling up the 6502 opcodes (e.g., at http://www.6502.org/tutorials/6502opcodes.htm), we can compare the approaches.

Approach #1 - all zero page: 13 bytes of code, 19 cycles to run no zero page: 19 bytes of code, 25 cycles to run Approach #2 - all zero page: 14 bytes of code, 26 cycles to run no zero page: 20 bytes of code, 28 cycles to run

Approach #2 is only one byte longer in code because it has to manipulate the stack at the end. Approach #1 might have to do copying, which would take more effort and make it much slower than #2, but if the linkage overlap idea works well that copying is rare. Approach #2 is always slower than #1; even when approach #1 can't use a zero page, it's faster than approach #2 when it's using the zero page. Is that a problem? Well, it depends.

Although approach #2 has its drawbacks by this measure, it’s much easier to implement, and there’s something to be said for easy.. no fancy zero-page allocation is required for this stack-based approach, so a macro assembler would be able to implement it (without peepholes, though peepholes would improve the results). In particular, approach #2 would be simpler for self-hosting.

This solution #2 is also more traditional, so link times would be much smaller.

**Notes on creating other languages:**

A finalization system would be useful, since a number of allocation requests could be to compensate for the inability of the system to have large local variables. Heck, an automatic garbage collector might be fine - mark & sweep collectors wouldn't have much to do, since there's not much memory to sweep.

Initialization and adjustment (like Ada 95) would be useful in eliminating common errors.
Inheritance can be implemented as with C++ or Ada 95. Dispatching would be little expensive, though; each indirection would require copying an address into the zero page and using it.

A completely controlled, unbounded-length string type would be very handy; it would make text manipulation a lot easier.

Obviously, for real efficiency this would need to be combined with many other techniques. For example, using registers is even faster, so using A, X, and/or Y to pass some data is even faster in most cases (no saving out and writing back). However, there are only 3 8-bit registers, and they are needed for other things too, so while using them in part for parameter passing is a good idea, it's limited in its utility. PHA and PLA and the zero page can be used to briefly store and retrieve registers, and there are lots of clever techniques for performance aids too.

It's critical to use 8-bit (char) values when you can, e.g., for booleans, because the 6502 is FAR more efficient at them. Even on the rare cases where Approach #1 has to do a copy, it's far less. Approach #2 even encourages this, because an error in typing is less catastrophic.

REFERENCES: