David A. Wheeler's 6502 Language Implementation Approaches

 dwheeler.com/6502/

This page has some information on implementing computer languages (beyond ordinary assembly language) on the extremely old and obsolete 6502 chips. I still find it to be an interesting intellectual challenge, even though it has no *commercial* use that I know of. Many programs for the 6502 were written in assembly language, because it was difficult to efficiently implement high-level languages on the 6502. People certainly use higher-level languages on 6502s, but I often wondered if there were better ways to handle them. I focus slightly on Apple II-based systems, but not much; for development, a 6502 is a 6502.

There's lots of general information on the 6502; here's a summary of the 6502 instruction set. The Virtual 6502 will let you do some quick playing with their assembly language via a browser. Warning: There's lots of variation in assembler syntax; this one that requires zero page accesses be specially marked with "*".

Ideally, a language could be self-hosting and interactive (allowing redefinitions on the fly), but also cross-compiling (emphasizing "best possible performance" and a nicer environment), and would let you do speed vs. space trade-offs. For self-hosting, a small size has value (so that there's room for other stuff!).

Note: I strongly prefer open source software for development tools, especially in this case since there's no real possibility of long-term for-pay support, so I'll specifically warn away from problem licenses (or non-licenses) if I know about them.

Right now I find Forth, PLASMA, and Atalan especially interesting in this area, as described below, but there are many other interesting approaches too.

My specialized approaches

Here is a discussion on approaches to implementing languages on 6502 chips, which discusses several fundamentally-different approaches to implementing languages on the 6502. The 6502 is not easy to generate good code for, and I think these approaches produce better results than the "obvious" approaches used by most.

Here is some demo code for the first approach, creating overlapping memory locations for use as local variables and for parameter passing (to and fro). (GPL). Using overlapping addresses works around the limited memory, inefficient pointer/stack manipulation routines, and 8-bit registers of the 6502.

Below are various other options. Forth, Action!, and Slang have especially interesting implementation approaches (for different reasons) if someone wants to create/improve new implementation approaches. c65cm (formerly 65CM) is also very interesting, especially for self-hosting; they greatly limit the language, but as a result make it trivial to compile.

Forth

Forth is a remarkably useful language if you intend to self-host useful and fast programs on an 8-bit system; it's relatively fast in execution, fast to develop in, powerful, and easily fits in tiny spaces (say, 8K for the Forth system and at least 1K of RAM to create new definitions). You have to get your head around reverse Polish, and be prepared for disaster if you don't put the right number of parameters on the data stack. Forth normally doesn't keep track of how many data values its words consume or produce, so programmers have to keep track (!). (If you don't use locals, it's even trickier; Gforth's locals and ANS Forth locals can make it easier to program in Forth, and there's a a short implementation of {...} local variables that makes Forth programming easier, but it creates inefficiencies if the compiler doesn't do any optimizations.) In my mind, Forth forces humans to switch to a less-common syntax

(reverse polish), making the human be the "parser", and almost no error-checking; in return, you get a simple, powerful system (Forth's ability to return multiple values from a single function is more capable than C's).

Starting Forth is the classic text for learning Forth; in fact, it's just a great book that shows how to make complicated topics easy to understand. *Thinking Forth* is good as well; it's basically the follow-on book.

If you want to get an idea of how Forth works, try out Forth in Javascript. From the command line you can do stuff like (a "\" precedes a line of comments):

```
\ Put 3 and 4 on the stack. The "+" function adds the top two numbers
\ on the stack (consuming them) and puts the result on the stack.
\ The "." function consumes and prints the top number on the stack:
3 4 + .

\ Create a new function named DEMO that prints "A".
\ The ":" starts the definition (followed by the name), "65 EMIT"
\ puts a 65 on the stack and then prints that character, and
\ the ";" ends the definition.
: DEMO 65 EMIT ;

\ Run the new function:
DEMO

\ Create a variable called "A":
VARIABLE A
\ Now any use of "A" puts its address on the stack.
\ Store "5" into variable A.  The function "!" takes a value from the stack
\ and stores it in the address at the top of the stack:
5 A !
\ Retrieve ("@") and print (.) the value stored in A:
A @ .
```

The original "fig-FORTH 6502" was released in in September 1980 under very generous legal terms, with complete source code and the ability to modify and release derivatives of it. It took me forever to track down its legal status; back then, many people didn't concern themselves about legal statements, which makes things painful to deal with in today's lawsuit-happy times. Basically, it merely states that you just have to give them credit.

If you use a fig-Forth derivative, just include the notice in further distribution from its assembly code listing, which states "RELEASE 1.1 WITH COMPILER SECURITY AND VARIABLE LENGTH NAMES ASSEMBLY SOURCE LISTING". The assembly listing states that "This public domain publication is provided through the courtesy of Forth Interest Group, P.O. Box 1105, San Carlos, CA 94070. Further distribution must include this notice. FORTH INTEREST GROUP * * * * * PO. Box 1105 * * * * * San Carlos, Ca. 94070". In general, FIG released code to the public domain, as you can see by their covers, and since they are the ones who made the official releases, I think it's reasonable to rely on the front covers that say that they are "released to the public domain" or at most merely require acknowledgement, in which case it's legal to use, modify, and re-release derivatives.

Here's the FIG-Forth assembly code as PDF including its legal notice. Here's my local copy of the FIG-Forth PDF page to verify this, in case that goes away. Here's a zipped version of FIG-FORTH for the (generic) 6502, and here's the FIG6502.ASM assembly code for FIG-FORTH for the (generic) 6502 as text via wrodiger's copy of FIG-FORTH for the 6502. The fig-forth glosssary describes every word and what it does, and is a gold mine for a simple explanation of how to use it. Here's the set of related PDF files, including the Apple II version (a specialized version of the 6502 version). The fig site only has the source code as PDF, but Dietmar Krueger (dkrueger, at retronym dot

de) has managed to recreate the Apple II version as text: MyFigForthApple2.zip includes MyFigForthApple2.s.txt.

Why Forth isn't slow discusses why Forth is remarkably fast; he determined that Forth had about a 29% overhead compared to similar assembly, which is remarkably small (there are BIG caveats with that calculation, but the point is that the overheads are relatively small). A direct threading implementation would probably be better than fig-Forth's indirect threading, but I digress (see Rodriguez' article for why). ProForth for the Apple II is derived from fig-Forth. The original fig-Forth for 6502 has a few known bugs; its UM/MOD has a bug that Garth Wilson found and corrected (here's the basics). Its UM* also has a bug, see here for a fix and optimizations (see also this 6502 software math page).

It might be easy to switch FIG-FORTH over to a split stack that stores low-byte/high-byte on the zero page as separate ranges (my approach #2). This would create lots of efficiency improvements: pushing and popping are shorter operations (1 operation instead of two, every time). Also, although this would make it slightly nonstandard for Forth, logical calculations and checks could use a 1-byte value and ignore the "high" byte, speeding/simplifying things further. You could have many other 8-bit operations too, which would speed things up. ANS Forth specifies a minimum of 32 cells of Parameter Stack {64 bytes, easily fitting in zpage} and 24 cells of Return Stack {usually this is the page 1 return stack} (Brad Rodriguez prefers 64 cells of each). (By having the TOS be 2 bytes in zpage, you can still use the TOS easily as a pointer.) (Sadly, pointer accesses are harder this way.)

Forth Interest Group (FIG) and Silicon Valley FIG has more info. Here's a partial list of Forth implementations. Papers about efficiently implementing Forth include "MOVING FORTH: Part 1: Design Decisions in the Forth Kernel" by Brad Rodriguez (a great paper explaining how to implement Forth), Threaded Code Variations and Optimizations by M. Anton Ertl, Threaded Code, "Evolution of Forth"'s discussion of future directions, "Threaded interpretive systems and functional programming environments" by Harvey Glass, ACM SIGPLAN Notices, Volume 20, Issue 4 (April 1985), pp. 24-32 (ISSN:0362-1340) and of course Wikipedia's article on threaded code. Forth meta compilation is not only interesting - it helps you understand Forth innards. Forth's advantages don't do much for systems with a Gig of RAM, and its disadvantages (difficult-to-read, little error-checking, etc.) have made it all but disappear from modern systems, but there's much positive to say about it for the 8-bit world. Stack machine briefly introduces the theory behind stack machines. Updating the Forth Virtual Machine (Pelc) has a discussion of interesting additions to the traditional Forth VM as is often implemented today.

Ron's Software Page has interesting things, including QForth. Forth.org has a set of Apple // Forth implementations.

SPL is a Forth-like language that is implemented in plain Python and generates 6502 assembly code.

I should note that there are many versions of Forth. The original fig-Forth is heavily related to the later Forth-79, and many Forths are based on fig-Forth. The later Forth-83 made a lot of big (controversial) changes: "true" changed from 1 to -1, the meaning of NOT changed, it mandated floored division (incompatible with prior usage), and some words like PICK and ROLL became 0-based. Here's guidance on converting fig-Forth to Forth-83. The later ANS Forth tried to make things less rough.

The big problem with Forth is that it's entirely postfix, which many people find hard to read. What can be done about that? Interestingly, several people have developed infix versions of Forth, which might produce the best of all worlds... the readability of more traditional notations, and the speed + small size of Forth. "An infix syntax for Forth (and its compiler)" by Andrew Haley (Red Hat) (2008) describes a nifty reader; "Infix Forth is not a translator from some other language to Forth, but an infix form of the language that doesnâ€™t change its semantics."

Many Forth systems can easily bring in an assembler. William F. Ragdale wrote a Forth assembler for the 6502 that takes about 1300 bytes.

I think that a programming language could be devised that combined the efficiency/simplicity of Forth implementations with more traditional syntax.

Forth on 6502 has more information.

PLASMA

PLASMA (Proto Language ASsembler for Apple) by David Schmenk is "a combination of virtual machine and assembler/compiler matched closely to the 6502 architecture". PLASMA source code is available on SourceForge. This was based on the author's experience in developing a Java VM for Apple //s (!) - he created a new VM, and a language for implementing it. It unambiguously open source software (GPLv2), which is great. It is currently focused on ProDOS, but I expect it would be easy to port to other operating systems (such as Diversi-DOS).

PLASMA is a new language for an old system, but it looks extremely promising. It has a much more traditional syntax, yet it is designed to work well within the very limiting 6502 structure. An interesting aspect is that you can easily state, for each function, one of three implementation techniques: interpreted bytecode ("def", smallest but slowest), threaded calls into the interpreter ("deft", in-between), and natively compiled code ("defn", fastest but largest). Thus, the programmer can declare for each function if it should optimize for speed, space, or between, which is promising. Unlike some languages (like Action!) it can handle functions that call themselves (limited by stack space, but important for things like self-hosted compilers).

It is designed to be self-hosting, which I find especially interesting.

The main high-level PLASMA source language is more capable than Integer Basic (e.g., it supports real functions and procedures), but it does not have support for many different types and only basic support for structures. It does not include built-in floating point support, for example. It looks like a promising language for self-hosted development of games and system applications.

The PLASMA source language uses ";" for comments that end at end-of-line. End-of-line ends an expression. Constants can be defined with "const NAME = VALUE", variables with "DATATYPE NAME = VALUE" (DATATYPE can be "word" or "byte"), and the basic "if" syntax is if ... then .. fin. Here are some examples, based on snippets from "pong.pla":

```
CONST FALSE     = 0
CONST TRUE      = NOT FALSE
CONST KEYBOARD  = $C000
CONST KEYSTROBE = $C010
CONST SPEAKER   = $C030
BYTE  EXITMSG   = "PRESS ANY KEY TO EXIT."
BYTE  SCORE[3]

DEF BEEP(TONE, DURATION)
  BYTE I, J

  FOR J = DURATION DOWNTO 0
    FOR I = TONE DOWNTO 0
    NEXT
    ^SPEAKER
  NEXT
END

DEF KEYPRESSED
  RETURN ^KEYBOARD > 127
```

```
END

DEF GETKEY
  BYTE KEY

  REPEAT
    KEY = ^KEYBOARD
  UNTIL KEY > 127
  ^KEYSTROBE
  RETURN KEY
END

DEF TEXTMODE
  RETURN ROMCALL(0, 0, 0, 0, $FB39)
END

DEF GOTOXY(X, Y)
  ^($24) = X
  RETURN ROMCALL(Y, 0, 0, 0, $FB5B)
END
```

The PLASMA run-time is a stack-based virtual machine (with stack on zpage). Since it's stack-based, it includes operations like ZERO (push zero on the stack), ADD (add top two values, leave result on top), and INCR (increment top of stack). Some instructions look like Forth, e.g., DUP (duplicate top stack value) and DROP (drop top stack value). Other instructions are clearly designed for a traditional programming language, and not Forth: CALL (subroutine call with stack parameters), ENTER (allocate frame size and copy stack parameters to local frame), LEAVE (deallocate frame and return from subroutine call) and RET (return from sub routine call). Its basic runtime implementation is pretty straightforward. The bytecode uses only even numbers (so the implementation doesn't have to multiply each instruction by two to use it as an index).

Here are the PLASMA VM opcodes (TOS=top of stack, with 16-bit values):

- ZERO - push 0 on the stack.
- ADD,SUB,MUL,DIV,MOD - compute TOS-1 op TOS, where op is add, subtract, multiply, divide, or modulo, popping both and pushing result. These are 16-bit values. E.G., "SUB" subtracts TOS from TOS-1. The implementation doesn't really pop and re-push, of course.
- INCR,DECR,NEG,COMP - Increment/decrement/negate/bitwise-complement TOS.
- BAND,IOR,XOR - Compute TOS-1 op TOS, where op is bitwise-and/inclusive-or/exclusive-or
- SHL,SHR - Shift TOS-1 left/right by TOS
- IDXW - Shift TOS left BY 1, then add TO TOS-1, replacing both with result. Presumably this is to compute index addresses given an index in TOS and base in TOS-1. (The documentation says it shifts TOS-1, but I think that's wrong).
- NOT - Logical not
- LOR,LAND - Logical or/and
- LAA - Replace absolute tag with address
- LLA - Load address of local frame offset

- CB - Load a constant byte (from the instructions to the stack)
- CW - Load a constant word (from instructions to the stack)
- IDXB - Code same as ADD. Presumably to index bytes, where index is TOS and address is TOS-1.
- DROP,DUP - Drop/duplicate TOS.
- OVER - OVER TOS-1 to TOS.
- SWAP - Swap TOS with TOS-1.
- BRLT,BRGT,BREQ,BRNE - Branch on less-than/greater-than/equal/not-equal
- ISEQ,ISNE,ISGT,ISLT,ISGE,ISLE - Compute is-equal, not-equal, greater-than, less-than, greater-equal, less-equal for TOS and TOS-1. Consumes both, and pushes to TOS either 0 (false) or $FFFF (true).
- BRFLS, BRTRU - Branches.
- JUMP,IJMP - Jump, indirect jump.
- CALL,ICAL,ENTER,LEAVE,RET,EXIT - Routines for handling subroutines.
- LB,LW - load byte/word given address TOS
- LLB,LLW - load byte/word value from local frame offset
- LAB,LAW - load byte/word from absolute address
- DLB,DLW - Store byte/word value to local frame offset without popping stack
- SB,SW - Store byte/word value TOS to address TOS-1
- SLB,SLW - Store byte/word value to local frame offset
- SAB,SAW - Store byte/word value to absolute address
- DAB,DAW - Store byte/word value to absolute address without popping stack

At this early stage the "native" version is pretty inefficient (it appears that the current early implementation doesn't have an intermediate format, leading to lots of inefficiencies). But the bytecode and threaded (subroutine) call approaches look pretty efficient for what they are; I looked over its basic wordset and it looks like a really good fit for the 6502. The threaded call approach in particular looks fairly efficient (for that kind of approach). This approach may make larger programs easier, since many programs have a few routines that are speed-critical and many others that are not.

The (initial) PLASMA summary page includes a link to a Plasma Apple][boot image you can just boot and use . I have had no luck getting it to work on cAndy Apple (an Android Apple][emulator), but on Android it works just fine with the KEGS KEGS IIgs Emulator by James Sanford. David Schmenk, the creator of PLASMA, reports that it also works on Virtual][. Once you start the boot disk, you will be dumped into the text editor. In this text editor you can edit text or press ESC to enter special commands. A simple way to see the instructions is to press ESC, then type the line "r edit.readme" (do not include the quotes, note the space after r, and end the line with RETURN). The text editor actually supports several commands when you press ESC, including "r FILENAME" (read in FILENAME and throw away the current buffer), "c" (show the disk catalog), and "x" (compile and execute). For a demo, you can do ESC r SPACE rod.pla RETURN, then x RETURN; that runs a short demo called "rod.pla" that shows simple color graphics.

There have been many refinements since then. The PLASMA github site has a significantly updated version, and adds some very interesting features such as multi-assignment. It is being used as the VM for a new new adventure game being developed for the Apple II and Commodore 64 called Lawless legends. However, one thing missing from the updated version is an IDE that runs on the Apple II; That is his current work-in-progress. He hopes to eventually write a JIT compiler, using the CPU type to compile optimized machine code from byte code as it loads.

Atalan

Atalan is a new language for 6502s and other 8-bit systems. It is open source software (released under the MIT license) and is also available via the Google code page for Atalan. This compiles straight to 6502 code, and they claim they do a good job. It's got a lot of language features; it looks like a very pleasant language to use.

Their documentation says that, "Procedure local variables and arguments are internally allocated as global variables. Atalan analyzes procedure call chain and if possible, reuses global variable space to several procedures." That was an approach I wrote about years ago!

Atalan *only* compiles to machine code; given the limited memory space, that could cause serious challenges for larger programs. It is designed to be cross-compiled; it is not clear if it could realistically support self-hosting.

Python

PyMite implements a Python subset for 8-bit systems; I can easily imagine this working for a 6502.

C and C-like

CC65 (6502 C compiler) exists, it's a descendant from Small-C. It's not open source software (it predates general awareness of licensing issues), unfortunately. It implements a fairly wide range of C, but the code it generates is inefficient. The "Internals doc for CC65" explains how it works (and why it's inefficient). In CC65, "The program stack used by programs compiled with CC65 is located in high memory. The stack starts there and grows down. Arguments to functions, local data etc are allocated on this stack, and deallocated when functions exit... the return address goes on the normal 6502 stack [page 1], *not* on the parameter stack... the AX register pair [is] the primary register. Just about everything interesting that the library code does is done by somehow getting a value into AX, and then calling some routine or other..." It then shows a calling sequence for i = baz(i, c); where i and c are global variables; it doesn't show what you to do work with the parameters, but it's nontrivial.

HyperC (for ProDos) is interesting; they added a "var" prefix for variables like Pascal (though if implemented using pointers, this could be quickly inefficient since pointer manipulation is painful on 6502s.. but if implemented using my approach #1 that'd be nice). They also added a "@address" statement, so you could allocate exactly where a variable would go. HyperC supports K&R C, not ANSI C, and omits stuff like scanf().

I briefly used Manx's Aztec C; it was awful. It generated bad code, and was a pain to use.

It's definitely costly and is not OSS, but the Micro/C compilers (based on C-Flea (see also here) is a C compiler suite based on C-Flea, a small virtual machine for C.

Sort-of-assembler

65CM implements a "pretty assembler" language for the 6502 (GPL). It's designed so that the compiling is run on a PC, and the resulting code is sent to the 6502. 65CM handles 2 types of variables (BYTE and WORD), and supports inline assembler. It's not high-level at all, so be prepared for that. What it calls "expressions" are actually constant expressions (they can't have any variables), the IF statement must be of the form:

```
IF VE1 ( '=', '<>', '<', '>', '<=', '>=' ) VE2 THEN
  ...
ENDIF
```

where "VE1" and "VE2" must be a variable name or a (constant) expression. Similarly, you can make calls using:

```
GOSUB {Variable (pointermode) OR Expression (Address) OR Label (Internal)}
 [ PASSING {Variable} AS ACC {Variable} AS XREG {Variable} AS YREG ]
```

You can assign a variable a value:

```
LET {Variable} = {Variable OR Expression}
```

But if you want to add something to the variable, that's another statement:

```
ADD {VE1} TO {VE2} GIVING {VE3}
```

"The COMFY 6502 Compiler" by Henry G. Baker (Nov. 1997) describes the COMFY language (and the paper includes the source code!). It is "intended to be a replacement for assembly languages when programming on 'bare' machines", but intentionally has a LISP-based syntax.

> *"COMFY-65 is a `medium level' language for programming on the MOS Technology 6502 microcomputer [MOSTech76]. COMFY-65 is `higher level' than assembly language because 1) the language is structured- while-do, if-then-else, and other constructs are used instead of goto's; and 2) complete subroutine calling conventions are provided, including formal parameters. On the other hand, COMFY-65 is `lower level' than usual compiler languages because there is no attempt to shield the user from the primitive structure of the 6502 and its shortcomings. Since COMFY-65 ismeant to be a replacement for assembly language, it attempts to provide for the maximumflexibility; in particular, almost every sequence of instructions which can be generated by an assembler can also be generated by COMFY. This flexibility is due to the fact that COMFY provides all the non-branching operations of the 6502 as primitives."*

COMFY-65 is implemented in Emacs Lisp; it's not clear how hard it would be to create a self-hosting version. Any LISP might be aided by my readable LISP work. Here's a background paper on COMFY. (Henry Baker has many other interesting papers, too.) There's a git-able version of COMFY-65; unfortunately, it's not open source software (not even slightly!), due to ACM licensing nastiness. That's unfortunate.

Andy Hefner (ahefner) wrote a Common Lisp-based 6502 assembler (sort of a macro assembler with Common Lisp syntax). He posted the source code on github (MIT license). He blogged about it, and wrote a little demo for an NES; this was then picked up by Slashdot.

I should definitely mention Sweet 16, the "pseudo microprocessor" implemented by Steve Wozniak. (Here's the SWEET-16 Wikipedia article). It simulates a simple 16-bit microcomputer (handy when you need to do a lot of 16-bit operations) with 16 registers, running about 10 times slower than native code but taking far less space (each opcode takes 1 byte, and the whole interpreter takes about 300 bytes). Many Apple assemblers include support for Sweet-16. Unfortunately, the rights to use its original implementation aren't entirely clear. Apple, even in its heyday, encouraged it use in programs. Since it was was copyrighted by Apple in 1977, its copyright *should* have expired in 2005. That's because the U.S. copyright act of 1976 didn't take effect until January 1, 1978, and the previous copyright term in the U.S. was 28 years plus another 28 if renewed. I doubt Apple would have renewed its copyright, since this was not a money-maker by 2005, so it *should* have gone to the public domain. (This is also true for Applesoft Basic, which was released on cassettes in 1977). Unfortunately, a 1992 revision to the copyright laws automatically renewed all copyrights after 1963. I find this absurd; how are innovators supposed to innovate if essentially everything created is walled-off?

Lisp/LOGO

The "skimp" page also includes a list of several 8-bit LISP implementations, including several for the 6502. This discussion lists more. One useful implementation discussion (of many!) is R. Kent Dybvig's "Three Implementation Models for Scheme". University of North Carolina Computer Science Technical Report 87-011 [Ph.D. Dissertation], April 1987.

neslisp is a small subset of Lisp that generates 6502 code (NES); the as of version 0.1 it is released under the GPLv2 license (and thus is open source software). An older version is available via neslisp on Google code.

Picobit (GPLv3) looks interesting. You program in Lisp (specifically a variant of Scheme) on a larger system; it then compiles to a VM that runs on the target. Currently it does not support the 6502, but the system is designed to be portable and it looks to me that it'd be relatively easy to port to a 6502.

Lysp (MIT license) is another small implementation of Lisp. It doesn't directly support 6502 and is in C, but it's small... so it might not be hard to transliterate. A related program from the same developer is Maru, which can self-host. This could be a reasonable starting point for creating a 6502 Lisp.

Turning the Apple //e into a lisp machine, part 1 refers to this Lisp interpreter for the Apple //e (MIT license). They also show a clever bootstrap approach: they send code via the audio jack (!).

There's more that can be done; Lisp was originally developed in 1965, on even smaller systems. The book "Lisp in Small Pieces", Program Transformation in Lisp, this Scheme page and these details on implementation are all of interest. R. Kent Dybvig's "Three Implementation Models for Scheme" shows how to implement Scheme while still keeping most stuff on the stack. Tagging could be difficult, but an alternative (that I believe was done for early Lisp implementations) is to partition the memory into regions; to determine the tag, just check which memory region it is in. It might be possible to start with PicoLisp to create a 6502 version. There is a trivial ACL2 implementation of Lisp (GPL) that I converted into Common Lisp (compiler, run-time; See Wolfgang Goerigk's "Compiler Verification Revisited" for more).

I know that there was an "InterLISP 65" available for Atari's 6502-based line.

Various full LOGO implementations were available as well.

Basic and Pascal

Of course, many people used BASIC and Pascal (esp. UCSD Pascal). That's well-documented elsewhere.

UCSD Pascal is now available for non-commercial use, including its source code. (It is *not* open source software, as misleadingly stated.) UCSD had its own P-code system, which let a lot of code fit in little space. It wasn't fast, and the only "fast" escape was to write assembly. UCSD Pascal also required you to use its operating system, which was a problem for self-hosting systems that needed to interact with the rest of the world. UCSD Pascal was very useful, and many programs (including the first Wizardry!) were written in it. The fundamental problem with UCSD Pascal is that its opcodes weren't designed to be easily-supported by the 6502; instead, its goal was cross-system portability. As a result, any 6502 implementation of them was going to have trouble, in part because all stack-based operations on the 6502 were slow. This discussion about p-code and Sweet-16 notes this difference - Sweet-16 was designed to be easily-implemented by the 6502, and thus was a nicer fit to the 6502.

In theory it wouldn't be hard to at least create a better version of BASIC (e.g., add stuff from True Basic like real subroutines) to run on these things.

Java

Believe it or not, you can run Java on an Apple // (and other 6502 systems). VM02 implements a (limited) Java

Virtual Machine on Apple // (6502) systems. (This is the same person who wrote PLASMA). Java is not designed for these kinds of systems; the implementation can do some things, and it's cool that it works at all, but it's probably not the best choice for many tasks. It's pretty cool though.

Other Languages

Slang is an interesting 6502 language, currently for the Commodore 64. It is more like a "clean Basic" than anything else - full expression handling, structured "if" and loop constructs, and so on. It handles subroutines by separating the call from the argument-passing, which is a little unfortunate but understandable. Its approach to handling local variables is very straightforward; since they go in and out of scope, that is used to assign specific locations (a simple and clever approach). It's a big compiler, so probably can't often have the compiler and the running program resident simultaneously. Overall, this is a nice approach, *way* better than the typical Basics that came with 6502s (both in language cleanliness and in speed of result). Unfortunately, I can't find any licensing info.

Action! was a popular self-hosting programming language for Atari 6502-based systems. It was reasonably readable (its syntax similar to ALGOL 68). Wikipedia notes that "Action! is significant for its high performance, which allows games and graphics demos to be written in a high-level language without use of hand-written assembly language code. Action! language constructs were designed to map cleanly to 6502 hardware." "Local variables are assigned fixed addresses in memory, instead of being allocated on the stack. This enables tight code to be generated for the 6502, but precludes the use of recursion." {Note: Action could have added a "recurse" keyword, as I noted my text about implementation, which would have then supported recursion but only where it was needed, so that only functions that could eventually produce a call to themselves would have to pay the price. Effectus is a cross-compiler on Windows that targets Atari 8-bit, and is intentionally similar to Action!, but I don't see any source code available at all (it certainly doesn't appear to be open source software).

Note: Both Slang and Action! do not support recursive functions. In both cases, a few small extensions to support recursive functions (e.g., a "recurse" statement to mark them, so that calls must save the old arguments and local variables) would make them efficient yet easy to use, and permit self-hosting. (Even if a function is recursive, noting simple self tail-recursion would be easy, and would be worthwhile since saving variables is expensive.)

Suites

The Amsterdam Compiler Kit is a "cross-platform compiler and toolchain suite that is small, portable, extremely fast, and extremely flexible. It targets a number of low-end machines [and] supports several languages, including ANSI C, Pascal and Modula-2". It has (or at least had) support for the 6502. It dates back to the early 1980s, and was originally developed by Andrew Tanenbaum and Ceriel Jacobs as a commercial product and used as Minix' native toolchain. "After eventually failing as a commercial project, it was made open source under a BSD license in 2003 when it looked like it was going to be abandoned and the code lost... The ACK contains compilers for ANSI C, K&R C, Pascal, Modula-2, Occam 1, and a primitive Basic." Its intermediate language "EM" is a stack-based language; here is the technical description of EM.

Assembly

6502.org lists lots of assemblers, xa looks promising, and it supports common syntax. Ophis is a 6502 assembler written in Python in 2006-2007; it looks nice, though its syntax is a little different).

Other stuff

If you want to go elsewhere to see related pages, feel free to look at other locations such as www.6502.org (general site on the 6502). You can also find reference material on the web, such as the 6502 opcodes. Bob Sander-

Cederlof's newsletter *Apple Assembly Line* was filled with useful information. Here's a list of 6502 bugs (important if you're doing assembly!).

The Apple II Programmer's Catalog of Languages and Toolkits has a long list of development tools. Apple8 languages has lots of downloads for Apple II development languages. Here's another list of 6502 development tools.

It's not directly 6502 assembly, but High-Level Assembler (HLA) for 80x86 is interesting, and might have some useful lessons. The HLA manual explains the rationale for a "high level" assembly language.

You'll need to transfer files to and fro; this is tricky with Apple IIs because they have an unusual disk format; ADTPro and its cousin ADT can help; it lets you send/receive disk images via serial lines, Ethernet, or even audio (!). DOS33fs by Vince Weaver is a nice, simple utility for Unix/Linux that lets you read/write DOS 3.3 disk images. Apple 2 stuff has links to more disk image manipulators and an Integer Basic compiler. Syndicomm has Apple // boot disks available. NuLib can handle various "shrunk" formats. Apple][emulator resources guide lists many emulators. Apple][FAQs is what it says.

Here's a Javascript-only Apple][+ emulator, you can emulate one just with a browser!

Apple][ROMs

It'd be neat if someone wrote/released an Apple][ROM as open source software. I've talked with Woz, who's told me there's no chance of that from Apple.

Steve Nickolas has developed a simple Apple][ROM and released it under a permissive FLOSS license, specifically University of Illinois/NCSA Open Source License. He calls this the "gameware" ROM. You can download it from him; I have a cached copy. He says that "while crude, I did implement such a thing a few years ago... This only implements part of the monitor, requires a 'C02, is buggy as hell, and doesn't include BASIC, but it's a start."

I've sent an email to VTech, and VTech's Corporate Communications Department sent me an email on 2012-06-15 saying that "We have no problem to release the code but it will take some time for us to find it out, as it is a very old product. We will keep you posted on any update." That is a very exciting development, I hope it works out!! The Apple][, and VTech's competing implementation of it, is an important part of history. I think this is a great way to prevent the loss of this history, as well as making it possible to have have completely legal emulations of it.

It might be possible to get a clone maker to release ROMs, such as Video Technologies (VTech)'s ROM for the Laser 128, or Franklin's later ROM.

AAL-8603 notes the special values required for a ROM for ProDOS 1.1.1 to boot. "There are two problems with getting ProDOS to boot on a non-standard machine. The first is the ROM Checksummer. This subroutine starts at $267C in Version 1.1.1, and is only called from $25EE. The code is purposely weird, designed to look like it is NOT checking the ROMs... The wizards who put ProDOS together figured out a fancy function which changes the 64 bits from $FB09 through $FB10 into the value $75. Their function does this whether your ROMs are the original monitor ROM from 1977-78, the Autostart ROM, the original //e ROM, or any other standard Apple ROM... The original Apple II ROM has executable code at $FB09, and in hex it is this: B0 A2 20 4A FF 38 B0 9E. All other Apple monitor ROMs have an ASCII string at $FB09. The string is either "APPLE][" or "Apple][". Notice that the "AND #$DF" in the checksummer strips out the upper/lower case bit, making both ASCII strings the same."

6502 on Minecraft

There's a 6502 emulator in Minecraft that runs Forth, which is further discussed here.

Miscellaneous

25c3: The Ultimate Commodore 64 Talk discusses the Commodore 64 in detail, and includes a discussion on the 6502 (really 651) machine code.

Dedication

I've provided this information to world in memory of Bob Pew, an old friend who died in 1994. Bob, you're missed.

Feel free to See my home page. You may also want to see my Apple][page.