Multiple Stack Zero Operand Computers Today

Stack Computers

Stacks are simple, a child intuitively understands a stack of things and how it works. A stack is a data structure that is accessed from one end or it is a Last In First Out buffer. Stack Computers are machines that have hardware and instructions to facilitate the use of stacks in programs. In the hardware within a processor a stack may be implemented using pointers on the processor or in memory to locations in memory. It can also be implemented as on-chip registers or it may have part of the stack cached in on-chip registers and have the rest of the stack in memory.

Multiple Large Stack Zero Operand Computers

In most architectures one stack is directly supported by hardware and is used to nest subroutine return address, to pass parameters between routines, and for temporary storage of less important local variables within routines. Multiple stack architectures use separate stacks for nesting return addresses and for holding parameters passed between routines. ML0 is a designation for a Multiple stack machine with Large stacks and zero operand notation in the instruction set. [Koopman, 1989]

Machines that connect the ALU to a parameter stack are classed as zero operand architecture because the top of the parameter stack is the implicit operand of instructions. This is also called "pure" stack addressing. There are no parameters added to instructions to specify register selection parameters. Instruction paths are simpler in hardware this way. *Stack Computers*, published in 1989, listed twenty machines in this catagory. The author, Dr. Koopman, worked on the design of a number of those machines and goes into great detail on the reasoning involved in the decisions regarding architectural tradeoffs. There have been many exciting developments since the book was published.

Minimal Instruction Set Computers

I have been involved with the development of several new generations of state of the art stack computers of a type that we refer to as Minimal Instruction Set Computers. Our goal was not to reach the minimum number of instructions since the minimum is zero and that is not useful. We sought the minimal effective complexity for an architecture that is significantly different than Reduced Instruction Set Computers. [Moore, 1995]

In 1986 I purchased a Forth Kit from Mr. Charles Moore. It used his NOVIX NC4016 microprocessor. His native code optimizing compiler was only a couple of kilobytes and could compile movified versions of the complete system including itself in miliseconds. The NOVIX design used separate busses for the external memories that held the parameter stack, the return address stack, and main memory. The design was remarkably simple and incredibly fast for the time. NOVIX easily beat 80386 with only 4000 gates.

The zero operand architecture seems ideally suited to implementing both the smallest and fastest designs in any given process. The NOVIX chip was faster than Intel's fastest design at the time despite using one hundred times fewer transistors and with Intel using custom VLSI and NOVIX using a gate array! The ML0 design used neither microcode like CISC nor a set of addressable general purpose registers like common RISC designs.

Consider what a typical RISC architecture does to execute an instruction. First an instruction is loaded from memory. Then the instruction is decoded and the type of instruction is known but parameters complicate the process. The hardware paths associated with the location of parameters are decoded and enabled. At this point the instruction can then actually be executed. To get faster performance hardware designers build in multiple execution units in a pipeline so that the multiple steps required by instructions can be overlapped and the machine can execute more instructions per clock cycle. Managing these multi-stage pipelines involves a lot of hardware complexity which in turn slows down the actual speed of each overlapped unit.

In addition to the hardware and software expenses of multi-staged pipelining these designs require a great deal of instruction bandwidth from memory because Reduced Instruction Set Computers often have large parameter fields in the instructions. Because the processors tend to be much faster than memory some on-chip cached memory is required or multi-level cached memory is used. This in turn adds further hardware complexity and expense. We are familiar with the RISC designs with tens of millions of transistors that cost billions of dollars to develop. They also require a great deal of silicon which results in a high manufacture cost and heat related problems.

A very significant characteristic of those chips is that the hardware solutions to their complexity problems results in very non-deterministic performance. An interrupt service routine that usually executes in one microsecond will take one hundred microseconds a small percentage of the time when the interrupt happens at a time that results in pipeline stalls and cache misses. So to meet any realtime requirements one must pay for hardware that is a hundred times faster than what you need almost all of the time.

In zero operand architectures the CPU are much smaller and need fewer stages because they don't decode parameters when executing instructions. The hardware paths for each instruction are hardwired. There are only two stages, decode and execute required, and those can be mostly overlapped without pipelining. Without those operands the instructions and programs become very small and the hardware required to run them at high speed becomes very small. Most of the complexity associated with RISC or CISC architectures simply isn't needed to get high performance. The tiny instruction format in MISC makes it easy to get high instruction throughput from memory. MISC might be described as the smallest VLIW (Very Long Instruction Word) architecure.

Forth

Charles Moore invented the computer language "Forth" and later founded Forth Inc. [Moore, 1991] [Moore, 1970] [Moore, 1980] [Rather, 1993] Forth is incredibly simple. [Brodie, 1981] It provides a communication mechanism between the programmer and a computer. It involves a simple virtual machine implementation and an extensible dictionary of WORDS that can be data, functions, subroutines, or programs. It often includes its own OS and can compile modified copies of itself. Most Forths today by count are written in 'C' because they are in workstation ROMS or part of Linux distributions in large numbers. Writing the Forth virtual machine in 'C' is the easiest way to get widely portable versions of Forth today in 'C' based environments.

To make it easy to factor code into many small routines that become named WORDS the virtual machine supports two stacks. This means that there is no overhead for building or removing stack frames when Forth WORDS are invoked. A Return Stack holds return addresses for nested WORD usage. A Parameter Stack, or Data Stack is where most parameters are passed between WORDS. The parameter stack provides a place for unnamed temporaries and a mechanism for information hiding allowing the programmer to focus on the named words.

A dictionary of WORDS is created starting with words with standard names that perform basic manipulations of the stack. This dictionary includes a compiler and command interpreter and all operating system services or at least access to them if the Forth itself is not the OS.. Forth has very little syntax and is more of a semantic language than a syntactic language. Like in English or other natural languages words are defined in terms of other words. Forth and 'C' actually have a lot in common. Both provide low level access, efficiency, and a form of portability. But the explicit use of a parameter stack and a dictionary in Forth and the inclusion of the Forth compiler as part of many applications highlight some of the differences.

The following 'C' and Forth programs simply print the message "Hello World!" and issue a new-line to the default console driver. The 'C' program requires a library file include and the use of the name "main" as the top of the application.

```
/* Hello World example in C                                      */
#include <stdio.h>

main () {
   printf("Hello World!\n");
}
```

In Forth the application is just another named word in the name/code dictionary(ies). In this example the name "Welcome" is used. Using the name inside of another definition will compile a reference to this word. Using the name outside of a new definition will execute the word.

```
: Welcome ( -- ) ." Hello World!" CR ;
```

In the above line of code the colon defines the new word called "Welcome" and compiles code to print the string specified and issue a newline. A stack diagram, **(--)**, is added as a traditional style comment to specify the overall stack effect of this word to the programmer. In this case the word niether takes nor leaves parameters on the stack so there is nothing before or after the dashes. The semicolon at the end of the line is not simply an end-of-line character as in 'C' but specifies both a return from the word "Welcome" and a transition from compile mode to execute mode in Forth.

The Forth word ***/**, pronounced star-slash, is widely used for scaled fractional arithmetic. This provides a fast alternative on machines that do not have floating point support in hardware. The stack diagram of ***/** is **(n1 n2 n3 -- n4)**. It multiplies integers n1 by n2 producing a double cell intermediate result d then it divides that by n3 producing a quotient n4. Because Forth passes the arguments as untyped cells on a parameter stack this word simply consumes three arguments on the parameter stack and leaves one as a result. To use the Forth word ***/** in a Forth program one simply uses the name ***/** and it will either be compiled or executed. In Forth the top of the stack is the implicit place for operands.

In some Forth implementations the source for ***/** would be 'C' code and would specifiy a function using parameters on a parameter stack. In those systems one would use ***/** just as one would in any other Forth system.

In a more typical 'C' program the function would not use a parameter stack and would specify inputs and outputs to the routine. If coded as a function which would be called to return a value the paramters would be typed and passed explicitly.

```
/* C program example demonstrating:                              */
/*     1. Declaration of variables, indicating type;            */
/*     2. Defining and using a funcion;                         */
/*     3. Data conversion.                                      */
/*                                                              */
/* "C" does not automatically come with I/O functions. We have to*/
/* include the library with all the I/O functions if we want to  */
/* do any I/O:                                                  */

   #include <stdio.h>               /* standard I/O header file */

/* Also, any functions we want to define and use must be        */
/* described even before "main()". This is called a             */
/* Function prototype:                                          */

   int starslash (int first_mult, int second_mult, int divisor);

/* This tells us that it has three arguments (parameters), each  */
/* of which is a 32-bit signed integer. It returns another "int" */
/* value.                                                       */

/* Note that we don't have to actually USE the names "first_mult"*/
/* etc. when using or defining the function, later. For some     */
/* reason, "C" needs this definition at the beginning of the     */
/* program.                                                     */

/* At last, we can indicate the "main" part of the program.      */

   main()
     {
      int a;                       /* First multiplier          */
      int b;                       /* Second multiplier         */
      int c;                       /* Divisor                   */
      int result;                  /* Result of starslash       */

      /* Of course, alternately we could have declared this as

      int a,b,c, result;

      */

      /* Finally, leave the program with a result code of zero.  */
      exit(0);

     }

/* In the "main()" section is the actual function definition:    */
/*                                                            */

   int starslash( int in_m1, int in_m2, int div )
     {

      /* Define temporary, "long" variables:                   */
      long l_m1, l_m2, l_div;
      long l_result;
      int  result;

      /* Convert input parameters to their "long" work values:   */
      l_m1  = in_m1;
      l_m2  = in_m2;
      l_div = div;

      /* Or, explicitely, by "casting": l_m1  = (long) in_m1;
                                        l_m2  = (long) in_m2;
                                        l_div = (long) div;     */

      l_result = (l_m1*l_m2)/l_div;     /* Perform calculation  */

      result   = (int) l_result;        /* Convert back to "int"*/

      return (result);                  /* return the result    */

     }
```

Forth is really a meta-language because one extends the dictionary of WORDS to create the language that one wants to use to solve a particular problem and write a particular application. At the top level Forth should read very much like natural language. At a lower level Forth operates like the a processor; take this and take this and then do this with them. At the top level Forth operates like a human; it describes the operation of the program at the highest level of abstraction. Forth requires a different sort of thinking about problems and matching code to problems.

As a language Forth is very misunderstood. It has a reputation as an oddball quirky language and that is about all that many people know about it. People also associate Forth with the early days of microprocessors when there were few languages and Forth could stretch the resources of machines with limited computing power and memory. Many people think that Forth has not changed in twenty years, and for some people it hasn't.

Forth has changed and become more modern. There are ANS and ISO standards for Forth. Forth has found its way into the ROMS and onto the bus of all Sun workstations via the Open Firmware standard. Forth is found in small embedded computers from the local parking lot to deep space probes outside of our solar system. And one big change is that we now have some exciting very cheap and very fast Forth hardware on which Forth is simpler and more powerful than ever. [ANS 1994]

Many people still think that Forth is only a slow interpreted language but for almost twenty years people have been compiling Forth to native code that for the most part is as fast as anything else. It is also true that Forth is mostly used in embedded processing were processors still tend to be under powered and memory is still in short supply. But today's PCs are only overpowered for yesterday's applications. Having worked in AI and VLSI CAD I want to push the limits of a PC and not just do what I was doing yesterday.

Forth Hardware

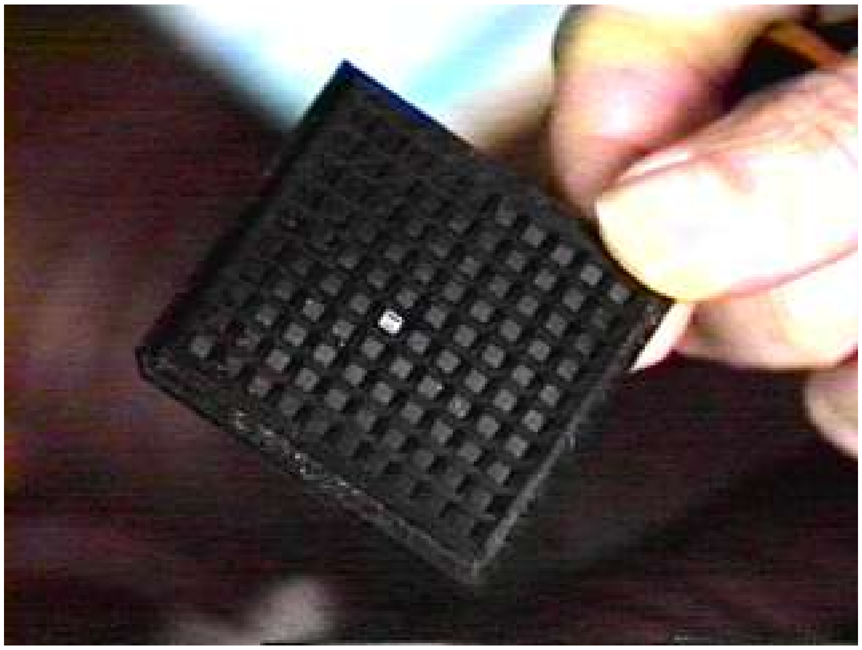
After twenty years programming in the language Charles Moore became interested in implementing the base of the language, the Forth virtual machine, in hardware. He left Forth Inc. to develop the NOVIX processor. The NOVIX was PGA technology which was fairly expensive at the time. Although FPGA are much cheaper today one still has to pay for the whole array of transistors and their fixed size and layout limits performance. The next design, the Harris RTX series used Standard Cell technology.

In 1988 Mr. Moore designed a 32-bit machine he called Sh-Boom. The zero operand architecture allowed instructions to be eight bits since the two stacks were the implied operands of most instructions. So four instructions could be packed into a 32-bit word and the CPU clock could be run at up to four times the memory clock cycle without the need for either pipelining or on-chip cache memory. The PGA design was sold and enhanced for a wider range of operations and PTSC has primarily marketed it as a Java chip, or a soft CPU for FPGA and ASIC designs that provides high performance while using few hardware gates.

Full Custom Very Large Scale Integration

I started working with Charles Moore in 1990 when he announced the he was going to write his own VLSI CAD software so that he could do custom VLSI designs without fighting conventional software that didn't understand the specific problems that CPU designers face. He stated that he was making the move to full custom because there was the potential to make designs with orders of magnitude higher performance and orders of magnitude lower cost than the technologies that he had used before. I worked with Mr. Moore for several years simulating various proposed instruction sets and architectural features and on his first VLSI design Mr. Moore used a CPU instruction set with 25 instructions thus instruction tokens were only five bits in length. His first VLSI engine processed these Forth tokens while optionally generating video.

An early MuP21 die in 1.2u scale. Today it could be made about four hundred times smaller.



Today these designs are available in many forms. Some in very high performance custom VLSI that have been done by Mr. Moore in his custom VLSI CAD software and some in portable high level description suitable for FPGA or ASIC construction. Twenty to fifty MIP CPU can be made in very small FPGA. Many parallel core can be placed in larger devices, or more gates can be used for memory for the small ML0 programs.

His first custom VLSI design, MuP21, was MS0 because it had small on-chip stacks. Dr. Koopman classified designs with more than eight cells supported in the stack to be large because above that stack spill/fill off chip was a very small percentage of the various applications examined in his research when the stacks become that large. The P21 design does not have the hardware to automatically spill or fill between on-chip stacks and memory stacks that is on the Sh-Boom and PSC1000 designs. When programs required larger stacks the programmer or the compiler would be responsible for stack spill/fill. It turned out that a fraction of one percentage of code that required spill/fill but I ordered larger on-chip stacks for the foxchip, the UltraTechnology F21.

Part of the reason for this was that my chip has analog, network, and video coprocessors on chip that can interrupt the CPU so more on-chip stack registers are useful to get faster interrupt service routines. For a number of years I simulated dozens of variations of the instruction set and architectures and wrote compilers and benchmarked simulated code to fine tune the instruction set to get the most performance on our actual programs. The 27 F21 instructions can represent about 75% of Forth programs directly as five bit tokens.

Cutting Edge of Silicon Circuit Design

Mr. Moore learned how to exploit the VLSI processes and push the theoretical edge for his components far beyond those widely used in the industry. The literature from the fab stated that their 1.2 micron process was fast enough to make a 55 MHz flip-flop. Chuck designed MuP21 in 1.2u and made a 100 MIP CPU. The literature said that 0.8 micron process was fast enough for a 75MHz flip-flop and Intel was able to make 64MHz processors in it. Chuck made systems of up to 500MHz in that same process.

For years we were told by many experts that our numbers for a 0.8 micron F21 chip, 2ns instructions, 500 MIPS, a programmable hundred megabit network coprocessor on-chip, a programmable composite and RGB video output coprocessor on-chip, a 40 MSPS analog I/O Coprocessor on-chip, a real time clock, an ultra high speed parallel port, and a 10 gigahertz echo-timer on-chip were simply impossible, that we were off by about an order of magnitude. The half million dollar CAD software packages had enough bugs in their silicon modeling and SPICE equations that they would confirm for the experts that the chips could not run even theoretically. Then we would show them running chips and people surfing the internet with them.

The F21 was designed to be, among other things, a very cheap and simpler personal computer. Because the design only requires 17k transistors it can be manufactured for a couple of cents for the silicon and a few more for the pins. The .8u process is what we were using for prototyping seven years ago. The design uses a scaled tiled layout that allows it to be made in any given process. Today the same prototype fab company that we used a few years ago offers 0.18u where we would have about 300 Pico second instruction time, 2700 MIPS, gigabit network I/O, 400 MHz analog I/O, 50 gigahertz echo timer, and from a chip that could be manufactured for a couple of dollars. If we had access to the cutting edge in silicon today such as a fab with .07u geometry and copper interconnect things would be about twice as fast again.

Some Applications

One of the things that I did to demonstrate one idea behind the F21 chip was that I removed the Motorola microcontroller chip from an old mouse and replaced it with one of my F21 prototype chips. The point is that the F21 is about the size and cost of the kind of micro that is in a mouse yet it is capable of making many users think that there was a Pentium PC inside of the mouse running some version of Microsoft Windows. I just plugged in an RGB monitor and ran a simulated desktop and application. [Fox, 2000]

F21 in a Mouse generating test pattern next to simulation running on a PC.

When I approached Mr. Moore I in brought an interest in parallelizing Forth. I was interested in networking many cheap Forth 'toy workstation' chips. This was the reason for a network routing coprocessor on-chip and the Forth Linda and F*F software. F21 addresses about 4 megabytes per CPU node which is far more room than is needed to implement a Posix compatible system. Chuck eventually became interested in the parallel programming paradigm. One might also note that all of Mr. Moore's designs have been word addressing machines and were not designed for byte addressing and byte manipulations. As a language Forth is about the semantics of words, computationally it has stacks of word sized cells. Of course including byte addressing is not a large hardware complication. It could be added and has to some of the other designs. [Fox, 1993]

One aspect of the architecture that interests me is that stack computers are quite adept at executing decision trees. They can do this as or more effectively than other architectures with hundreds or thousands of times more circuitry. So they are ideally suited to expert systems and many other AI programs often expressed in LISP. They are somewhat of a poor man's LISP or Prolog machine.

Perhaps the most important aspect of this technology is that it is intellectually penetrable. The assembly language is a high level language, understanding how it works takes a matter of minutes. Understanding how the hardware or the software works is remarkably simple because the hardware was designed with this in mind. We developed some remarkably simple and cheap Internet appliances at iTV that we hoped could popularize Forth Markup Language on the Internet, but none were ever brought to market. A few hundred million $25 Forth Internet computers online would change a lot of things.

When I was Director of Software at the iTV Corporation I trained the programming staff. I recall training people who had never heard of the architecture or used the language. I could describe the architecture and go over each and every instruction in full detail, explain the theory of the language, go over the details of the compiler and development tools in two hours and have the new engineers writing and testing their own new code the next day. I wonder how long it takes someone to learn every detail of the Pentium architecture, each and every instruction, and all about using an associated set of development tools, the high level language that they will use, the specific compiler, and all about their OS.

iTV Internet ready 5 inch B/W TV AM/FM with PC keyboard and example on the desk of the circuit board in the battery compartment in the TV.

It was designed to be something like a $25 iMac. Something very easy to use as a TV or an Internet browser and email appliance. Users loved it and it ran a nice Forth script on HTML pages. A few hundred million of these on the Internet would have changed a lot of things. The camcorder did not pick up the animated graphic and text on the Internet TV monitor well at the demo at the Forth Interest Group meeting. The idea was popular in Russia, China, Japan, India, Indonesia and other places where people were not conditioned to have PC expectations. In fact, many of our test users had PCs but prefered to use our simple instant-on Forthchip Internet appliances. We made a settop box, a handheld lcd based version, and a card that could be produced for about the cost of disposable batteries and went into the battery compoartment on a tiny really cheap TV.

Other Software

Our programs and our tools are written in Forth. Forth is a natural language for us because we have already been using it and now most of will be built into our hardware. Forth software on Forth hardware is hard to beat. With tiny tokens to represent the most common Forth functions our programs are remarkably small and modular. They are remarkably fast because very simple hardware is needed to decode and execute these tokens. This is true whether one talks about portable high level chip designs for FPGA or full custom VLSI chips.

Other languages can of course be used and people have writeen a variety of them in Forth or for Forth hardware including C, LISP, Java, Prolog, and LOGO. In order to be widely used by professionals a wide range of software support will be required. Our CAD software, our simulation software, and all our development systems are written in Forth so my personal experience has focused on Forth software.

Chuck Moore's latest Forth software is strongly influenced by his hardware designs and what he considers good style in Forth. He says that Forth is factored into many small easy to write, debug, and maintain sections of code called WORDS. He says a Forth is like a compression mechanism for code and that well written programs approach their minimal effective complexity. Effective complexity is the absolute minimal amount of code needed to represent some particular logical action. Chuck Moore created his own website last year called colorforth.com where one can get the story firsthand.

His hardware was designed to support a simple approach to software and to be fast and cheap. The software was designed to be as simple and efficient as possible. While a couple of C compilers have been produced they have not included significant optimization. There remains much work to be done to integrate this technology to the larger world of standard development tools. Forth was made public domain long ago and there was a PD Forth community before the new Free Software community. The fact that these designs and the required software is so simple makes them ideal for Free Software projects.

Mr. Moore met John Sokol while doing a presentation for the Tech Startup Connection, formerly the Parallel Processing Connection, about these designs and his approach to VLSI CAD. Mr. Sokol, well known in the BSD community and as an Internet video pioneer, expressed interest in a new design for a cluster chip with multiple CPU and memory nodes connected via register based communication busses. Mr. Sokol brought the interest and expertise to support integration with conventional software tools and Enumera was formed.

25X a 63,000 MIP Parallel Multiple Stack Computer Cluster Chip for $1

The latest chip design is more difficult to describe because the outside of the chip, the action of the pins, is programmable. It can be programmed to mate directly to an 18-bit cache RAM without the need for a circuit board. It essentially uses the same CPU architecture and instruction set as F21 from a programmer's point of view with an increase of the instruction count to 29. But the layout of gates is quite different and new and Chuck is exploiting some new tricks to push the state of the art in hardware performance. In addition to being an asynchronous processor design this design also uses more dynamic logic and has a more high level description of the layout in colorforth source.

25X it is designed to be a very small chip that could be sold for about $1 in quantity. Small volume prototype runs are substantially more expensive. 25X contains 25 CPU core each with its own memory. They are connected to each other via internal communication busses and to the outside world via the pins. With 25 2400 Forth MIP CPU on a one dollar chip the metric is 63,000 MIPS per dollar of hardware. A production cluster chip could be larger than a one dollar chip for millions of MIPS but a tiny one with only 25 CPU can be prototyped more easily. As Mr. Moore said, "They are small. They don't do much, but they do it very fast." [Moore, 2002]

The chip could be programmed to act as other more conventional parts, as a single chip or be used as a node in a larger parallel computer. Like an FPGA or general purpose computer it is very difficult to describe exactly what the things you can make with it could do. 25X is more like an FPGA with 2400 MIP Forth engines as logic blocks than a typical desktop processor. I think the 25X design looks like most of a multi-line gigabit Internet router on a $1 chip, something that one would snap on the end of a fiber to connect it to other fibers or anything else.

Because these designs are simple at both an abstract and gate count requirement level they seem to be the little engine that can. These highly gate and code efficient designs can often be the most efficient way to perform the programmable functions in some device. They allow cheaper FPGA or cheaper custom VLSI part options for many mundane computing operations because they are small and run compact code and they simplify the task of programming. To become more mainstream these designs will require development tools and pre-packaged hardware devices with software libraries. As more people become aware of these technologies perhaps people in the Free Software community may find ways to integrate the technology.

The UltraTechnology Website has many files and videos about the hardware, software, the failed and the sucessful experiments. We have established three mail lists to discuss developments in the field. There are many HTML pages and hours of online videos of Charles Moore, Jeff Fox, Dr. C.H. Ting, and John Rible giving presentations and tutorials on these subjects. Some of the more widely distributed free Forth systems today include gForth, BigForth, and PFE (Portable Forth Environment). iForth is also a commercial alternative for Linux with particular attention to floating point optimization.

References:

[Koopman, 1989] Koopman, Philip, Stack Computers: the new wave, Chichester England, Ellis Horwood, 1989, ISBN: 0-7458-0418-7.

[Moore, 1995] Moore, Charles H. and Ting, C.H., MuP21 a High Performance MISC Processor, available from Offete Enterprises, San Mateo, CA 94402.

[Moore, 1991] Moore, Charles H., Forth, the Early Years, 1991

[Moore, 1970] Moore, Charles H. and Leach, Geoffrey C., FORTH - A Language for Interactive Computing (pdf) (html), Amsterdam NY: Mohasco Industries, Inc. (internal pub.) 1970.

[Moore, 1980] Moore, Charles H., The evolution of FORTH, an unusual language, Byte, 5:8, 1980 August.

[Rather, 1993] Rather, Elizabeth D., Colburn, Donald R. and Moore, Charles H., The Evolution of Forth, in History of Programming Languages-II, Bergin T. J. and Gibson, R. G., Ed., New York NY: Addison-Wesley, 1996, ISBN 0-201-89502-1.

[Brodie, 1981] Brodie, Leo, Starting FORTH, Englewood Cliffs NJ: Prentice-Hall, 1981, ISBN 0 13 842930 8.

[DPANS 1994] (Draft) ANS Programming Language Forth, document number X3.215-1994, available from American National Standards Institute Sales Department or Global Engінerring Documents.

[Fox, 2000] Fox, Jeffrey A., F21 in a Mouse, March 8, 2000

[Fox, 1993] Fox, Jeffrey A. and Montvelishsky, Michael, F21 and F*F, Parallelizing Forth, FORML Conference Proceedings.

[Moore, 2002] Moore, Charles H., Internet Chat, May 5, 2002

Author:

Jeff Fox was born in Des Moines, Iowa and raised by foxes. He produced his first educational video about computers in 1965. He attended Southern Methodist University and the University of Iowa with majors in astrophysics, mathematics, dance, and Japanese, and studied digital sound synthesis, AI, and computer architecture at MIT. He moved to Berkeley, CA in 1978, and worked for the Bank of America and Training Department of Pacific Bell.

He wrote AI, expert systems and neural nets in Forth. In 1991 founded UltraTechnology to design and build an advanced Forth processor for parallel multimedia and AI and began working with Charles Moore. In 1995 he joined a new NASA Incubator startup, the iTV Corporation, to build Internet appliances using these Forth chips and became Director of Software. He was a former Vice-President of the Forth Interest Group.

He has practiced Zen and martial arts for over forty years with over thirty years of teaching experience. He received his fifth degree black belt in Aikido in 1995. He has published articles in several national magazines.