The Design of a Forth Computer

James C. Vaughan

2293 Santa Ana Street
Palo Alto, Ca 94303

Robert L. Smith

2300 St. Francis Dr.
Palo Alto, Ca 94303

Abstract

A relatively fast but simple computer architecture is outlined in which the machine language is very closely related to the high level language Forth. Speed is obtained by the direct implementation of Forth primitives and the parallel operation of a sequencer, arithmetic logical unit, data memory and stacks. Suggestions are made for certain improvements including the possibility of reducing the computer to VLSI.

Introduction

Most computer programmers prefer to write their programs in a High Level Language, but most computers execute in Low Level "Machine Code." Machine code varies greatly from one computer to another. Typically, a compiler is used to translate from the High Level Language to Machine Code. The difference between the two forms has been called the "Semantic Gap" [KAV]. It is well known that many processing errors occur during the initial phases of writing a program. When there is a large "Semantic Gap", it is frequently difficult or possibly costly to report the error to the programmer in terms of his High Level Language. Computers with a large Semantic Gap frequently execute less efficiently than computers with a small Semantic Gap.

For 20 years, computer architects have been interested in High Level Language Computers. These computers would either execute high level language source directly or require only a small amount of translation. The idea of a High Level Language Computer is certainly appealing. It would tend to reduce the number of conceptual levels that a programmer needs to know to be truly effective. Since one high level instruction often replaces several low level instructions, there is the possibility that the High Level Language Computer would execute programs more rapidly. Since the cost of developing software is usually much greater than the cost of the hardware, it appears that the hardware should be built to take over some of the software programming tasks. (Or at least some of the compiler writer's tasks.)

Critics of High Level Language Computers [KAV, DIT] have raised some doubts about the reduction of system costs, pointing out that the arbitrary migration of software to hardware will result in the exchange of software releases for engineering change orders [KAV]. It is almost universally believed that High Level Language computers are necessarily more complicated than current computers. A large complicated computer will not necessarily

either reduce costs or increase system speeds, and it would certainly reduce reliability. But is it necessary for a High Level Language Computer to be complicated? The goal of our current effort is to produce a High Level Language Computer which is both fast and simple.

Over the years programmers have tried to make their lives simpler and more productive by inventing a number of high level languages. ALGOL, which was conceived in the late 50's, has inspired a large group of languages which form the basis for structured programming.

Structured programming appeared in the late 60's as an attempt to make programs more readable and more reliable. Structured programming eliminates or at least severely restricts the use of the GOTO construct within a program. The restriction on GOTO results in programs which are linear and considerably less cluttered than they formerly were. The only constructs allowed are:

—Sequential operations
—IF instructions
—Repeat instructions
—Subroutines

The architecture of these structured algorithmic languages is simpler than the architecture of the computers on which they run.

Tree Structure

One of the classic methods for solving a complex problem is to divide it into a number of simpler problems and then proceed to solve each simpler problem. This procedure forms a hierarchy of problems or blocks. The top block represents the main problem and the lower blocks represent the sub-problems. This approach to problem solving is often called Top Down Design. The result is a tree structure. Structured programming is a method of implementing Top Down Design.

A program written using a structured Algorithmic language has a tree structure. The program is organized as a series of blocks under the main program. Figure 1 shows the possible structure of a simple program. All programs start with the top element of the tree called the root. The program is divided into a hierarchical fashion until the operations are performed at the very bottom of the tree structure. Each of the structured constructs can be represented by a different type of block. (Figure 2)

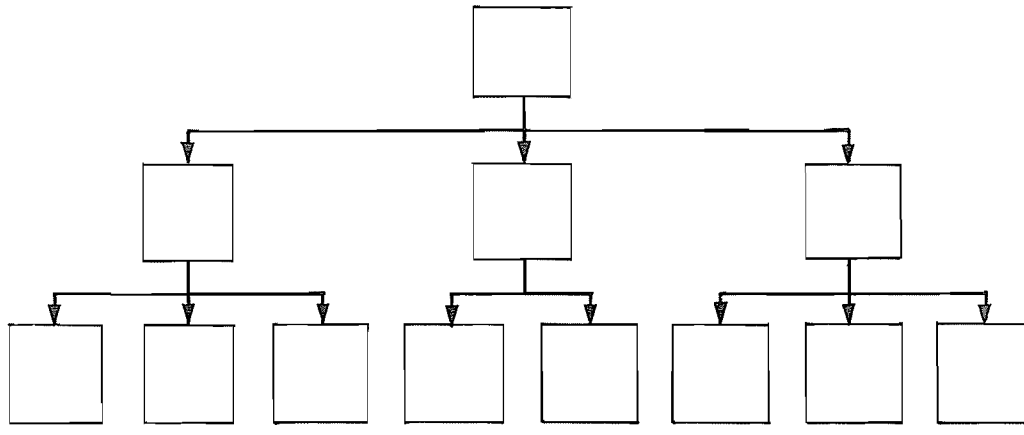

Figure 1. Tree Structure

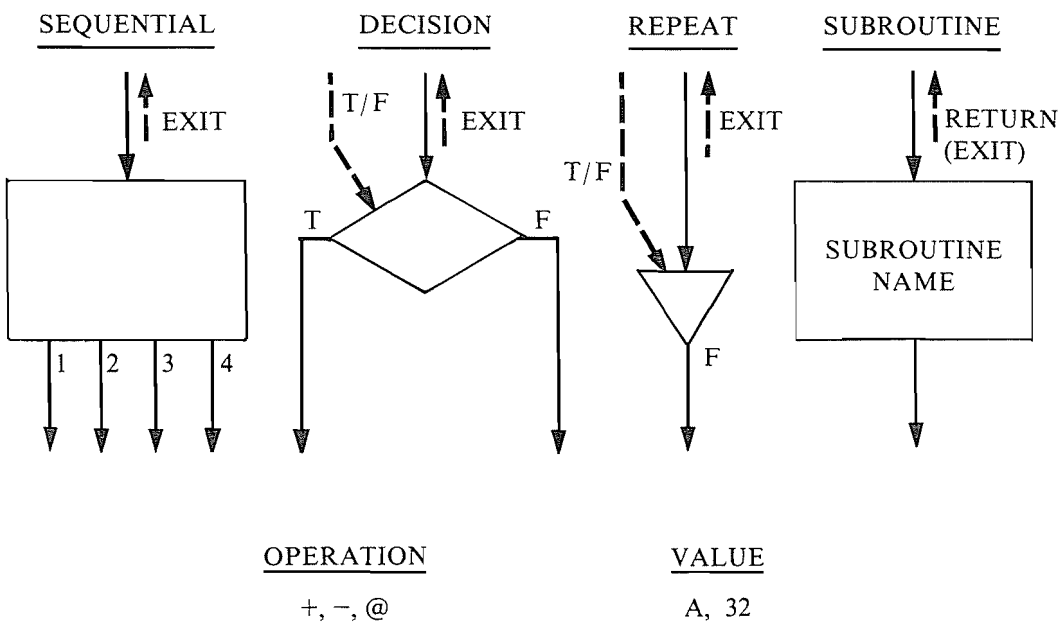

Figure 2. Tree Diagram Symbols for Structured Constructs.

The sequential block allows its statements to be executed left to right in sequence. In this construct jumps are not allowed. In most programming languages, this construct consists of sequential assignment statements.

The conditional block allows either the true statement or the false statement to be executed, but not both. The construct IF . . THEN . . ELSE . . ENDIF is the representation of the conditional block in most languages. The postfix operations in Forth make the appearance slightly different, but the structure is the same.

The repeat block allows its statement to be repeated as long as the condition is false and will exit when the condition becomes true. REPEAT . . UNTIL is a common form of the repeat construct. Figure 3 shows how the structured constructs can be represented in a tree.

There have been proofs that it is possible to write any program using the three constructs [DIJ]. The GOTO statement does not need to be used at all. Since the structured concept became popular, some method of prematurely exiting a block has been found to be convenient. The GOTO structure in Pascal and the "Exception" in ADA are used for this purpose. These constructs are intended to be a very restricted form of the labeled GOTO.

The subroutine construct allows portions of the program to be used several times. Subroutines are not strictly necessary, but allow compact code. Much of the clarity of good structured code involves the use of many small subroutines. Subroutines allow the encapsulation of complexity. After each subroutine is written, it can be used without having to know the details of how it was implemented.

Eventually, as the problem is divided into smaller and smaller pieces, there comes a time when some work has to occur. In a computer, that work is in the form of arithmetic and logical operations. Assignment statements such as A := B+C and logical expressions such as D<E are reduced to a series of arithmetic operations performed in a computer.

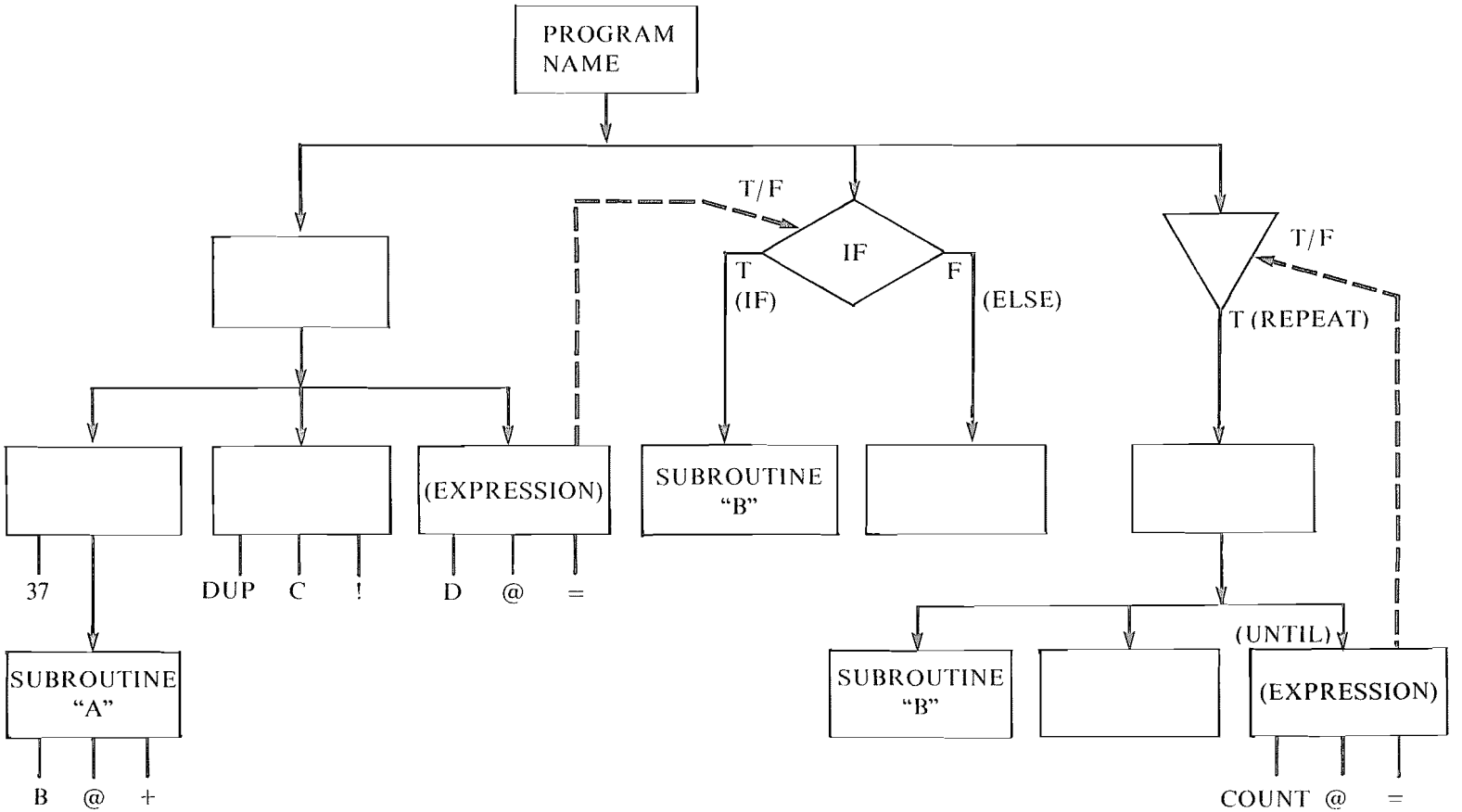

Figure 3. Examples of Structured Blocks in a Tree.

These operations appear in the tree structure as terminal nodes at the bottom of each of the lowest branches in the tree.

A computer is usually thought of as a machine which does arithmetic. However, the computer actually spends a great amount of its time finding its way through the tree to determine what operation to do next.

Thus we see that there are two different types of procedures to be performed in executing a structured high level language in a computer. The first is to work through the tree structure to find out which operation to do next and the other is to perform the operations. The two functions are very different.

All of the so-called structured constructs are included in the tree part of the model. The Forth functions, IF .. ELSE .. THEN, BEGIN .. END, and most of DO .. LOOP, are performed by the simple use of conditional and unconditional branch statements. The functions CALL and RETURN use a dedicated stack. The traversal of the tree is very straight forward and its method is outlined in any good book on data structures [KNU]. It is possible to design a very simple hardware device to perform this function.

The expression and assignment statements are performed at the terminal nodes by "operations." The hardware necessary to perform the operations is also well known. Since most high level language assignment statements are eventually decomposed into reverse Polish Notation (RPN), the arithmetic can easily be performed with a stack architecture.

It is possible, therefore, to separate a computer into two parts, as shown in Figure 4. Each of the two pieces performs a completely different function. The Sequencer scans the tree structure and determines the sequence of operations but does no arithmetic on its own.

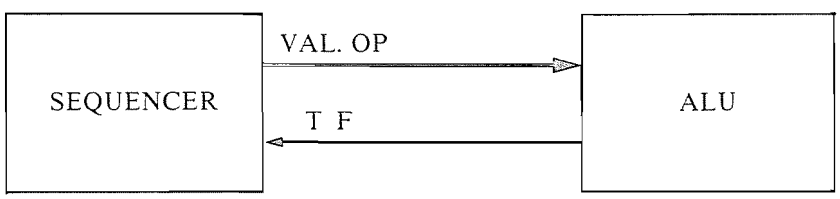

Figure 4.

The ALU performs the arithmetic and data memory references, but does not enter into the sequencing. The communication between the Sequencer and ALU is very simple. The Sequencer sends instructions and data to the ALU. The ALU sends a TRUE/FALSE signal as a reply. The TRUE/FALSE reply indicates the result of a condition calculated within the ALU and is used by the Sequencer to direct a conditional branch instruction.

The ALU can be designed to do 8-bit integer arithmetic or 32-bit floating point arithmetic without affecting the design of the Sequencer to any great extent. The Sequencer design can remain substantially unchanged for a large variety of ALU designs.

Our Forth computer therefore features a Dual design philosophy. The Sequencer's sole responsibility is to determine the next operation or next value needed for the program. It sends operations or values to the ALU. The ALU performs the operations without having to know how to get the next operation. The ALU sends a single bit of information back to the Sequencer to indicate a true or false condition so that the Sequencer can execute conditional branches. The ALU also has to send addresses and data to the Sequencer for starting tasks, answering interrupts and loading programs into program memory.

One of the advantages of this approach is that each of the two units is very simple. In this case simple means that there is a very short and direct data path. This allows very short timing cycles since the data does not have to travel through many series gates. It is possible to have the Sequencer operate totally concurrently with the ALU. The only time the two devices need to interlock is when the Sequencer has a command for the ALU or when the ALU has a condition necessary for the Sequencer to make a decision.

The sequentual procedures and the assignment or expression operations should be performed in as short a time as possible. Typically each operation requires a single microcycle for execution. There are obviously functions which require more than one cycle. Multiplication and Division in our implementation are done by a sequence of instructions.

Most previous high level machines have tried to directly implement the total language in hardware. Some have gone as far as trying to perform the compilation in hardware at run time. Usually the hardware complexity has more than outweighed the advantage of direct execution [KAV, DIT].

Our approach has been to try to make the architecture as simple as possible, consistent with direct execution of Forth primitives. We have tried to capture the essence of the language without worrying too much about the the little used or more complicated functions which require a lot of hardware but do not add much speed. We have done this by breaking the model into pieces which have specialized functions. Each function can be optimized for its specific needs without the burden of having to perform multiple functions. The goal has been to have as few series elements in the data path as possible. The result is a design which is very similar to a conventional bit slice processor. The Program memory has a very short microcode field, which allows the instruction to point to a much longer microinstruction field.

Using this approach we can take advantage of the inherent speed of a bit slice approach without having to have a very long instruction word.

In practical terms this means that a full operation can be performed in as little as 100 nanoseconds using TTL logic. This speed is several times what is normally accomplished in the 16-bit processors such as the 68000 microprocessor. We can build sufficient power into the indirect microcode word to implement a large number of arithmetic, logical and memory operations in a single microcycle. A 68000 microprocessor requires a minimum of 4 clock cycles to perform a simple memory fetch, and most instructions are much longer.

Speed

For this project the first major goal is speed of execution of programs written in a high level language. A good measure of the power of a computer is the time it takes to solve a problem. Most current machines are designed to run assembly level programs very fast. However, most programs are written in a high level language of some type. The basic need is to allow a high level language to run very fast. If the user is willing to write programs in highly optimized assembly code, the current microprocessors are very fast and efficient. However, the time required to write at that level of efficiency is much longer than writing the same algorithm in a high level language. A computer which is optimized to run a high level language at a similar speed, provides efficiency both for programming and running.

Speed is obviously dependent on the technology used for implementation. No matter what technology is used, a good way to shorten execution time is to shorten the data path for each cycle as much as possible. A short data path infers a very simple structure unencumbered by a lot of gates and multiplexors. The cycle time is basically established by the memory access time and the time to add two numbers. Using Fast TTL, it is possible to obtain cycle times faster than 100 nanoseconds. ECL allows cycle times less than 25 nanoseconds. Speed is generally very costly. Fast static RAMs are much more expensive

than dynamic RAMs. This design lends itself to fast static RAMs. The implementation leads to applications where small memories and high speeds are useful.

Ease of Conversion to Integrated Circuits

The second goal is to produce a design which can be converted to integrated circuitry at some later date. A problem with our design is the pin intensive architecture. One possible solution is to split the circuit. The most probable partition for the circuit would be into two integrated circuits, a Sequencer and an ALU. The Sequencer design will probably remain fairly constant for different computer generations, but the ALU would change depending on the requirements. At the present time, it appears that it would be possible to implement each of the two circuits using gate arrays.

Choice of a Language

The choice of a language to be implemented is very important. One of the most difficult parts of a project of this sort is the writing of the compiler. By using Forth as the target language for the computer, the effort of writing the compiler is greatly reduced. A computer can be built to run Pascal, C, or ADA, but the investment in building a compiler is enormous. Forth is simple, open and easily programmed.

The Forth language uses the principles of Structured programming. Although it is possible to use the unstructured "GOTO" construct in the Forth language, this construct is not part of the mainstream of Forth. This is different from languages such as BASIC and FORTRAN where the "GOTO" construct is the main branching element. In these languages the structured constructs have only recently been patched into the language. The Forth language is structured very simply, uses integer arithmetic and uses only global variables. The compiler is open for inspection and in fact is part of the language. Since Forth uses stacks extensively, it lends itself quite naturally to a stack architecture for a computer.

The computer is built to execute the run time portion of the language in as efficient a manner as possible. The compile time portion is handled as an application program. The results of the compilation are placed in the data memory and transferred to the program memory after compilation. Forth is different from many languages in that it allows incremental compilation. Each colon definition is compiled as each line is typed instead of waiting until the whole program is entered.

The interactive features are implemented with the use of a combination of compilation and run time code. The speed of many interactive processes is controlled by some human interaction such as waiting for a response on the keyboard. There is little to be gained by building special hardware for these interactive functions. Since the run time code is faster, this speed will also be reflected in the compiler as well.

Architecture Decisions

The first area where we can gain speed is by separating the instruction memory from the data memory. Each instruction requires a memory cycle and at least 30% of the instructions also require a data memory cycle. By separating the two, we can gain speed. The separation also makes each of the two halves of the machine much simpler. The Sequencer does not have to do very much arithmetic, and the ALU portion does not have to do very much sequencing.

One of the effects of the separation is to simplify each section. This reduces the length of the data path within each unit and effectively decreases the cycle time.

The two halves of the processor have independent timing. The Sequencer can look ahead several steps until it finds something for the ALU to do. The ALU contains a very simple microcode Sequencer and can perform operations which are several steps long. Using

this approach, it is possible to implement most operations directly in microcode. Many Sequencer operations can occur simultaneously with the operators and therefore do not occupy any time of their own. Branches, Calls and Returns can be pipelined in this way.

A conventional computer and the Forth pseudo-computer normally use only one memory for both program and data. In this computer there are two separate memories, one for data and the other for program. The program memory stores the program field addresses (PFA's) for each compiled word. The header name and other links are stored in the data memory. All variables are stored in data memory, but the addresses of these variables are stored in program memory.

There are several types of words in Forth. The structure words, such as IF, ELSE, BEGIN, are all implemented in the Sequencer part of the computer. These words are made up of the primitive operations, BRANCH, CONDITIONAL BRANCH, CALL, RETURN and CONTINUE. Since the Sequencer runs concurrently with the ALU, the structured constructs consume very little time in the program.

The arithmetic instructions are the commands which are normally used at run time. These are the stack operations such as DUP and SWAP, the arithmetic operations such as + and *, and the memory operations such as @ and !. Generally the structure and arithmetic operations are the instructions which are normally thought of as the language.

Forth also has operations for displaying data and for interpreting and compiling. Some have multiple functions. The Forth computer executes the structure and arithmetic operations as fast and directly as possible. Most of these instructions are executed in a single cycle. For instance, SWAP, DUP, +, and — are all single cycle instructions. The instructions * and / are multiple cycle instructions in this implementation, however * can be performed with a dedicated multiplier in a single cycle. The instruction ! is a 2 cycle operation since the stack must be popped twice.

Compiler instructions such as : are written as subroutines since they are quite complicated and are not part of the run time environment of Forth. Display formatting instructions are probably not included in the microcoded instructions set since these instructions are normally complicated and are too display dependent.

Sequencer

The sequencer consists of a program counter, a multiplexor, a program memory, a stack and registers to hold an operation for use by the ALU and a register to hold a value for the ALU. The stack is slightly more specialized than the Forth return stack in that it cannot hold the loop count or be used as an auxiliary stack for arithmetic operations.

The Sequencer, Figure 5, only has to execute a very few types of instructions, including BRANCH, CONDITIONAL, CALL, RETURN, OPERATION and VALUE. Since the Sequencer only has to execute a very few instructions, it can be made very simple and also very fast. The speed is controlled primarily by the speed of available memory. NMOS and CMOS static RAMs are available with access times as low as 25 nanoseconds. The memory speed of ECL is not much faster. We therefore chose to use high speed static RAMs in this design. At this time the memory devices are more expensive than the DRAMs used for most computers. The memory speed limitation probably limits the memory size.

The Sequencer is designed to allow for the processing of interrupts and for accepting tasks from the ALU. An interrupt is handled as an externally generated procedure call. Tasks from the ALU are handled in a similar fashion. During an interactive operation, the ALU interprets the command from the keyboard or source list and calculates the address of a routine stored in program memory. The ALU passes the address to the Sequencer as a task. The Sequencer uses the task as the address of a procedure in the program memory.

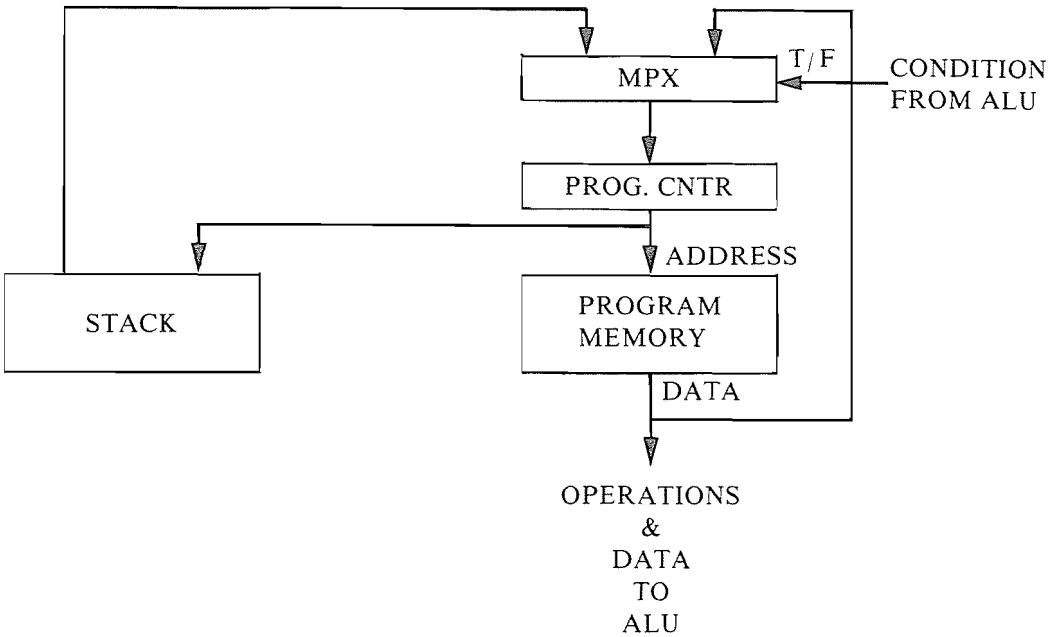

Figure 5. Sequencer

Since this is a stack architecture, there is very little necessity to store multiple registers whenever a subroutine is called or a task switch made. At that point most of the local storage is on the stack. The parameters will reappear when the interrupting routine completes. Forth naturally works this way. The structured languages probably can be made to work in the same way.

ALU

The ALU, Figure 6, employs a stack architecture consisting of a data memory, two independent stacks and an arithmetic integrated circuit for performing addition, subtraction and logical instructions. A microcode instruction PROM decodes the operation from the Sequencer. There is a very simple Sequencer built into the ALU. Instructions which require more than one microcycle, are programmed using the Sequencer in the ALU. This approach allows microcoding almost all of the Forth operations. Many operations require no more than a single micro-cycle.

The stack is implemented with two registers connected to the A and B busses and a RAM for the remainder. We have implemented two stacks in our machine. The parameter stack is used for normal arithmetic operations and the loop stack is used to hold loop indexes and also for the arithmetic functions normally performed by the return stack. The arithmetic element is a simple combinational arithmetic unit. We are using a standard 74181 for this purpose. Currently, we are doing multiply and divide with a shift-add algorithm, but we may add an integrated multiplier in the future. The data memory is a high speed static RAM.

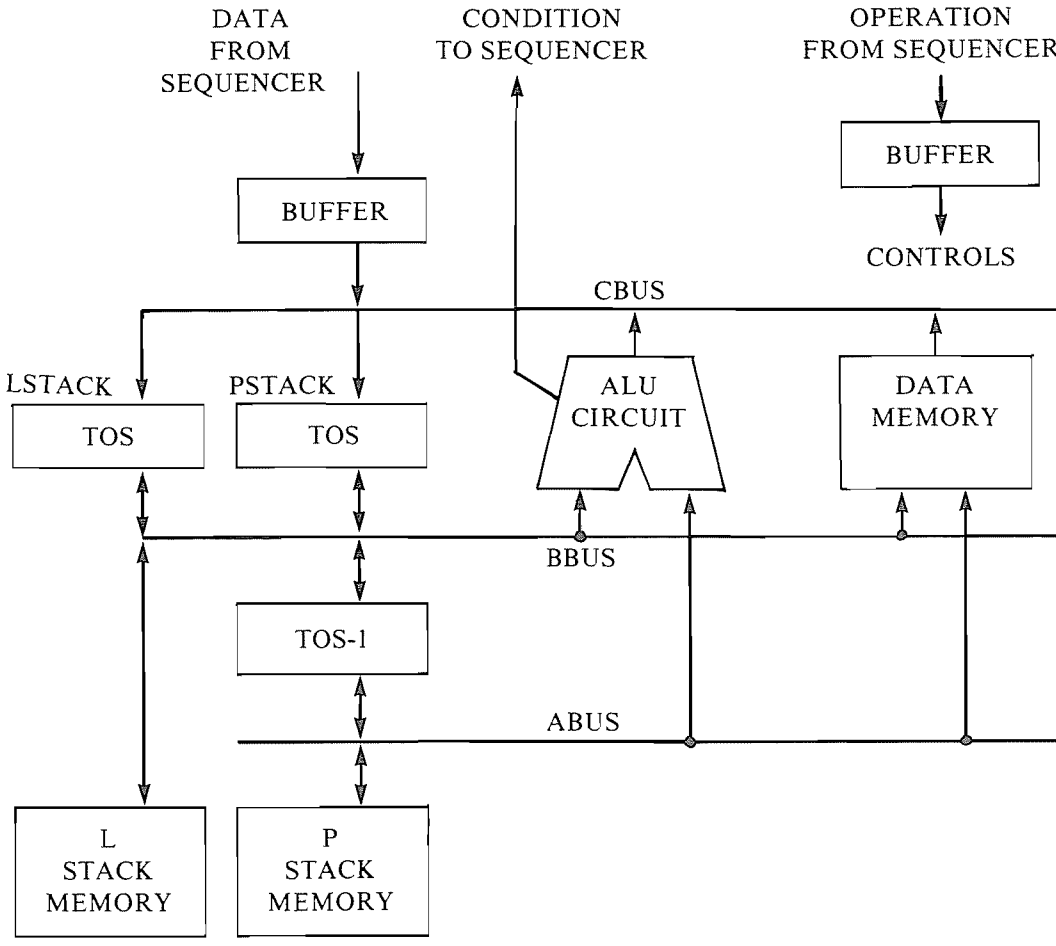

Figure 6. ALU

The data path for each microcycle typically consists of a register clock to output delay, a memory access time, a register set-up time and possibly a buffer or multiplexor delay. Each operation is designed to require a minimum gate delay. There is a single phase clock in the system. Very careful attention must be paid to the delays caused by such signals as status flags, because they often end up controlling the cycle time.

Compiler

Compiling a Forth program and running it on this machine is somewhat different than on conventional computers. The compilation is done within the ALU in the same way as any other application program. The compiler extracts and builds a symbol table in the data memory along with the appropriate set of flags and links. As each segment is compiled, the links and data memory addresses are transferred to the Sequencer for storage in the program memory. In this way only the code necessary at run time ever enters the program memory. If desired, the remainder of the code, consisting of the symbols, remains in the ALU data

OP	OPERATION
VAL	DATA OR DATA MEMORY ADDRESS
CALL	PROGRAM MEMORY ADDRESS
BRANCH	PROGRAM MEMORY ADDRESS
CONDITIONAL	PROGRAM MEMORY ADDRESS
RETURN	CAN BE COMBINED WITH OP OR VAL

Figure 7. Sequencer Instruction Format.

memory. The instruction format for a program is shown in Figure 7. Only 4 bits are required for commands, leaving 16 bits for the parameter field. The parameter field contains the BRANCH address, VALUE or the OPERATION, depending on the command field. An example of a compiled program is shown in Figure 8. On the left side of the page is the Forth source code taken directly from an existing Forth program. The right side of the page shows how this code can be compiled and stored in program memory. CREATE and VARIABLE assign memory locations in data memory in the ALU. CONSTANT assigns the value 0. In the colon (:) definition, each address or constant is represented by a VAL tag. The operations such as +! have a tag field of OP. The sequencing constructs such as BEGIN and REPEAT are built of BRANCH and CONDITIONAL instructions. CALL and RETURN manipulate addresses in the program memory. Figure 9 shows the tree structure of the program in Figure 8.

If a program is to be run, the ALU passes the address of the routine to the Sequencer and starts a task. A task is similar to a procedure call except that the starting address comes from the ALU instead of from the program memory. A CALL or TASK will put a return address onto the return stack. In a typical configuration the lowest level program in the computer is some kind of monitor. The monitor queries different devices until there is some function to perform. When an action is required, the start address of that action is passed to the Sequencer from the ALU by means of a task. When the task is complete, the task return causes the program to execute the monitor.

Microcode

The final Forth computer will not have an assembler in the conventional sense. The machine code is Forth. Since the computer is in two separate parts, there may be some differences between the Forth running on this machine and some of the other implementations. For instance FIG Forth is built on a model which assumes a single memory. Some of the constructs may not work with a double memory approach. The 79 Standard was designed to allow the separation into two memories. It is hoped that there are no major problems because of the use of separate memories.

Each instruction in the computer is microcoded. In most instructions the microcode is a single cycle long. There are a few instructions such as * and / which require more than a single cycle. One way to handle this case is to build a set of single cycle primitives into a

Forth Program		Compiled Code	
CREATE SCR#S 14 ALLOT		ASSIGN SCR#S -->	1000
VARIABLE PAGE#		ASSIGN PAGE# -->	1014
0 CONSTANT LOGO		ASSIGN LOGO -->	0
	ADDRESS	MEMORY	TAG
: PR (S SCR --)	: PR	1	VAL
1 SCR#S +!		1000	VAL
SCR#S @ DUP 2 * + ! ;		+!	OP
		1000	VAL
		@	OP
		2	VAL
		*	OP
		+	OP
		!	OP
			RETURN
: PR-FLUSH (S --)	: PR-FLUSH	1000	VAL
SCR#S @		@	OP
IF		TO THEN	CONDITIONAL
BEGIN	BEGIN	1000	VAL
SCR#S @ 5 <		@	OP
WHILE		5	VAL
LOGO PR		<	OP
REPEAT		TO REPEAT	CONDITIONAL
LOGO PR		0	VAL
THEN ;		TO PR	CALL
	REPEAT	0	VAL
		TO PR	CALL
	THEN		RETURN

Figure 8. Example Forth Program

colon definition and treat it as a subroutine. This allows the timing interaction between the ALU and Sequencer to be much simpler, but does not allow the fastest possible operation since there is a little overhead in a subroutine call and return.

An alternate approach is to build a very simple Sequencer into the ALU. The Sequencer allows the ALU to execute instructions with more than one cycle. Function such as * and / can now be treated as a single operation by the Sequencer. The command list can start looking almost exactly the same as standard Forth.

Since the Sequencer can now get ahead of the ALU, it is possible to increase the length of the pipeline by putting a FIFO (first in, first out) buffer between the Sequencer and ALU. A pipeline of this type works very well until a conditional branch command is executed. The Sequencer needs the feedback from the ALU to determine the direction of the branch. At this time the pipeline must be stopped until the branch is resolved. It is possible to partially

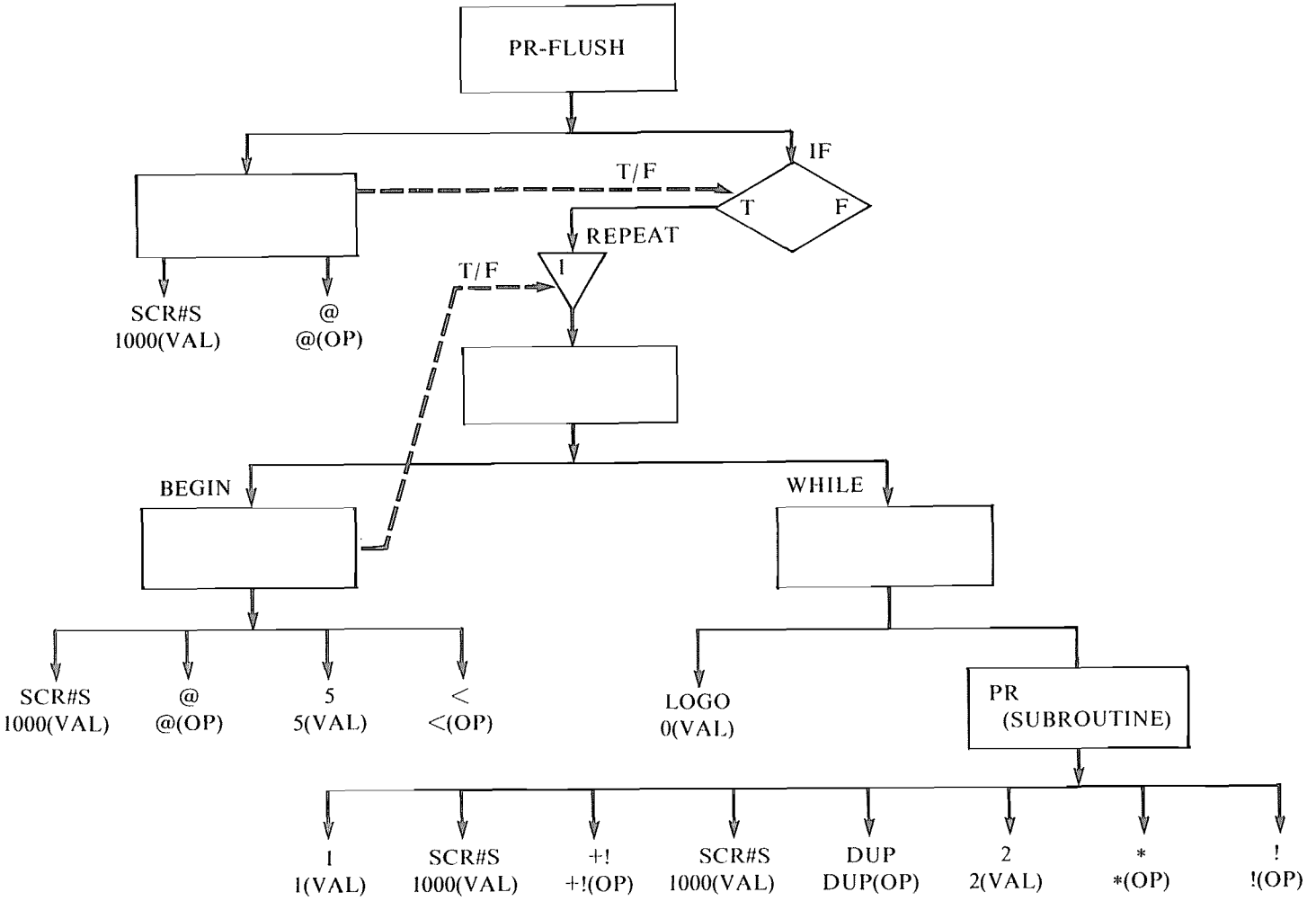

Figure 9. A Tree Representation of the Program in Figure 8.

resolve the pipeline problem by anticipating the direction of the branch and only aborting the pipeline when a wrong guess was made. Loops normally use a FALSE branch to loop.

The firmware will consist of a hierarchy of code. The lowest level is the microcode which will include all the run time operations in Forth. The next level will be a ROM in program memory and Data memory which will contain the Forth instructions and Variables which cannot be programmed as microcode. These routines will include such programs as the compiler and I/O controller.

Simulator

The design of a computer of this type is very difficult to debug. We are testing the structure of the computer on a simulator. Both the simulator and microprocessor are written in Forth. The simulator simulates each register and arithmetic element in the computer. It is possible to print out the state of each element after each clock cycle of the machine. The simulator should allow us to predict exactly how the hardware will behave for each microinstruction. The simulator for this computer required about 28 Forth screens of code. The simulator was written on a register level. Each internal register and memory location is a variable. The simulator steps through the machine states cycle by cycle and can display the internal state of any register at any time. Such simulators are simple to program and provide an insight which is difficult to see in the hardware without expensive tools such as logic analyzers.

The microassembler was written to aid in the development of the microinstructions used in the ALU. This microassembler was trivial to write in Forth. The <BUILDS DOES> construct in Forth allows writing the assembler kernel in one or two screens. The rest of the assembler consists of tables of op codes. An assembler of this type can be made to look and act exactly like a conventional assembler. We have used this same approach very successfully in the past for previous bit slice processor designs.

Prototype

A breadboard computer is being built on Multibus Cards to demonstrate feasibility. The Sequencer with 8K 20 bit words is on one card, and the ALU with 8K 16 bit words of data memory is on the other card. A commercial SBC 80/10 single card computer is in the third slot. The SBC 80/10 is used to provide an interface to a Kaypro 10 computer which is used as the host computer.

Microcode is assembled on the Kaypro and downloaded via the SBC into writeable control store on the Sequencer and ALU cards. Each bit of microcode RAM can be accessed via a serial chain of shift registers so that the host computer can view or modify the instruction at any time. The internal busses are also available on the diagnostic loop. By injecting microinstructions into the loop it is possible to control the internal operation of the computer.

The amount of memory used on the prototype was dictated by economics. The purpose of the prototype is to demonstrate that a computer of this type can work effectively. The amount of memory that this computer can address is limited only by the width of the memory address bus. In a Forth machine this is typically 16 bits. There is no reason why the address bus could not be 20 or 32 bits wide. The other consideration for memory size is memory access time. The larger memories suffer with a longer access time because of electrical characteristics of the drive lines.

At this time the prototype is in the process of being wired. We hope to have the Sequencer portion working first, followed by the ALU. After the hardware works, the Forth microcode will be implemented.

Evaluation Model

The prototype will be built slightly differently from the breadboard. The writeable control store will be replaced with PROM and the diagnostic loop will be removed. All microcode will be developed on the breadboard and burned into PROM for the prototype. The host computer will no longer be needed.

A full Multibus interface will be added to the prototype so that standard multibus interfaces can be used for the computer. Since the processor depends on high-speed memory, the Multibus interface will allow the use of standard memory for memory expansion. This will slow down the operation for accesses into the external address space because of memory speed for all accesses onto the bus but will allow expansion of the memory space. The internal memory is now used as Cache memory for fast operations. The Multibus also allows use of standard disk controllers or standard I/O modules.

Conclusion

The philosophic essence of Forth is simplicity. But simplicity depends on the context in which you are working. Consider two different types of machines in which each could be said to be simple. One machine could have extremely simple and fast hardware. A Turing machine might be of this type. The use of a high level language on this machine may require a substantial number of internal operations for a given conceptual operation. There may be many levels between the high level language and the native machine.

Consider another machine in which virtually every high level construct is interpreted directly by the machine. From the software point of view, that is simplicity. From the hardware point of view, it is likely to be a nightmare. Such machines have been built. Typically such machines have a large number of parts, causing a decrease in reliaibility and a fairly slow throughput because of the large number of gate delays or microinstructions required to execute a given function. The "art" of designing a high level language computer is the simultaneous selection of a hardware architecture and a suitable subset of functions to maximize speed and minimize the hardware requirements, and yet present a simple interface to the programmer. We do not claim that our current machine meets all of these goals, but we do feel that it is a strong step in the right direction.

Bibliography

[DIJ] *A Discipline of Programming* by Dijkstra, Edsger W., Prentice Hall, 1976.

[DIT] "Retrospective on High-Level Language Computer Architecture" by Ditzel, David R. and David E. Patterson, *Proceedings of the Seventh Annual Symposium on Computer Architecture*, May 1980, pp. 97–104.

[KAV] "HLL Architectures: Pitfalls and Predilations" by Kava, Krishna et al. *Proceedings of the Ninth Annual Symposium on Computer Architecture*, 1982, pp. 18–23.

[KNU] *The Art of Computer Programming, Fundamentals of Algorithms*, Vol. 1, Knuth, Donald E., Addison-Wesley, 1973.

[TAN] "Implications of Structured Programming for Machine Architecture" by Tanenbaum, Andrew S., *Communications of the ACM*, March 1978, Vol. 21, No. 3, pp. 237-246.

[WAK] *Microcomputer Architecture and Programming* Wakerly, John F, John Wiley & Sons, 1981.

[CHU] *High-Level Language Computer Architecture*, Edited by Chu, Yaohan, Academic Press, 1975.

Manuscript received April 1984.

Mr. Vaughan received his BEE degree from Cornell University in 1959. He has developed the hardware for a large number of digital systems including the Count Down Clocks for the Manned Space Center in Houston, display systems, Electrostatic printers and automobile traffic intersection controllers. Recently he has developed a series of high speed signal processing computers built around bit-slice components and 68000 microprocessors. The support software for these procssors has been the language Forth.

Dr. Smith received a BS in physics from California Institute of Technology in 1953 and a MS and PhD from Stanford University in 1955 and 60 respectively. In addition to many years of work in the field of very low frequency radio propagation, he has worked with computers since 1955. He is currently the secretary of the FORTH Standards Team.