QFORTH: A Multitasking FORTH Language Processor

Christopher Vickery

Queens College of CUNY
Flushing, NY 11367

Abstract

This paper presents the architecture of a computer designed to execute FORTH language programs efficiently in a multitasking, single-user, real-time environment. Issues addressed in the design include: a stack mechanism allowing high-speed primitive operations as well as dynamic stack creation and deletion, memory configurations tailored to the different requirements for code and data memories, and processor-controlled rapid context switching in response to interrupts and program events.

A prototype implementation using commercially available parts is under development.

Introduction

High-level language processors have garnered considerable research interest because of their potential for providing a better match between the hardware of a system and the software to be run on it. The result of research in this area should be more efficient compilers and run time environments than are currently available and, indirectly, higher programmer productivity. Chu (1975), Myers (1978), and Yamamoto (1981) discuss these issues in some detail and survey work being done in the area.

A more recent trend in computer architecture is the design of extremely simple and regular architectural structures, such as the RISC processor of Patterson and Séquin (1981). Such machines reimpose the burden of increased number of instructions per function performed, which is traded off against a very high rate of instruction execution, simple rules for compiler code generation, and regularity of design suitable for VLSI implementation.

Two factors suggest that high-level language architectures remain an important field of investigation despite the current vogue of RISC systems. First, one may predict that current work to develop more sophisticated CAD/CAE tools will reduce the advantage of RISC-like architectures for VLSI implementation by making less regular control and data sections more manageable than at present. Regular structures will always enjoy both a compaction advantage and an easier route to fault-tolerance through redundancy by iterated replication, but the design time for both reduced and complex systems should improve toward a floor in which the difference is not significant in the overall cost of the systems. The second factor to consider is the argument that reduced systems, being minimal by design, do not have the potential to yield performance improvements due to evolving hardware sophistication in their implementations, but more complex structures do. (Norton and Abraham, 1983).

A number of FORTH-language processors are on the market, and several noncommercial FORTH processors have been designed or built as well. Many of these processors are either conventional processors with FORTH software programmed into ROM or are

implementations of a FORTH instruction set as the microcode of the machine, using somewhat *ad hoc* techniques to provide the data paths and operations for that microcode. This paper takes a somewhat different approach from either of these. First, the present effort is tied to no existing model of either CPU design or FORTH implementation. A counter-example to this approach would be the Rockwell 65F11 and 65F12 processors which are tied to both a particular CPU (the Rockwell 6502) and a particular implementation model (the Fig-FORTH model, Ragsdale, 1981). As a result of independence, one may hope to abstract the features of FORTH which would profit most from hardware assistance and to use hardware structures most appropriate to the nature of the language. Not only should the resulting design show good utilization of hardware resources, but it may also prove suitable as an execution vehicle for code written in other high level languages insofar as "abstract FORTH" is representative of other languages.

In a time of rapid technological advances, it seems important to keep an architecture and its implementation separate. Rather than design a processor which can simply execute FORTH programs as quickly as possible using current technology, our goal has been to produce a design which relies for its power on structures which are not only appropriate but which also should benefit from evolving technological trends. Examples of such structures are memories and rich control logic as opposed, for example, to complex data paths. This approach may result in an architecture which is relatively expensive to implement using presently available resources, but which should become more attractive in the future rather than simply obsolete.

Of course a design cannot be made in a vacuum; its applications must be considered. The two environments we hope to serve are single-user program development and multi-tasking execution of real-time process control applications. Thus, the present design is not concerned with multi-user protection schemes nor with any but the most loosely coupled multiple processor configurations. While such a FORTH processor might coexist with other processors in a personal workstation in order to share peripherals, the processors would not work together on common chores. On the other hand, the execution environment requires multiple tasks to share the FORTH CPU effectively, with possibly severe real-time constraints placing a premium on rapid context switching and efficient intertask synchronization and communication.

This paper begins by exploring the architectural decisions that must be addressed before designing a FORTH-language processor to operate in the environments outlined above. We then present an architecture in the context of those decisions, and finally we describe briefly a current project to implement that architecture.

Architectural Decisions

With some overlap, key issues in the design of a FORTH architecture are stacks, control flow, memory, and (in the present case) multitasking performance. One way to evaluate the ability of an architecture to address these issues is to consider the number of microcycles needed to perform various operations. In this way, different approaches may be compared independently of absolute clock speeds. A microcycle corresponds to one control state setting, and would be equivalent to one microinstruction in a microprogrammed processor. Unfortunately, microcycles alone are not perfectly indicative of the times required for operations. Other variables include the number of clock phases required per microcycle and how machine instruction cycles are constructed from the microcycles. Furthermore, delays due to memory cycles, which may be initiated in one microcycle and awaited for completion in the same or subsequent microcycle, must also be considered.

Stacks

Pushdown stacks are the natural mechanism for temporary operand storage and for ordering control flow in a FORTH system. Although Charles Moore has claimed that "there

must be exactly two stacks, no more, no less" (Moore, 1980), additional stacks such as a "loop stack" for run time index values should improve performance as well. The importance of architectural support for stacks is further underlined by the fact that each task in a multitasking system must have its own set of stacks if one is to allow preemptive scheduling. This is because task activation is not nested the way procedure calls are in a single-task environment. Assuming that tasks are to be created and deleted dynamically, then it must be easy to create and delete stacks dynamically too. The consequent problem of fragmentation of stack memory space must also be considered. Although only those stacks belonging to a particular task need to be accessible at one time, real-time constraints require that changing the set of accessible stacks must be done rapidly.

A further stack-related architectural consideration is the observation that the top two elements of the parameter stack normally serve as ALU operands or as memory address and data. For example, Wada (1982) has designed a FORTH system which includes two busses connecting the top two parameter stack elements to the appropriate memory and ALU ports.

Finally, FORTH's stack primitives (DUP, DROP, SWAP, OVER, ROT, PICK, R>, R, >R) should be implemented efficiently. Ideally, each operation should require just one microcycle to execute.

Control Flow

FORTH is closely associated with the notion of threaded code. Two common forms of threaded code are directly threaded code, DTC, (Bell 1973), and indirectly threaded code, ITC, (Dewar, 1975), which is used in the Fig-FORTH implementation model (Ragsdale, 1980). There is nothing about the language to prevent compilation into directly executed machine code, and several "native mode" compilers claiming significant execution speed advantage over threaded interpretation are now on the market. Nevertheless, DTC and ITC provide extremely modest memory requirements, the advantages of interactive program development, and ITC is a particularly convenient way to provide for the DOES> word of meta-definitions. A processor which provides threaded interpretation as its "native mode" could provide both good run time performance and convenient implementation facilities. Of course, ITC must always pay a speed penalty for extra memory cycles compared with DTC.

Memory

The issues to consider for the memory of a FORTH system include segmentation, protection, and allocation. Ignoring stack memory for now, there remain different characteristics between the memory used for code and the memory used for data. Code memory, which might also contain program constants, may be allocated statically as each task is created, and should allow code to be shared among tasks. Certain applications may require all or part of code memory to be implemented as ROM. A single-user environment places little premium on memory protection schemes for code memory, and FORTH's vocabulary links in the dictionary structure may be used for controlled access to "kernel" code routines.

Data memory, on the other hand, must provide for both shared and private areas of memory for different tasks. Buffer pool management and recursion suggest that areas of data memory should be allocated to tasks dynamically, and the algorithms for such allocation are well enough established to consider providing this feature in the processor's instruction set. Debugging can be facilitated if some form of access checking to data can be performed by hardware.

These differences between code and data memories suggest that they might profitably be designed as separate entities. Code memory would be unsegmented, unprotected, and allocation would be under software control. Data memory would support segmentation with hardware allocation and limit registers for access checking. In addition to the advantage of specialization of function, memory separation provides a natural opportunity for parallellism through concurrent accesses to both memories. A possible disadvantage of this approach is

the inability to partition code and data functions to suit particular applications. A solution to this problem is to provide a very large address space for both types of memory, and then to populate those spaces with actual memory cells according to individual system needs.

Multitasking

The fundamental issues in a multitasking system are scheduling, synchronization, and communication. In addition, task switch time and interrupt response time are important parameters of system performance. If a task and its processor state can be clearly encapsulated, there is no reason that the processor itself cannot do preemptive event-driven task scheduling. A requirement of such a system would be to provide primitives in the instruction set for intertask signalling (semaphores) and communication (mailboxes). As the Dorado processor has demonstrated (Lampson and Pier, 1980), single microcycle task switching is possible provided a set of processor registers is available for each separate task.

The context switch time in a FORTH machine is confounded by the stack system. Conventional "hardware stacks" cache the top few elements of a stack in CPU registers and store the remainder of the stack in primary memory (Lonergan and King, 1961). The more stack elements are kept in the CPU, the faster basic stack operations occur, but each cached stack element adds one extra memory cycle to the context switch time, and the initial stack accesses after a context switch are at memory cycle speed as the stack cache is refilled.

A FORTH Architecture

Data Section

The data section of the "QFORTH" architecture is presented in Figure 1. The four major components of the processor are the stack system (S-memory), the task environment memory (E-memory), code memory (C-memory), and the data memory (D-memory).[1]

The stack system is a controller and memory which operate asynchronously with respect to the rest of the processor. It can perform each of the stack operations listed in Table I in one "stack cycle." A single stack cycle may span more than one microcycle, depending on the implementation chosen for this subsystem and the particular function performed. Stack selection is based on the P, Q, and R stack ID registers, with P and R normally representing a task's parameter and return stacks respectively. The stack system places no constraint on the number of stacks which may exist (this number is determined by the number of bits in the stack ID registers), nor on the size of individual stacks except that the aggregate size of all stacks at one time may not exceed the amount of memory available to the stack system. An output line to the processor causes an interrupt if this situation should arise. The S and T ports normally output the second and top elements of the selected stack, but special operations allow the tops of two different stacks to be output for WHILE and DO loop testing. The normal inputs to the U port are results from the ALU or a stack ID register for stack selection. Four status lines indicate when the S and T ports are available and whether they contain valid information or not. Validity refers to the existence of an element in the selected stack, while availability is used to synchronize processor access to values at the ports. While the intended implementation for the stack system is linked lists, the QFORTH architecture itself does not rule out alternative techniques.

The task environment memory (E-memory) contains "task state words," one for each active task in the system. Loading a task ID number into the E address register causes a new task state word to be loaded from E-memory and the current task state word subsequently to be rewritten, effecting a complete context switch in a single cycle of this high-speed memory.

[1]The four components are named S, E, C, and D in recognition of Landin's architecture for evaluating lambda expressions (Wegner, 1968). The actual relationship between Landin's SECD machine and the present architecture is beyond the scope of this paper!

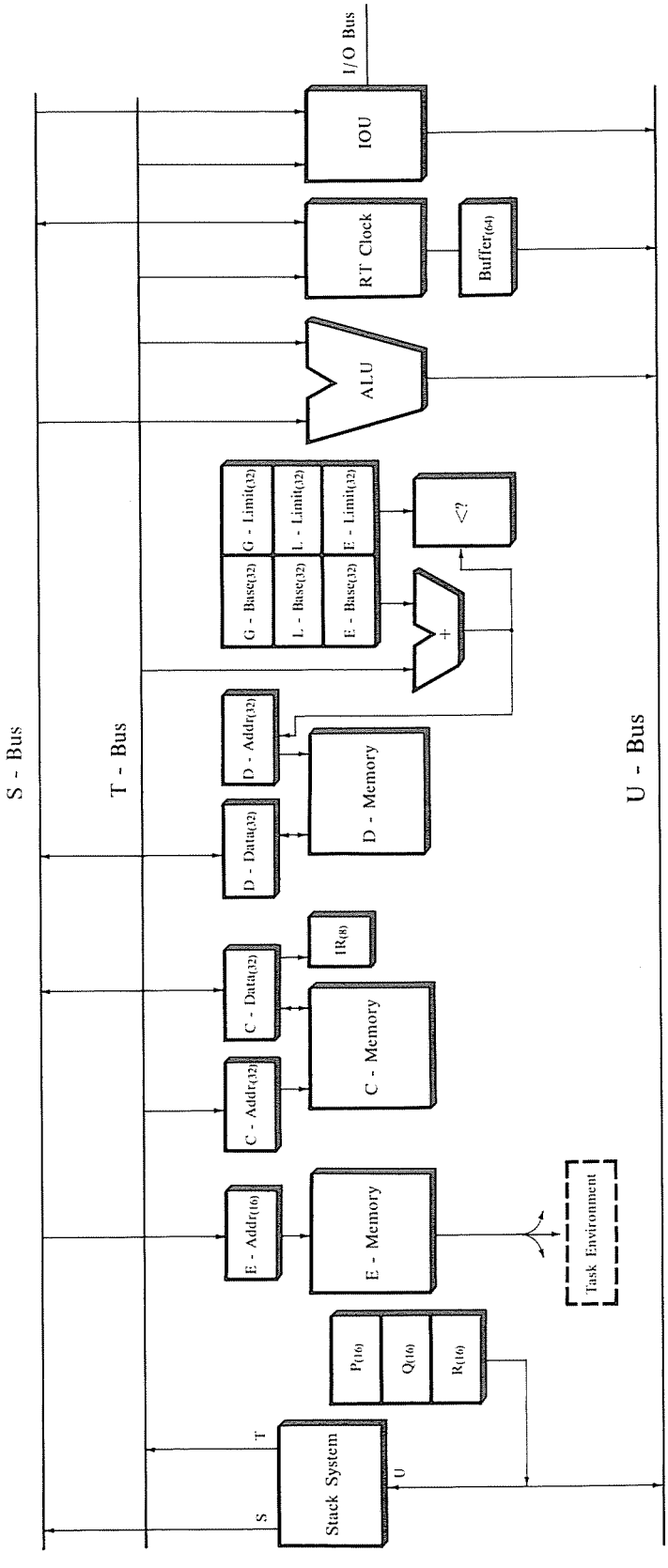

Figure 1. Data section of the QFORTH processor. Busses are 32 bits wide, and register sizes are indicated in parentheses.

Task state words consist of a status register (task allocation status, task priority, condition code, and stack status), the program counter, three stack ID registers, and three pairs of D-memory base/limit registers.

Code and data memories are separated for the reasons mentioned in the previous section, with an unsegmented 2^{32} byte code memory (C-memory) plus a separate data memory with three 2^{32} byte segments and hardware memory allocation. The C-memory is shared among all tasks, with a software kernel maintaining a map of vocabularies resident at a particular time. Dictionary headers are separate from code and are kept in D-memory segments. The D-memory segments for each task are the global and local data/dictionary segments (G-segment, L-segment), and an "extra" segment (E-segment). There is just one G-segment for all tasks, starting at absolute D-memory address zero (there is no actual G-segment base register). The G-segment contains the interrupt vector (a list of task IDs), the C-memory vocabulary map, ready and sleeping task queues, and the global dictionary structure. Traditional "user variables" may be stored in the G-segment as well. A task's L-segment (which it creates itself) holds vocabularies and data structures unique to that task. Since a segment is defined by a pair of arbitrary 32-bit values (base and limit), subsegments may be created and "catalogued" in the G-segment (as user variables) to allow controlled sharing of data structures and multitasking objects (semaphores, mailboxes, queues) with other tasks. The E-segment register pair is used for accessing shared memory areas and objects efficiently. As in other FORTH machines, busses connect the memory address and data ports with the ALU inputs and the two stack system output ports.

The remainder of the data section of the processor is devoted to the ALU, the I/O unit, and a real-time clock. The ALU performs Integer, Unsigned, Doubleword, and Floating-point operations. Byte string operations are also provided with operands assumed to be formed with the string length in the first byte; to facilitate dictionary header searches, a string mask register specifies those bits in the first byte of a string which are to be ignored in determining the string length value. The I/O unit controls its own bus with 16 address and 16 data lines. DMA operations require a separate controller not shown in Fig. 1 linking the I/O bus and separate ports to C-memory and D-memory. I/O devices may cause interrupts at any time. The CPU response is to access a task ID from one of 256 locations in the G-segment and to add that ID to the ready queue, which is ordered by priority. A task switch occurs any time the head of this queue has a higher priority than the running task.

To provide adequate real-time support, the real-time clock is a processor in its own right. It maintains a microsecond counter and a date-time ASCII string, which can be pushed onto the P-stack or moved to D-memory respectively. These operations work on buffered values so as to appear indivisible even though they may span several clock cycles. In addition, the real-time clock keeps a list of "wakeup" times, and causes one task to be moved from the sleeping to the ready queue in G-segment as each wakeup time is reached. As with I/O interrupts, inserting a task on the ready queue with higher priority than the running task causes an automatic context switch.

Instruction Set, Control Flow, and Data Types

The design of the instruction set is somewhat RISC-like in philosophy. Rather than a multitude of varying length instructions with a rich set of operand addressing modes, the instruction stream consists simply of a sequence of operation codes (8 bits) and 32-bit words. The words may contain C-memory pointers, pseudo-pointers, or values. Although data operands may be stored and accessed as bytes, halfwords, words, or doublewords, all values in the instruction stream (and all stack entries) are word-sized. Values in the instruction stream may be fixed or floating point numbers or pointers to byte strings or other data structures in D-memory.

Instruction processing is quite conventional despite the integration of (direct) threaded interpretation into the control flow scheme. Two of a task's three stack ID registers hold IDs

for the conventional parameter and return stacks. Execution of operation code bytes from C-memory proceeds until a CALL instruction is executed, which loads the program counter (PC) from the pointer value following the op code byte and pushes the address of the next byte onto the return stack as in a conventional machine. A conventional return instruction (RET) pops the top of the return stack into the PC. A threaded return op code (TRET), however, not only copies the top of the return stack into the PC, but also replaces that value with itself incremented by four (the address of the next pointer word in the instruction stream). "P1" is pseudo-pointer with the unique numerical value of all binary ones which is placed at the end of a list of pointers in the instruction stream. When the processor executes a TRET and finds a P1 to put in the PC, it discards the pseudo-pointer and loads the PC with the next word on the return stack. This operation may recurse for a single TRET. Although it is not logically essential, a second pseudo-pointer, P0 (a word with all binary ones except for the rightmost bit) is also defined which increments the PC by four rather than loading it from the return stack. Figure 2 demonstrates control flow using P1 and P0

Memory

Address	Contents
0500	CALL
0501	1000
0505	-P0-
0509	OP
⋮	
1000	CALL
1001	1500
1005	2100
1009	-P1-
⋮	
1500	CALL
1501	1700
1505	1900
1509	-P1-
⋮	
1700	OP
⋮	
1717	TRET
⋮	
1900	OP
⋮	
1919	TRET
⋮	
2100	OP
⋮	
2121	TRET

Execution Trace

PC	R-stack
0500	
1000	0505
1500	0505, 1005
1700	0505, 1005, 1505
⋮	
1717	0505, 1005, 1505
1900	0505, 1005, 1509
⋮	
1919	0505, 1005, 1509
2100	0505, 1009
⋮	
2121	0505, 1009
0509	
⋮	

Figure 2. Sample program in memory and execution trace of Program Counter and Return Stack demonstrating threaded interpretation and pseudo-pointers. "-P0-" and "-P1-" are "pseudo-pointers" as described in the text, CALL and TRET are machine language operation codes also described in the text, and OP represents an arbitrary machine language operation code.

pseudo-pointers. Bits in the task status word are set whenever P0 or P1 values are fetched from C-memory, so user-defined interpreters can easily test for these values as well.

The instruction set is very rich with approximately 200 of the 256 possible operation codes assigned. Table II outlines the operations by category, with the number of op codes in each category in parentheses. The combination of threaded control flow and stack-based addressing modes in the instruction set make comparison of the QFORTH architecture with conventional FORTH implementations difficult at best. When FORTH is implemented on a processor such as the PDP-11, a rich set of addressing modes facilitates interpretation of either direct or indirect threaded code, albeit at a price, over straight machine code which depends in part on the degree to which a particular application has been divided into a hierarchy of word definitions. In particular, user-defined interpreters may run faster on QFORTH more because of the PDP-11-like addressing modes available for accessing lists of pointers (auto-increment) than because of the TRET instruction and pseudo-pointers. Furthermore, user-defined interpreters pay a speed penalty over colon defined words in QFORTH both because of the use of DTC rather than ITC and because conventional interpreter functions (Fig-FORTH's DOCOL, DOCON, DOVAR, DOUSE, and DODOE) are, in essence, part of the instruction set. A primary goal of implementing QFORTH is to provide an efficient vehicle for evaluating the importance of these variables on performance.

Implementation

The goal of implementing the QFORTH architecture at this time is to provide a vehicle for collecting data to analyze its performance. Satisfying this goal requires effective means for testing and monitoring the processor, but does not necessarily require implementing all of the processor's logic in hardware. The resulting approach may be thought of either as a hardware-assisted simulator or a simulator-assisted processor. In either event, the scheme adopted is to use a conventional Multibus-based microcomputer system, complete with CPU (an 8086), memory, peripherals, and a real-time multitasking operating system (iRMX-86) as a testbed for the QFORTH processor. The QFORTH processor is to be built on wire-wrapped boards which plug into the Multibus backplane and which connect to one another through their own set of edge connectors. In this way, the QFORTH processor may use windows into Multibus memory for its own memories, saving initial development of that part of the system hardware as well as providing a convenient mechanism for the 8086 processor to set up test routines and to monitor execution by the QFORTH processor. Initially, the 8086 and a simple hardware interface will also be used to simulate the QFORTH stack, I/O, and real-time clock systems as well.

References

Bell, R. Threaded code. *CACM*, 1973, 16(6), 370–372.

Chu, Y. *High-Level Language Computer Architecture.* New York: Academic Press, 1975.

Dewar, R. B. K. Indirect threaded code. *CACM*, 1975, 18(6), 330–331.

Lampson, B. W. and Pier, K. A. A processor for a high-performance personal computer. *Proc. 7th Intntl. Symp. on Comp. Arch.*, SigArch/IEEE, 1980, 146–160.

Lonergan, W and King, P. Design of the B5000 system. *Datamation*, 1961, 7(5), 28–32. Reprinted in Siewiorek, D. P., Bell, C. G., and Newell, A. *Computer Structures: Principles and Examples*, New York: McGraw-Hill, 1982.

Moore, C. H. The evolution of FORTH, an unusual language. *Byte*, August, 1980, 76–90.

Myers, G. J. *Advances in Computer Architecture*, New York: Wiley, 1978.

Norton, R. L. and Abraham, J. A. Adaptive interpretation as a means of exploiting complex instruction sets. *Proc. 10th Intntl. Symp. on Comp. Arch.*, SigArch/IEEE, 1983, 277–282.

Patterson, D. A. and Séquin, C. H. RISC I: A reduced instruction set VLSI computer. *Proc. 8th Intntl. Symp. on Comp. Arch.*, Sig/Arch/IEEE, 1981, 443–457.

Ragsdale, W. F. *Fig-FORTH Installation Manual — Glossary, Model, Editor*. San Carlos: FORTH Interest Group, 1980.

Wada, K. System design and hardware structure of a FORTH machine system. *Syst. Comp. Controls*, 1982, 13(2), 11–18.

Wegner, P. *Programming Languages, Information Structures, and Machine Organization*, New York: McGraw-Hill, 1968, 216–223.

Yamamoto, M. A survey of high-level language machines in Japan. *Computer*, 1981, 14(7), 68–77.

Manuscript received May 1984.

Christopher Vickery is Associate Professor of Computer Science at Queens College of CUNY. He received the BA from Hamilton College in 1964, the MA from Columbia University in 1966, and the Ph.D. from CUNY in 1971. He is a member of IEEE, ACM, and FIG.

Table I
Stack Operations

Function	Description
0	CREATE a new stack. Return a unique identifier for the stack at port T.
1	DELETE a stack and reclaim its memory space. The ID for the stack to be deleted is presented at port U.
2	SELECT a stack. A stack ID is input at U, and the top two elements will be available at T and S.
3	DUPLICATE the top element of a stack.
4	DROP the top element of a stack.
5	SWAP
6	OVER
7	ROT
8	PICK
9	PUSH a new value from U.
A	SPLICE the top *n* elements from the top of the selected stack onto the stack whose ID is at U.
B	Copy the top of the selected stack to the top of the stack whose ID is at U.
C	MODADIC operation. The top of the selected stack is replaced by the value at U.
D	DYADIC operation. The top two elements of the selected stack are replaced by the value at U.
E/F	NOP

Table II
QFORTH Instruction Set Summary

I. Dyadic Parameter Stack Arithmetic and Logic
- Integer arithmetic and logic (13)
- Unsigned arithmetic (2)
- Doubleword integer arithmetic (4)
- Floating-point arithmetic (6)
- Convert from numeric to byte string (3)

II. Monadic Parameter Stack Operations
- Negate, Absolute Value (6)
- Increment, Decrement (8)
- Convert from numeric or byte string to numeric (6)
- Integer sign extension (1)
- Numeric test (1)
- Signed, Unsigned, Doubleword compare (3)

III. Stack Manipulation
- FORTH operators (DUP, etc.) (7)
- Push small constants (4)
- Copy top of one stack to top of another (1)
- Move *n* words from one stack to another (1)
- Two stack dyadic operations (6)

IV. String Operations
- Move, Concatenate (2)
- Compare (1)
- Insert, Delete, Extract (3)

V. Memory Operations
- Control Memory
  - Byte (op code) Fetch, Store (2)
  - Word fetch, Store (2)
- Data Memory
  - Create, Delete segment (2)
  - Push, Pop segment register pair (6)
  - Fetch and Follow (2)
  - Data Access (48)
    - Fetch, Store
    - Byte, Halfword, Word, Doubleword
    - Global, Local, Extra segment based
    - Direct, Autoincrement

VI. Control Flow
- CALL, RET, TRET (3)
- Conditional Branch (21)
- Single-byte interrupt (1)
- Software interrupt (1)
- Sleep, Simulate wakeup (2)
- Enable, Disable stack interrupts (2)
- Enable, Disable arithmetic interrupts (2)
- Wait for interrupt (1)
- Load string mask (1)

VII. Input/Output
- Input byte/word (2)
- Output byte/word (2)
- Push timer (1)
- Copy date-time string (1)

VIII. Multitasking
- Create/Delete task (2)
- Send/Receive/Accept semaphore (3)
- Send/Receive message (2)
- Queue Processing (8)
  - Get/Put
  - Byte/Halfword/Word/Doubleword