A 32 Bit Forth Microprocessor

John R. Hayes
Martin E. Fraeman
Robert L. Williams
Thomas Zaremba

Johns Hopkins University / Applied Physics Laboratory

ABSTRACT

We have developed a simple direct execution architecture for a 32 bit Forth microprocessor. The processor has two instruction types: subroutine call and user defined microcode. The processor's data path was designed so that most Forth primitives can be represented with one microcode instruction that executes in a single cycle. The processor uses a single large, uniform address space (2^{31} words) for program, data, and stack storage. The top portions of the parameter and return stacks are cached in the microprocessor to improve performance while retaining a single data path between memory and the CPU. A Forth outer interpreter that supports inline code expansion was written.

1. Introduction

This paper describes the architecture of a 32 bit microprocessor designed for the direct execution of Forth programs. Two versions of this architecture have been implemented: a prototype built with 4μm Silicon on Sapphire (SOS) and a complete version built with 3μm bulk CMOS. The processor has a large uniform address space and operates on 32 bit quantities. It also has good program execution performance because most Forth primitive operations are executed in one cycle. This single cycle execution and the small number of instruction formats makes this architecture another example of a Reduced Instruction Set Computer (RISC) [Patterson85].

For many years our group has used Forth to program embedded computers, especially for spacecraft. We recently built a bit-slice board level Forth processor [Ballard84] for use in the Hopkins Ultraviolet Telescope (HUT) which was to have flown on the Space Shuttle in March, 1986 (rescheduled to January, 1989). The project described in this paper was undertaken to show that a systems design group could cost effectively develop and use custom VLSI circuits to enhance system capabilities. A single chip Forth processor could replace the 72 in^2 circuit board used for the HUT processor and increase performance by a factor of 5-10 while operating on 32 bit rather than 16 bit numbers. Because of lack of time and budget and because most of the embedded systems we have built are not available for study*, no rigorous program based architecture studies were performed. Consequently, many of our architectural decisions were based on simple experiments, experience, and intuition.

This paper is a summary of our work (so far) on Forth processor architectures. More detailed treatment of the topics discussed in this paper can be found in [Fraeman86], [Hayes86], and [Williams86]. The paper begins by identifying three features of Forth that benefit from hardware support and defining an instruction set to provide this support. Next, our processor's data path and how it implements Forth's primitives is described. The following section discusses the on-chip stack caches. Section 5 briefly describes some special features of the Forth outer interpreter needed to support a direct execution machine. Finally, some results from both implementations of the architecture are described.

---

* It is difficult to profile a program written for a satellite after the satellite has been launched into a 600 mile polar orbit.

2. *Instruction Set Architecture*

Three aspects of Forth can benefit from architectural support. The first is Forth's inner interpreter. In Forth implementations on traditional processors, the inner interpreter emulates the fetch-execute cycle of an abstract Forth machine. This overhead typically uses 35%-50% of the CPU time. Our processor architecture allows most Forth primitives to execute in one instruction and the inner interpreter simply becomes the fetch-execute cycle of the processor.

The second feature of Forth that can take advantage of architectural support is Forth's two stack programming model. Most Forth primitives take operands from one or both stacks, increment or decrement stack pointers, and return a result to one of the stacks. To achieve our goal of executing one Forth primitive per cycle, these stack accesses must also occur in one cycle. Our solution is to cache the tops of the parameter and return stacks on the chip. Our caching algorithm keeps the top four stack elements available in cache and allows the cache to automatically overflow into memory. The stack caching algorithm is discussed in detail in section 4.

Finally, Forth's heavy use of subroutine calls can benefit from architectural support. Good programming practice partitions a Forth program into many short, simple words. Profiles of running Forth programs are consequently dominated by subroutine calls as shown by the execution profiles of three Forth programs in Table 1. In the first two profiles, calls and returns are the most common operation. The third profile of a much smaller program is dominated by a single loop and shows few calls and returns. Call and return each take one cycle to execute in our processor.

The processor has only two instruction formats (see Figure 1) with the most significant bit (msb) of a 32 bit instruction determining the interpretation of the remaining 31 bits. If the msb is zero, the instruction is the address of a subroutine to call. Therefore, a list of addresses that defines a Forth word is also a program to execute that word. This approach is used in many Forth engines including the HUT DEP [Ballard84] and the Novix family of processors [Golden85]. The only disadvantage of this approach is that one half of the address space cannot be used to hold programs. This is less of a problem in 32 bit processors than in 16 bit processors.

If the msb of an instruction is one, the rest of the instruction is microcode that directly controls the data path of the processor. The microcode consists of ten fields that each control a resource in the data path. Almost all of Forth's primitive stack manipulation and arithmetic words can be implemented with a single microcode instruction. The details of the microcode instructions and the data path are discussed more thoroughly in the next section.

Both of the instruction types described above are executed while the next instruction is being fetched. However, some Forth primitives, such as @ and !, disrupt the instruction prefetch and consequently require an extra cycle to fetch the next instruction. Conditional and unconditional branches also need two cycles to execute since the instruction and a 32 bit destination address must be fetched.

3. *Instruction Execution*

All of the elements in the processor's data path are a full 32 bits wide (see Figure 2). Most of the elements communicate over the *Bbus*. A short auxiliary *Abus* is used in calculating the address of the next instruction.

The elements in the data path include parameter and return stack caches, an *ALU*, a one bit shifter, and a temporary Data Latch (*DL*). One input to the *ALU* always comes from the *Bbus* while the other input can come from the top of the parameter stack (*TOS*) or from the *Abus*. The *Abus* connection permits the *ALU* to increment the program counter at a time when the *ALU* would otherwise be idle. The *TOS* connection is used to execute Forth binary operations such as +. A one bit Flag Latch (*FL*) can save a selected *ALU* condition. The *FL* can subsequently be driven onto the *Bbus* or be used to control a conditional

TABLE 1. Primitive Execution Frequencies

MC68010 Metacompilation		HUT DEP Flight Code		1802 Data Acquisition Code	
Primitive	Frequency %	Primitive	Frequency %	Primitive	Frequency %
(;)	13.9	(:)	17.3	+	10.9
(:)	13.6	@	10.9	(arrays)	10.9
?branch	7.8	(constant)	9.4	i	10.5
dup	7.7	(;)	7.5	(loop)	10.2
@	6.3	wait	5.6	c@	8.1
(constant)	5.6	(literal)	5.5	constant	6.1
(variable)	5.4	?branch	4.7	-	4.8
(literal)	4.9	r@ (i)	4.1	(:)	4.4
branch	3.5	swap	3.8	(;)	4.4
swap	2.4	(/loop)	2.8	@	4.0
and	1.9	-1	2.8	(lit8)	3.5
!	1.9	and	2.7	!	3.2
r>	1.9	1	2.6	dup	3.2
>r	1.8	(variable)	1.8	?branch	2.1
+	1.8	not (0=)	1.8	swap	2.1
c!	1.6	drop	1.8	0<	2.0
over	1.6	0<	1.8	j	2.0
c@	1.5	dup	1.7	branch	1.7
1+	1.3	over	1.5	and	1.1
drop	1.3	or	1.4	(lit16)	1.1
1-	1.1	rotate	1.4	r>	1.0
cmove	1.0	=	1.0	>r	1.0
other	10.4	*other*	6.3	*other*	1.5

msb	Argument	Action
0	address	subroutine call
1	control fields	user defined microcode

Figure 1. Instruction Formats

branch. The program counter consists of the Instruction Address Register (*IAR*) and the Address Latch (*AL*). The data path also includes an Instruction Register (*IR*), a path to the external address/data bus (*Port*), the processor status word (*PSW*), and four global User Defined Registers (*UDRs*).

The microcode instructions execute in two steps. In the first step operands are transferred to the *ALU*, a result is temporarily saved in *DL*, and a *FL* condition is latched. The results are forwarded to a destination register during the second step. Tables 2* and 3 describe the microcode fields that control the operations.

The interpretation of the fields in Tables 2 and 3 is straightforward except for

* It is interesting to compare this table with the equivalent table from the first version of the processor [Fraeman86]. In the second version, some small changes were made to the *Shift*, *Cin*, and *Flag* fields to better support multi-precision arithmetic.

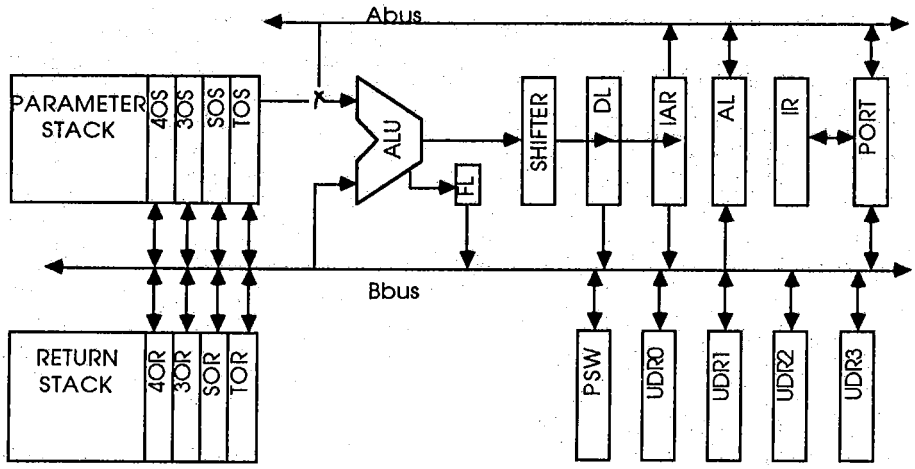

Figure 2. Data Path Block Diagram

the *Stackop* and *Postfetch* fields. The programmer sees the *Stackop* operation occurring 'magically' between step one and step two. Thus, an instruction that accesses the top of the parameter stack in both steps is referring to two different physical registers if the instruction also pops or pushes the stack.

The one bit *Postfetch* field is set when an extra instruction fetch cycle is necessary because a memory load or store operation prevented the normal instruction prefetch. The postfetch cycle is also used to implement conditional branches. In a postfetch cycle the value in *FL* determines the address of the next instruction. If the *FL* is set, the program counter has the address of the next instruction and if the *FL* is cleared, the *IR* has the address. A conditional branch consists of a microcode instruction that performs a test, conditionally sets the *FL* and specifies a postfetch cycle. During execution of the microcode instruction, a 32 bit destination address is fetched from the instruction stream into the *IR* as if it were an instruction. The postfetch cycle will either branch to the location held in the *IR* or continue based on the value of *FL*. Load and store instructions which also require a postfetch cycle must arrange to set *FL* and unconditional branches must clear *FL*.

The basic two step microcode instruction can be summarized in the register transfer notation shown at the top of Table 4 where step one is on the left and step two is on the right. Table 4 also shows how some representative Forth primitives are implemented. The stack operations that push the parameter stack or pop the parameter stack are denoted by ↓P and ↑P respectively. The last entry in the table shows how multiple Forth primitives can be packed into one microcode instruction [Hayes86].

4. Stack Caching

An overflow/underflow mechanism allows a stack to grow larger than the space available in the on-chip memory. The method is similar to an algorithm analyzed by Hasegawa and Shigei [Hasegawa85] which they call Cut-Back-K. When the on-chip memory is full and a stack push occurs, the bottom K words of the on-chip memory are written out to main memory. If the on-chip memory is empty and a stack pop occurs, K words are read in from main memory. This algorithm is not directly applicable to our architecture for two reasons. First, our instruction encoding

TABLE 2. Step One Instruction Fields

Field	Action	Size
Bbus	source register of *Bbus* *TOS, SOS, 3OS, 4OS* *TOR, SOR, 3OR, 4OR* *IAR, PSW, UDR0, UDR1, UDR2, UDR3*	4
Shift	select shifter operation arithmetic shift right none	1
ALUop	*ALU* operation	8
Cin	carry input 0, 1, FL, ¬FL	2
Flag	*Flag* condition *0, Z, N, C, V, NxorV, ¬C+Z, (NxorV)+Z, 1, ¬Z, ¬N, ¬C, Nop, ¬(NxorV), ¬(¬C+Z), ¬((NxorV)+Z)*	4
Xfer	bus transfer *Abus→AL|PORT*,read *Bbus→PORT*,read *Bbus→AL|PORT*,read *Bbus→PORT*,write	2
Stackop	stack operation push parameter stack pop parameter stack push return stack pop return stack pop both stacks push parameter stack, pop return stack pop parameter stack, push return stack nop	3
Total step one bits allocated		**24**

allows access to the top four stack elements, so these elements must always be available in the cache. Second, our implementation of the algorithm uses high priority interrupts to handle stack overflow and underflow, so at least one stack location must be available for use by the interrupt service routine. However, merely by pretending that there are five less locations available in on-chip memory allows us to apply Hasegawa's analysis.

Each stack cache in the current implementations of the architecture consists of sixteen 32 bit words. The choice of sixteen words was dictated almost solely by available chip area. The stack cache can be modeled as an eleven state Markov chain. A pop will cause the system to follow the left arrow (see Figure 3) from its current state to its new state. Similarly, a push will cause a transition to the right. If neither a push nor a pop occurs, the state remains unchanged. There are eleven states in the model because that is the maximum excursion that the top of stack can make within the cache without causing an overflow or underflow. When the cache is in state eleven and a push occurs, the cache overflows and K cached stack words are written to main memory. In Figure 3, K=8, and state four is entered following an overflow. If eight more pushes occur, the cache will overflow again.

TABLE 3. Step Two Instruction Fields

Field	Action	Size
Bsrc	*Bbus* source register *DL, FL, PORT, TOS*	2
Bdest	*Bbus* destination register *TOS, SOS, 3OS, 4OS* *TOR, SOR, 3OR, 4OR* *PSW, PORT, UDR0, UDR1* *UDR2, UDR3,* none	4
Postfetch	execute post-fetch cycle	1
Total step two bits allocated		7
Instruction Word Size		31

TABLE 4. User Defined Microcode for Some Typical Forth Primitives

Primitive	Action, step one	Action, step two
Generic Actions	source op TOS → DL; cc → FL; stackop	source → dest
dup	TOS → DL; ↓P	DL → TOS
over	SOS → DL; ↓P	DL → TOS
>r	TOS → DL; ↑P; ↓R	DL → TOR
r>	TOR → DL; ↑R; ↓P	DL → TOS
1+	TOS + 1 → DL	DL → TOS
0=	TOS → DL; Z → FL	FL → TOS
+	SOS + TOS → DL; ↑P	DL → TOS
<	SOS − TOS → DL; NxorV → FL; ↑P	FL → TOS
(;), exit	TOS → AL |PORT, read; ↑P	
@	TOS → PORT,read; 1 → FL	PORT → TOS; postfetch
!	TOS → PORT,write; 1 → FL; ↑P	TOS → PORT; postfetch
?branch	TOS → DL; $\overline{Z}$ → FL; ↑P <target address>	postfetch
over 0< if	SOS → DL; N → FL <target address>	postfetch

Hasegawa and Shigei's analysis of the Cut-Back-K algorithm assumes that the top of the stack does a random walk, i.e., that the probabilities of a push or a pop in a given instruction are independent of what happened in the previous instruction. The probability of push is also assumed to be equal to the probability of a pop. The analysis found that the expected duration of the random walk the top the stack makes before an overflow or underflow occurs is:

$$D_K = \frac{K\,(N-K)}{1-r} \qquad (1)$$

where

K is the cut back value
N is the number of states + 1
r is the probability that an instruction neither pushes nor pops

D_K is maximized by setting K to N/2

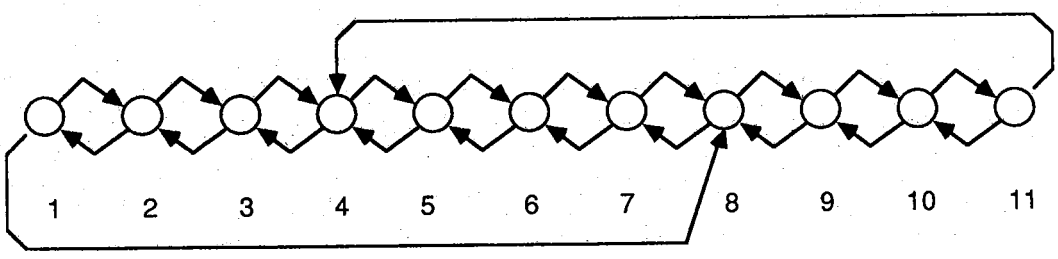

Figure 3. Cut-Back-K Algorithm: K=8

yielding:

$$D_{max} = \frac{N^2}{4(1-F)} \qquad (2)$$

With a cache of sixteen words and the top four stack elements always in the cache, the optimal K is 6. The state diagram in Figure 4 represents this variation of the algorithm. This figure and the previous equation bear out the intuitively appealing notion that intervals between falling off the end of the diagram are maximized by starting at the center of the diagram. Our current chip design uses the K=8 version of Figure 3 instead of the optimal algorithm because an extremely simple VLSI implementation was found for K=8 [Fraeman86].

In practice, the average depth of a Forth stack varies slowly while the actual depth experiences small, rapid variations. The slow variation contributes little to a program's stack caching overhead. However, if the amplitude of the rapid oscillations is sufficiently large, the stack underflow/overflow mechanism will cause thrashing between the cache and main memory. Oscillations that are greater than three quarters of the on-chip cache size will always produce this thrashing. Also, with K=8, oscillations with amplitudes down to one quarter of the on-chip cache size can produce thrashing if the initial stack depth is at an inopportune value.

An experiment was done to characterize the stack depth behavior of a typical Forth program. The inner interpreter of a conventional Forth system was modified to record the current stack depth for each primitive* executed. A trace of the stack depths from the first 1,000,000 primitives executed in the metacompilation benchmark (Table 1) was fed to a simulation of the caching algorithm. The simulation was parameterized in the size of the cache, the number of items initially on the stack, and the Cut-Back-K value. In addition to caches of size 16, 32 word caches were also simulated. Equation 2 above indicates that the length of the random walk is proportional to the square of the number of states in the model, so doubling the size of the cache should reduce the number of stack interrupts by at least a factor of four. For cache sizes of 16, eight different runs were performed with each run having a different number of items initially on the stack ranging from 16 to 24. This allowed observation of the worst and best case performance of the algorithm. For caches with 32 words, sixteen runs were done.

The results are summarized in Table 5. With an on-chip cache size of 16, the worst case performance of the stacks is quite poor, while the best case performance is very good. Doubling the on-chip stack size to 32 reduces the worst case behavior dramatically. This data indicates that stack sizes of 16 are often sufficient but that sizes of 32 are preferable. This single experiment is not conclusive and the performance of the cache running real code

* A primitive is any code word defined in the system ranging from dup to the dictionary look up word (find).

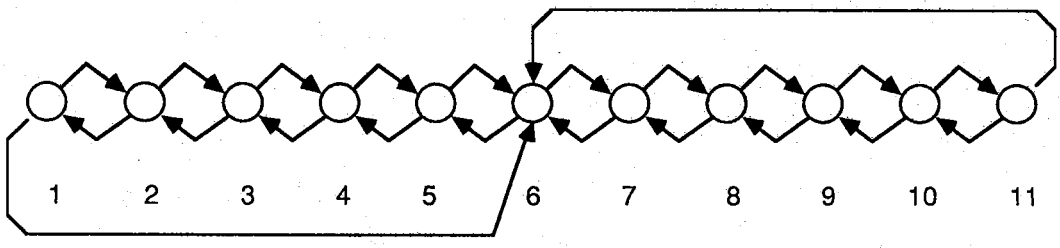

Figure 4. Cut-Back-K Algorithm: K=6 (Optimal)

remains to be seen.

5. A Forth Interpreter for a Forth Chip

A Forth incremental compiler and interpreter have been written for our processor [Hayes86]. Surprisingly, an interpreter for a conventional microprocessor needed only slight modification to work on our chip. An unusual dictionary structure was added to support the direct execution features of the processor.

Most conventional Forth systems use either direct or indirect threaded code. When a primitive such as **dup** is used in a colon definition, the primitive is referenced either via one level of indirection (direct threaded code, see Figure 5) or two levels of indirection (indirect threaded code). With direct execution the actual object code for the primitive is used in the colon definition (see Figure 6). To accommodate this inline code expansion each dictionary entry has a code length field. This field specifies the number of words in the code field that should be copied when expanding the definition inline. Non-inline words will have a code length of zero.

By default most words are not brought inline. Most defining words set the code length to zero when the dictionary header is created. However, if a definition is followed by the word **inline**:

```
: 2dup over over ; inline
```

that definition will always be expanded inline. **Inline** works by computing the length of the most recently defined word (less one for exit) and storing the result in its code length field. Currently, the defining words **constant** and **variable** always create inline words for efficiency. Consequently, constants and variables behave exactly like literals.

6. Results

Two versions of our processor have been built. A prototype chip was implemented in 4μm Silicon on Sapphire (SOS) CMOS. SOS was chosen because of its radiation tolerance in space environments [Williams86]. Fully static design principles were followed so that the chip could be used reliably as a component in a spacecraft.

When the prototype chips were received, we discovered that a design rule violation had disastrously affected yield. However, enough partially functional chips were found to verify the correctness of the design. One chip worked well enough, albeit at a low clock rate, to run an interactive Forth interpreter and incremental compiler.

Following this mixed success, we have reimplemented the architecture in bulk CMOS. The architecture was enhanced slightly to better support multi-precision arithmetic. Although the architecture remained fundamentally the same, the differences between SOS and bulk CMOS dictated an entire redesign at the circuit level. The bulk CMOS design was initially fabricated with 3μm feature sizes. However, since we used scalable design rules, we will be able to fabricate the design

TABLE 5. Stack Interrupt Behavior

Stack Interrupts per 1,000,000 Primitives Executed				
Algorithm	Parameter Stack		Return Stack	
	Best	Worst	Best	Worst
size=16, K=8	6	28366	1019	4949
size=16, K=6	2	4831	751	2236
size=32, K=16	0	1	0	315
size=32, K=14	0	1	0	4

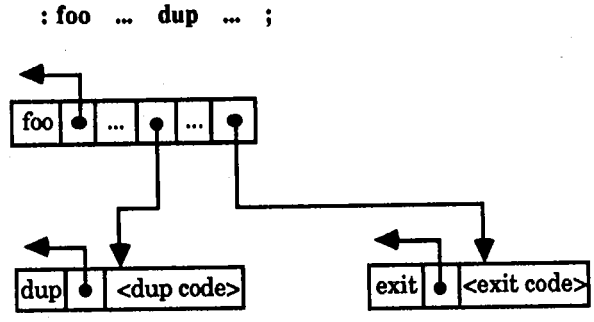

Figure 5. Direct Threaded Use of Primitive

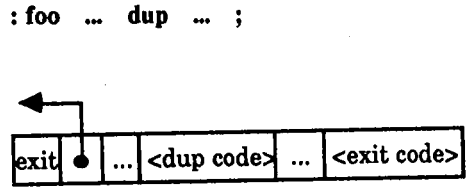

Figure 6. Direct Execution of Primitive

with 1.2μm features in the future yielding a 2-4 improvement in performance. We expect to receive 3μm chips in June of 1987.

7. Acknowledgements

The authors wish to express their gratitude to Dr. R. P. Rich and Dr. L. C. Kohlenstein for their support and encouragement of this work. We also wish to thank R. E. Jenkins for his assistance in obtaining access to MOSIS and express our appreciation to the USC/ISI MOSIS service for fabricating the prototype circuits. This work was done under Navy contract N00039-87-C-5301.

8. References

[Ballard84] Ballard, B. "FORTH Direct Execution Processors in the Hopkins Ultraviolet Telescope", *Journal of Forth Applications and Research, 2,1* 1984, pp. 34-47.

[Fraeman86] Fraeman, M.E., Hayes, J.R., Williams, R.L., Zaremba, T. "A 32 Bit Architecture For Direct

Execution of Forth", *Proc. of the Eighth FORML Conference*, 1986.

[Golden85] Golden, J., Moore, C.H., Brodie, L., "Fast Processor Chip Takes Its Instructions Directly From Forth", *Electronic Design*, March 21, 1985, pp. 127-138.

[Hasegawa85] Hasegawa, M., Shigei, Y. "High-Speed Top-of-Stack Scheme for VLSI Processor: a Management Algorithm and its Analysis", *Proc. of the 12th Annual International Symposium on Computer Architecture*, 1985, pp. 48-54.

[Hayes86] Hayes, J.R. "An Interpreter and Object Code Optimizer for a 32 Bit Forth Chip", *Proc. of the Eighth FORML Conference*, 1986.

[Patterson85] Patterson, D.A. "Reduced Instruction Set Computers", *Communications of the ACM 28,1*, January, 1985, pp. 8-21.

[Williams86] Williams, R.L., Fraeman, M.E., Hayes, J.R., Zaremba, T. "The Development of a VLSI Forth Microprocessor", *Proc. of the Eighth FORML Conference*, 1986.