AN ARCHITECTURE FOR THE DIRECT EXECUTION OF THE FORTH PROGRAMMING LANGUAGE

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 can directly access a linear address space of over 4 gigawords. Two instruction types are defined; a subroutine call, and a user defined microcode instruction. On-chip stack caches allow most Forth primitives to execute in a single cycle.

1. Introduction

This paper describes the architecture of a 32 bit microprocessor designed for the direct execution of Forth programs. 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 architecture is another example of a Reduced Instruction Set Computer (RISC)[7]. A prototype of the architecture has been implemented as an integrated circuit in CMOS/SOS technology with 4µm feature sizes.

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[1] for use in the Hopkins Ultraviolet Telescope (HUT) which was to have flown on the Space Shuttle in March, 1986 (rescheduled to June, 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.

The paper begins with a brief description of Forth, particularly how the language can be tailored to a specific application. Then three features of Forth that would benefit from hardware support are identified, and an instruction set to provide this support is defined. Next, our processor's data path and how it implements Forth's primitives is described. The next two sections discuss the on-chip stack cache and a peephole optimizer. Finally, some results from our prototype chip are described.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

2. Why Forth?

Why use Forth in the first place? Our design group has found Forth extremely useful in developing embedded systems. The hardware environment of an embedded system is usually inadequate for supporting its own software development. Consequently, the traditional edit-compile-debug cycle is worse because of an extra step: edit-cross compile-download-debug. The software debug phase is typically performed with logic analyzers and in-circuit-emulators. This painful process is avoided by running a Forth interpreter and incremental compiler on the target embedded system. This is made possible by the small size of a Forth programming support system.

Forth's extremely simple syntax and trivial parser allow it to run in impoverished hardware environments. Forth's lexical properties are also simple. Forth subroutines, called words, are delimited by spaces. The words themselves can consist of any characters other than the delimiter. This simplicity keeps the interpreter small allowing full featured Forth systems to fit comfortably in as little as 8kbytes of memory.

Programming in Forth consists of defining new words in terms of existing words. The new word is incrementally compiled and can be invoked interactively by the programmer. Thus, the usual benefits of interpreted languages are reaped, especially simplified testing and a resulting higher confidence in program correctness.

Adding new words to the system is one way in which Forth is extensible. However, more interesting forms of extensibility, made possible by Forth's simple syntax, allow the language to be adapted for a specific application. Forth

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

© 1987 ACM 0-89791-238-1/87/1000-0042 $00.75

has two types of macros which are especially useful in this regard. An example of how one type of macro is used to extend the language occurs in the Forth interpreter. The interpreter must be able to recognize and discard Forth comments which consist of any text surrounded by parentheses. A macro named '(' is defined (see Figure 1) that scans the input stream until a closing parenthesis is found and then discards the text. Other uses of this type of macro, called a compiling word, can be found in the interpreter. For instance, Forth control structure words (**begin**, **if**, **while**, etc.) are implemented as compiling words. The second type of macro, called a defining word[4], is used to create new families of words where all the words in a given family share similar properties. These macro capabilities allow the creation of simple languages that are more applicable to a specific problem than a conventional general purpose language.

A second example shows how Forth can be molded to suit an application. The example consists of an assembler for the PDP-11. The traditional way to implement an assembler is to write a program that reads assembly language statements from a data file, translates the statements into object code, and writes the results into an output file. The Forth approach to this problem is considerably different. A different syntax is used for the assembly language statements (see Figure 2). For each possible PDP-11 instruction (mov, inc, etc.) a Forth word is defined to generate the appropriate piece of object code for that instruction. Similarly, for each PDP-11 addressing mode a Forth word is written which modifies the most recently assembled instruction to use that addressing mode. The assembly language statements have become a Forth program that assembles itself. The data has become the program! The point is not that this is the best way to write an assembler but that Forth's flexibility allows the language to be adapted to solve the problem at hand.

3. Instruction Set Architecture

Three features of Forth can benefit from architectural support. The first feature is the fetch-execute cycle of Forth's virtual machine that is emulated in software on conventional processors. Forth systems are usually implemented using threaded code [2,8]. A definition of a Forth word consists of a list of addresses pointing to the definitions of its component words. When a word is executed, a tiny

```
( A macro is defined to ignore Forth comments          )
( surrounded by parentheses. The colon begins the      )
( definition followed by the name of the definition.   )
( CLOSEPAREN, the predefined ASCII value for a         )
( closing parenthesis, is passed to a routine named    )
( word which scans the input stream for the closing    )
( parenthesis. Word returns the address of a buffer    )
( holding the text which is thrown away by drop. The   )
( semicolon ends the definition and immediate          )
( flags it as a macro.                                 )

hex 29 constant CLOSEPAREN
: (   CLOSEPAREN word drop ; immediate
```

Figure 1. Definition of '(' Macro

DEC notation		Forth notation	
mov	(r0)+,r1	mov	r0)+ r1
inc	(r3)	inc	r3)
add	r2,-(r1)	add	r2 r1 -(

Figure 2. PDP-11 Assembler Example

program known as the inner interpreter (not to be confused with the outer interactive interpreter) traces the addresses, nesting down if necessary, until a primitive word defined in assembly language is found. Control is then transferred to the primitive. The inner interpreter consumes 35%-50% of the CPU time in Forth systems running on conventional processors. Consequently, hardware support for the inner interpreter is vital. The simplicity of Forth's primitives makes this goal easy to achieve. Our architecture can represent most Forth primitives in one instruction and the inner interpreter is simply the fetch-execute cycle of the processor.

The second feature that benefits from architectural support is Forth's two stack programming model. One stack, called the return stack, holds control flow information such as subroutine return addresses. The other stack, called the parameter stack, is used for computation and argument passing. Most of the primitive words in the Forth kernel operate on this stack. For example, **dup** is a word that duplicates the value on the top of the stack. Another example is + which pops two values from the stack, adds them together, and pushes the sum on the stack. Therefore, efficient access to two stacks is desirable. Our processor supports this feature with two on-chip stack caches (see sections 4 and 5).

Thirdly, Forth makes heavy use of subroutines, so a very fast call instruction is important. The single line of code needed to implement the '(' macro in Figure 1 is typical of Forth programs. It is considered good programming practice to partition a program into many short, simple words. Experiment shows that execution profiles of Forth programs are dominated by subroutine calls. Table 1 shows the dynamic execution frequencies of Forth primitives measured on several Forth implementations running a variety of programs. The first two profiles do show high frequencies of calls, (:), and returns, (;)*. The third profile of a much smaller program is dominated by a single loop and so shows fewer calls and returns.

The processor has only two instruction formats (see Figure 3). The most significant bit (msb) of the 32 bit instructions distinguishes the two formats. If the msb is zero, the remaining 31 bits are the address of a subroutine to call. So, a Forth definition consisting of a list of pointers into the lower 2^{31} words of address space is a program that executes that definition. Consequently, all programs must reside in the bottom half of the address space.

If the msb of an instruction is one, the remainder 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.

Microcode instructions that disrupt the prefetching of instructions by doing loads or stores require two cycles to execute. All other instructions, including call, execute in only one cycle. Two cycle conditional branch and unconditional branch instructions are implemented with a microcode instruction that reads a 32 bit destination address

* Note that there are a different number of calls (:) and returns (;) in the middle column. In this Forth implementation **(variable)** and **(constant)** are called as subroutines and have an embedded return.

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 3. Instruction Formats

from the instruction stream. In fact the subroutine call instruction could have been implemented similarly so that only one instruction format would have been necessary at the cost of doubling the execution time of calls. The profiling data could be used to guide the selection of other instruction types for future versions of the architecture. **Literal**, **?branch** (conditional branch), and **branch** (unconditional branch) are obvious candidates.

4. Processor Data Path

The processor has an internal 32 bit wide data path. The data path resources include several special purpose registers, an arithmetic logic unit (ALU), and two stack memories. These resources communicate over a single common bus and a short auxiliary bus. There are also several unidirectional local communication paths between specific resources. A block diagram of the data path is shown in Figure 4.

Most of the data path resources communicate over the main *Bbus*. The *Abus* is used primarily for program counter address calculations. The port is a multiplexer that connects either the *Abus* or the *Bbus* to the external address/data bus. In general, the *Abus* connection is used for instruction fetching and the *Bbus* connection is used for memory loads and stores.

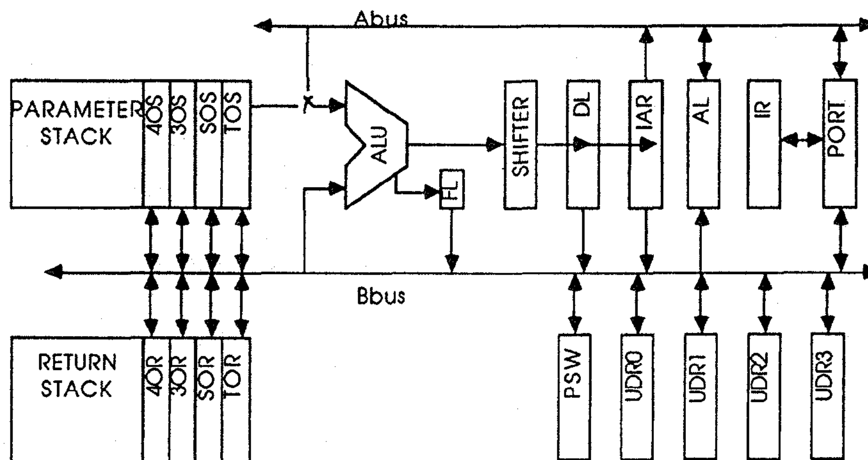

Figure 4. Data Path Block Diagram

Two on-chip stack memories are provided to support Forth's two stack programming model. The current instruction encoding permits access to the top four elements of either stack. A mechanism is provided to allow the stacks to overflow into main memory using the stack caching algorithm described below. Four global User Defined Registers (*UDR*s) are also included in the architecture. Although these registers are unnecessary to support Forth, they are useful. For instance, two *UDR*s are currently dedicated to the stack caching algorithm.

The ALU performs the expected arithmetic and logic operations. One input to the ALU is attached to the *Bbus* and the other is connected to a multiplexer. The multiplexer can connect the top of the parameter stack or the *Abus* to the ALU. The stack connection is used to implement Forth's stack operators. The Abus connection is used to allow the ALU to increment the program counter, which consists of the Instruction Address Register (*IAR*) and the Address Latch (*AL*), during a time when the ALU would otherwise be idle. A single bit temporary flag latch (*FL*) holds a selected ALU condition. The ALU output goes into a single bit shifter, then to a temporary Data Latch (*DL*).

The programmer views the microcode instructions as executing in two phases. In the first phase operands are fetched from registers, delivered to the ALU, and results are computed into the *DL* and *FL* latches. During the second phase a result is written into a destination register. Tables 2 and 3 show the microcode instruction fields used to control the data path during the two phases.

TABLE 2. Phase 1 Instruction Fields

Field	Action	Size
Bbus	source register of *Bbus* *TOS, SOS, 3OS, 4OS* *TOR, SOR, 3OR, 4OR* *UDR0, UDR1, UDR2, UDR3,* *IAR, PSW*	4
Shift	select shifter operation logical shift left logical shift right arithmetic shift right none	2
ALUop	*ALU* operation	8
Cin	carry input	1
Flag	*Flag* condition $0, Z, N, C, V, N xor V, \neg C{+}Z, (N xor V){+}Z,$ $1, \neg Z, \neg N, \neg C, \neg V, \neg(N xor V),$ $\neg(\neg C{+}Z), \neg((N xor V){+}Z)$	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 phase 1 bits allocated		**24**

TABLE 3. Phase 2 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* *UDR0, UDR1, UDR2, UDR3,* *PSW, PORT,* none	4
Postfetch	execute post-fetch cycle	1
Total phase 2 bits allocated		**7**
Instruction Word Size		**31**

The interpretation of the instruction fields is generally straightforward[3] except for the *Stackop* and *Postfetch* fields. The programmer sees the *Stackop* operation occurring 'magically' between phase 1 and phase 2. Thus, a microinstruction that accesses the top of the parameter stack in both phases is referring to two different physical registers if the microinstruction 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 instruction register (*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 phase microinstruction can be summarized in a register transfer notation shown at the top of Table 4 where phase 1 is on the left and phase 2 is on the right. Table 4 also shows how some representative Forth primitives are implemented. The stack operations that push or pop the parameter stack are denoted by ↓P and ↑P respectively.

5. Stack Caching

An overflow/underflow mechanism allows the stack to grow larger than the space available in the on-chip memory. The method is based on an algorithm analyzed by Hasegawa and Shigei[5] 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, words are read in from main memory. This algorithm is not directly applicable to our architecture for two reasons. First, our instruction encoding 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.

TABLE 4. User Defined Microcode for Some Typical Forth Primitives

Primitive	Action, phase 1	Action, phase 2
Generic Actions	source op TOS → DL; cc → FL; stackop	source → dest
dup	TOS → DL; ↓P	DL → TOS
over	SOS → DL; ↓P	DL → TOS
+	SOS + TOS → DL; ↑P	DL → TOS
0=	TOS → DL; Z → FL	FL → TOS
@, load	TOS → PORT,read; 1 → FL	PORT → TOS; postfetch
!, store	TOS → PORT,write; ↑P; 1 → FL	TOS → PORT; postfetch
?branch, if	TOS → DL; $\overline{Z}$ → FL; ↑P <target address>	postfetch

Each stack cache in the current implementation of the architecture consists of sixteen 32 bit words. The choice of sixteen words was dictated almost solely by available chip area (see Figure 7). 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 5) 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 5, K=8, and state four is entered following an overflow. If eight more pushes occur, the cache will overflow again.

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 a 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 yielding:

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

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 6 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 5 instead of the optimal K=6 algorithm because an extremely simple VLSI implementation was found for K=8[3].

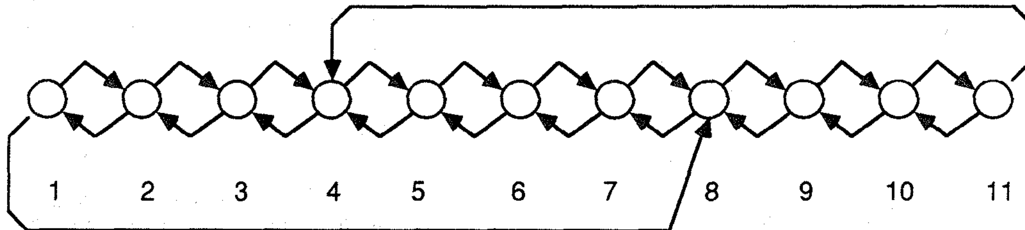

Figure 5. Cut-Back-K Algorithm: K=8

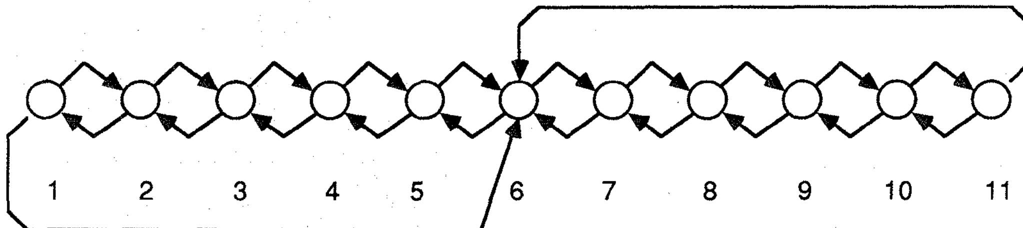

Figure 6. Cut-Back-K Algorithm: K=6 (Optimal)

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. 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.

TABLE 5. Stack Interrupt Behavior

Algorithm	Stack Interrupts per 1,000,000 Primitives Executed			
	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

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 remains to be seen.

6. Object Code Improvement

The processor's data path is actually more general than the execution model needed for Forth. For example, a Forth binary operation takes two operands from the stack, performs a calculation on them, and pushes the result onto the stack. In a generic binary operation using the processor's data path, one operand comes from the top of the stack but the other operand can come from almost any register. In addition, the result can be sent almost anywhere and most combinations of pushing and/or popping both stacks is possible. Consequently, it is often possible to execute multiple Forth primitives with one microinstruction. For example use of **dup**, **over**, etc. for positioning operands is entirely overhead. If the following instruction consumes the new top of stack value, the **dup** (or **over**, etc.) can in many cases be combined with the following instruction.

A peephole optimizer was implemented as part of the metacompiler for our processor to perform simple instruction compactions[6]. A Forth metacompiler is a program that produces stand-alone object code from Forth source code that had previously been interpreted. Since this stand-alone code represents a completed application program, it is worth while to go to the trouble of generating higher quality code. Therefore, the optimizer is part of the metacompiler and is not used in the ordinary Forth interactive incremental compilation environment.

The primary two goals of the peephole optimizer were to reduce the execution time and volume of object code. However, a third important capability emerged as the optimizer was developed. By combining an instruction that does a push with an instruction that does a pop so that the push and pop cancel, the resulting object code generates fewer stack overflow/underflow interrupts. Table 6 shows a realistic example of code compaction. A code sequence that takes four cycles to execute has been converted to one that takes two cycles and a potentially costly push has been avoided.

TABLE 6. Code Compaction

Primitive	Action, phase 1	Action, phase 2
	original code sequence	
over **0=** **if**	SOS → DL; ↓P TOS → DL; Z → FL TOS → DL; $\overline{Z}$ → FL; ↑P <target address>	DL → TOS FL → TOS postfetch
	compaction of **over** and **0=**	
<**over 0=**> **if**	SOS → DL; Z → FL; ↓P TOS → DL; $\overline{Z}$ → FL; ↑P <target address>	FL → TOS postfetch
	compaction with **if** test	
<**over 0= if**>	SOS → DL; Z → FL <target address>	postfetch

7. Results

A prototype chip was implemented in MOSIS's (discontinued) 4μm CMOS/SOS process. SOS was chosen because of its radiation tolerance in space environments[11]. Fully static design principles were followed so that the chip could be used reliably as a component in a spacecraft. The design used a 7.9 x 9.2 mm MOSIS standard frame.

The logic design, layout, and simulation of the 18,000 transistor prototype took approximately 9 man-months after the architecture was specified. The architecture took 5 man-months to design and another 2.5 man-months were spent porting CAD tools[9,10] to our Unix work station. The simplicity of the instruction set and the care that went into the design of the architecture helped produce the simple and clean layout shown in Figure 7. Very little instruction decoding was necessary, and the control logic of the chip occupies less than 5% of its area. The floor plan is dominated by the two 16 word stack caches.

When the prototype chips were received from MOSIS, 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. These partially functional chips executed simple diagnostic programs at speeds up to 1.5MHz. One chip worked well enough, albeit at a low clock rate, to run an interactive Forth interpreter and incremental compiler.

Despite the poor yield, we feel that the project was a success. Working on shoestring budget, we have created a high performance 32 bit architecture that directly executes Forth. The initial results were sufficiently encouraging that we have reimplemented the architecture in MOSIS's scalable CMOS process. The design will be fabricated with 3μm feature sizes in the summer of 1987 and should execute one Forth primitive every 300ns.

8. Acknowledgments

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 N00024-85-C-5301.

9. References

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

2. Bell, J.R. "Threaded Code", *Communications of the ACM 16,6* , June, 1973, pp. 370-372.

3. 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.

4. Harris, K. "Forth Extensibility: Or How to Write a Compiler in Twenty-Five Words Or Less", *BYTE 5,8*, August, 1980, pp. 164-184.

5. 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.

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

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

8. Ritter, T., Walker, G. "Varieties of Threaded Code for Language Implementation", *BYTE 5,9*, September, 1980, pp. 206-227.

9. University of California, Berkeley, "1983 VLSI Tools: Selected Works by the Original Artists", Report No. UCB/CSD 83/115, March, 1983.

10. University of Washington/Northwest VLSI Consortium, "UW/NW VLSI Release 3.0".

11. 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.

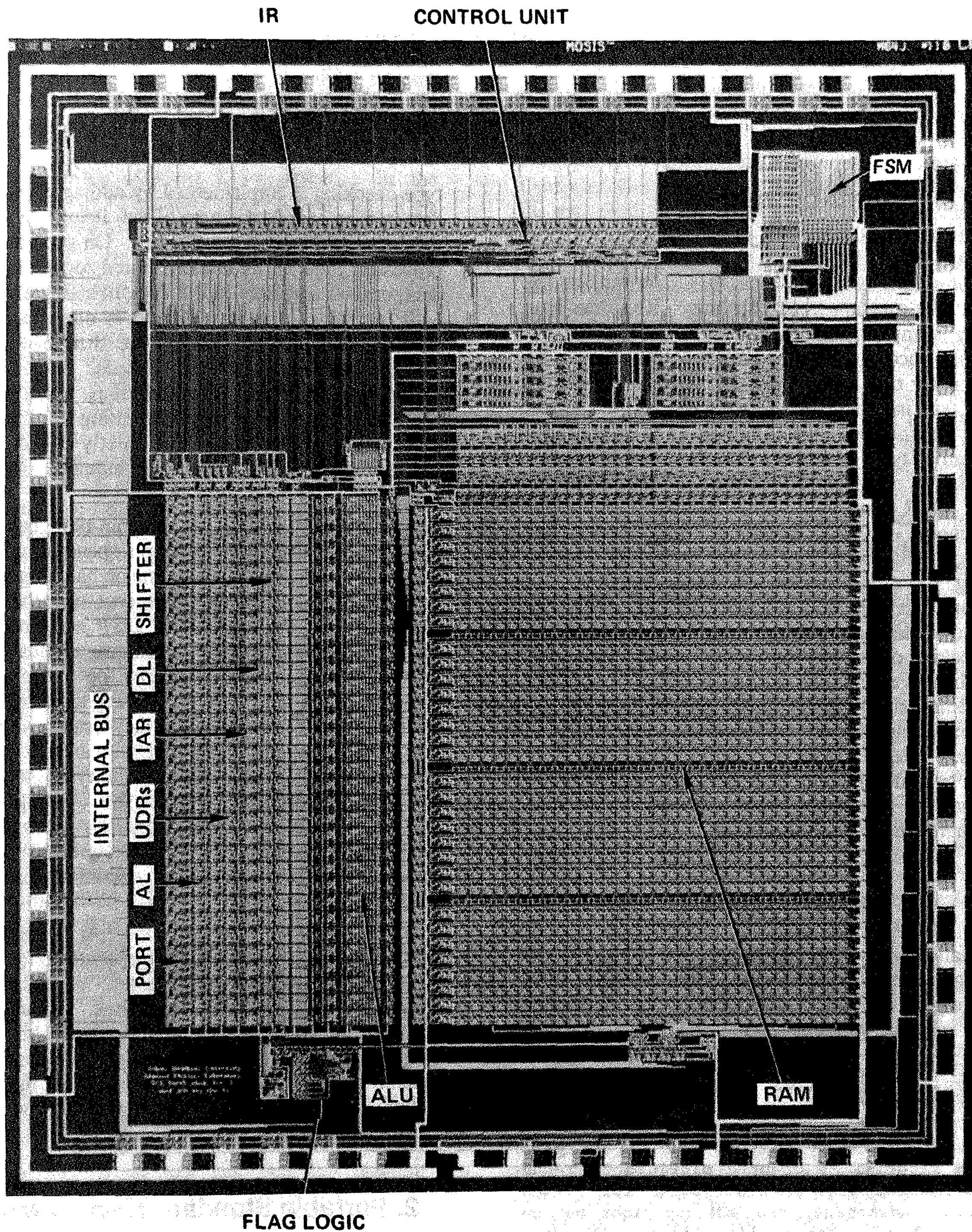

Figure 7. Photograph of Prototype Chip