The Architecture of the SC32 Forth Engine

John Hayes
Susan Lee
Johns Hopkins University
Applied Physics Laboratory

Abstract

We have designed a 32 bit microprocessor that directly executes the Forth programming language. This is the third in a series of Forth Reduced Instruction Set Computers (FRISCs). The SC32 Forth engine improves on our previous designs in its more efficient load, store, literal, and branching instructions; better support of multiplication and division; and a better approach to stack caching. The processor's instruction set consists of eight instruction types in three formats. The 32 bit wide data path contains two stack caches. 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. The processor spends less than 1% of its time managing the stack caches on typical Forth programs.

1. Introduction

This paper documents the third in a series of Forth Reduced Instruction Set Computer (FRISC) processor architectures. Many aspects of the Forth programming language are directly supported in the SC32 Forth engine architecture. The architecture can execute most Forth primitives in a single cycle. All resources and data paths in the processor are 32 bits wide and external memory is addressed as 32 bit words.

We gained much experience and insight into Forth oriented architectures in the design of FRISC 1 and 2. **[FRAE86]**, **[HAYE86]** and **[WILL86]** Many elements of the earlier architectures are clearly visible in the SC32. (The SC32 was originally called FRISC 3. **[HAYE88]**) The SC32 improves on our previous designs in its more efficient load, store, literal, and branching instructions; better support of multiplication and division; and a better approach to stack caching. The processor has been licensed to a commercial venture (Silicon Composers Inc., Palo Alto, CA).

The SC32 instruction set was explicitly designed to implement Forth. Each instruction executes in one cycle except for loads and stores which take two. Most of Forth's primitive operations can be represented with one SC32 instruction. For example, Forth's binary arithmetic operations such as + and - are single instructions and execute in one cycle. Similarly, binary logic operations like **and**, **or**, and **xor** execute in one cycle. In fact, any possible Boolean function of two variables is possible. A rich set of binary comparison instructions (e.g. **<**, **>**, **<>**, **u<**, **<=**, etc.) are present. The SC32 also has many unary comparison and arithmetic instructions (e.g. **1+**, **1-**, etc. and **0<**, **0=**, **0>**, etc.)

An SC32 instruction can access the top four items on either the parameter stack or the return stack. This allows the single cycle implementation of the usual stack manipulation operators (e.g. **drop**, **over**, **>r**, **r>**, **r@**, **nip**, etc.) Access to the top four return stack locations provides easy use of two sets of **do...loop** indices. Access to the top four parameter stack locations allows a single cycle (Forth-83) **2 pick** or **3 pick** in addition to **over** and **dup**.

The SC32 instruction set is more general than Forth's pure stack virtual machine. As a result, sequences of multiple Forth primitives can frequently be implemented with one instruction. For example, arithmetic and comparison operators can often be combined with preceding stack manipulation operations (e.g. **over +**, **r> 0= ,over over =**, etc.) This capability is especially useful with the SC32's load and store instructions. The load and store instructions provide a single, powerful addressing mode that covers the most common array and data structure access operations. So, fetching a variable at location 134 (e.g. **134 @**) or fetching the eighth cell of a data structure (e.g., **over 8 + @**) can be done with one instruction. Forth's vanilla **@** operator is simply a special case of the general purpose load instruction. (A Forth **!** actually takes two instructions: one to do the store and a **drop** to clean up the stack. The extra **drop** can frequently be folded into a following instruction.)

The SC32 has single cycle call, branch, and conditional branch instructions. The call and branch instructions directly implement Forth's colon nesting operation and **branch** operation. **?branch** is implemented with two instructions: an instruction to test the value on top of the parameter stack followed by a conditional branch. The test can frequently be combined with preceding operations (e.g. **0< if**, **over > while**, **dup 0= until**, etc.) resulting in two cycle test-and-branch operations. Similarly, Forth-83's **loop** and **+loop** can be done in two cycles.

The SC32 also has fast literal, quick return, and multiply and divide step instructions. The fast literal allows a 16 bit literal value between 0 and 65535 to be pushed on the stack in one cycle. The quick return provides a way for many returns to be done in zero time. The multiply and divide steps can be used to create efficient multiply and divide operations.

One of the most important considerations in the design of a Forth processor is delivering stack operands to the processor. A consequence of executing one instruction every cycle is the need to fetch a new instruction every cycle. This leaves no spare bandwidth in the processor-to-memory port for fetching stack data. Our solution is stack caching: the top portion of each stack is buffered on chip. The remainder of each stack is in the same memory as instructions and data. The stack cache hardware gives the programmer the illusion of having arbitrarily large on-chip stacks.

As the stack moves up and down within the on-chip stack cache, the cache occasionally overflows or underflows. On overflow, instruction execution is suspended for two cycles while a value is moved from the stack cache to main memory. Underflow is handled similarly. The overhead of managing the stack cache is small: less than 1% of the processor's time is spent on cache management for typical Forth programs. This is a small price to pay for the advantages of the stack caches. Since the stacks are kept in the same address space as instructions and data, only one address/data bus is needed to access them. This results in a small 84 pin package. This also allows a stack to potentially grow as large as the address space of the processor (2^{32} or 4,294,967,295 cells). Finally, since instructions, data, and stacks, are in the same address space, they can all be kept in the same memory chips.

The remainder of this paper describes the architecture of the SC32. Sufficient detail is provided to enable the reader to understand the workings of the processor and to be able to program it. The following section describes the microarchitecture of the data path. Section 3 describes how the instruction set is used to control the data path. The next section describes how instructions execute and how the instruction set implements Forth. Section 5 discusses the stack caches. Section 6 describes some aspects of the processor's external interface. Section 7 discusses the rationale for some of our architectural decisions. Finally, section 8 gives some results of the implementation and first fabrication run.

2. Micro Architecture

Forth programs use two stacks: the parameter stack which is used for passing arguments to functions and a control flow stack which primarily holds subroutine return addresses and is consequently called the return stack. Most Forth primitives take operands from one or both stacks, push or pop the stacks, and return a result to one of the stacks. To achieve the execution of one Forth primitive per cycle, these stack accesses must occur in one cycle. This is supported by two stack caches in the data path.

Forth's heavy use of subroutines is also directly supported. The instruction set and the arrangement of the data path allows subroutines to be called in one cycle and many returns to occur in zero cycles.

Those elements of the data path visible to the programmer and their connectivity are shown in Figure 1. The parameter and return stack caches each consist of sixteen 32 bit registers. The caches each have two read ports and one write port. The instruction set allows access to the top four locations of either stack. Cache overflow and underflow are handled transparently to the program (see Section 5).

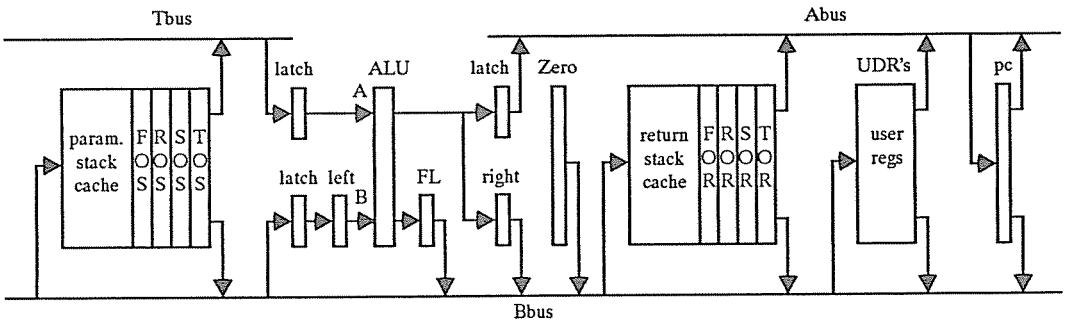

Figure 1. Data Path

In addition to the stack caches there are four global utility registers (called *UDR*s for historical reasons). Two of these registers are dedicated to the stack caching algorithm but the other two may be used as a system designer sees fit. For instance they could be used to implement an additional stack or a frame pointer for a traditional language such as C.

The ALU provides the expected logic and arithmetic functions. A single bit left shifter on the input side of the ALU and single bit right shifter on the output are available for multiplication and division steps. A single condition code flag (*FL*) is provided. The flag can be loaded with any one of sixteen ALU conditions or the shift out bit from one of the shifters. Subsequently, the flag can control a conditional branch, be fed into the ALU's carry input for doing multiprecision arithmetic, or be read onto a bus yielding a 32 bit 0 or -1 truth value.

Zero is a read only register that always returns the value zero. This register is useful for constructing literals and addresses for loads and stores (see Section 3). *PC* is a program counter. There is also a processor status word (*PSW* not shown in Figure 1) that contains the state of the interrupt system and the stack caches.

Three global busses provide communication between the resources described above. At the beginning of the execution of an instruction, *Bbus* delivers an operand to the ALU from a register resource (stacks, utility registers, zero register, etc.). The other ALU operand arrives on the *Tbus* and is always either the top of the parameter stack (*TOS*) or a literal value from the instruction word. After the ALU operates, the result is sent to a destination register via the *Bbus*. The *Bbus* is connected to the off-chip data bus when doing load or store instructions.

The *Abus* addresses the external world. Normally, the program counter is driven onto the *Abus* to fetch the next instruction. The top of the return stack (*TOR*) can also drive the *Abus* so that a return from a subroutine call can occur concurrently with execution of many instructions. During load/store instructions the ALU is used to calculate an address which is subsequently driven onto the *Abus*. The *UDR*s may also drive the *Abus* during stack cache management operations. Not shown in the figure is a path from the instruction register to the *Abus* for doing branch or call instructions (see below).

3. *Instruction Set Architecture*

The SC32 instruction set consists of eight instruction types. There are three control flow instructions, four load/store instructions, and a microcode instruction. All SC32 instructions are 32 bits wide. Each of these three instruction categories are reflected in the three instruction formats shown in Table 1. The three most significant bits (msbs) of the instruction determine its type and the interpretation of the remaining 29 bits.

control flow	Type:3	Address:29				
load/store	Type:3	Next:1	R1:4	R2:4	Stack:4	Offset:16
micro	Type:3	Next:1	R1:4	R2:4	Stack:4	ALU:16

Table 1. SC32 Instruction Format

The control flow instructions are call, branch, and conditional branch. The destination is an absolute address embedded in the instruction. The conditional branch will be taken if the flag (set by a previous instruction) is 0.

The upper sixteen bits of the load/store and micro instructions have the same format. In both formats, the R1 field selects a source register, R2 selects a destination register, and Stack selects any combination of pushing or popping the parameter and return stacks. The Next field

Field	Action	Size
Next	next instruction address *PC* *TOR*	1
R1	source register of *Bbus* *TOS, SOS, ROS, FOS* *TOR, SOR, ROR, FOR* *UDR0, UDR1, UDR2, UDR3* *PC, PSW, Zero*	4
R2	Bbus destination register (source during stores) *TOS, SOS, ROS, FOS* *TOR, SOR, ROR, FOR* *UDR0, UDR1, UDR2, UDR3* *PC, PSW, Zero*	4
Stack	stack operation push parameter stack pop parameter stack push return stack pop return stack pop both stacks push both stacks push parameter stack, pop return stack pop parameter stack, push return stack nop	4

Table 2. Common Instruction Encoding

determines whether the incremented program counter or the top of the return stack is used to provide the address of the next instruction. The common instruction encoding is shown in Table 2.

The four load/store instructions are load, store, load address low (lal), and load address high (lah). In these instructions, the ALU operation is always addition performed on R1 and the unsigned Offset embedded in the low 16 bits of the instruction. A register transfer level notation summarizes the operation of these instructions in Table 3. The * in Table 3 denotes an address computation so, for a load instruction, R1+Offset is the address of data to be loaded into R2. A single addressing mode, register indirect plus offset, is provided. Degenerate cases of this addressing mode yield other useful modes. Setting the offset to zero produces a register indirect mode. Setting R1 to the *Zero* register allows absolute addressing within the low 64kwords of address space.

load:	*(R_1 + Offset) → R_2
store:	*(R_1 + Offset) ← R_2
load address low (lal):	R_1 + Offset → R_2
load address high (lah):	R_1 + Offset ·2^{16}→ R_2

Table 3. Load/Store Instructions

The load address instructions are degenerate loads in that an address is computed but no data is fetched. Instead the address is saved in R2. The lah instruction is similar to lal except that the offset is shifted left sixteen bits before being added to R1. The primary use for these two instructions is the construction of literals. Sixteen bit literals can be produced by a single lal instruction. Any 32 bit literal can be constructed by an lah followed by an lal.

Forth Code	SC32 Instruction
`@`	*(TOS + 0) → TOS
`!`	*(TOS + 0) ← TOS; ↑P ↑P
`avariable @`	*(Zero + 327) → TOS; ↓P
`over @`	*(SOS + 0) → TOS; ↓P
`over anarray + @`	*(SOS + 1234) → TOS; ↓P
`dup 9 + @`	*(TOS + 9) → TOS; ↓P
`1234`	Zero + 1234 → TOS; ↓P
`fedcba98`	Zero + fedc0000 → TOS; ↓P TOS + ba98 → TOS

Table 4. Uses of SC32 Load/Store Instructions

Table 4 shows several example uses of the SC32 load/store instruction category. In the table, stack operations that push or pop the parameter stack are denoted by ↓P and ↑P respectively. The first two examples are implementations of `@` and `!`. The next example shows the fetch of a variable named `avariable` at address 327. Since the compiler knows this address at compile time, there is no reason not to bring it in line as a literal and combine it with the `@` that follows. Indexing into an array or fetching a member of a record structure can be managed similarly. A

peephole optimizer has been written as part of our metacompiler that handles all of these cases. The load address low instruction allows the creation of small literals in the range 0 to 65535 in one cycle. Most literals found in Forth programs are within this range. Larger literals can be built in two cycles.

The micro instruction is the workhorse of the processor since it is used to implement most of Forth's primitive operations. All micro instructions consist of an operation performed on R1 and *TOS* with the result stored in R2. The ALU field selects the operation performed. This field has two formats, one for doing arithmetic or logic operations and one for doing shift, multiply, or divide steps (see Table 5). The Sel field selects the interpretation of the remainder of the ALU field. The encoding of the ALU subfields is shown in Table 6.

arith	Sel:1	Bsrc:1	ALUcond:4	Cin:4	Flag:1	ALUop:7				
shift	Sel:1	Bsrc:1	ALUcond:4	Cin:4	Flag:1	Shift:2	Sin:1	Step:2	Flagin:1	?:1

Table 5. ALU Instruction Format

In the arithmetic micro instructions, the ALUop field controls the ALU. The carry input to the ALU, selected by the Cin field, may be either 0, 1, or the flag value computed in an earlier instruction. If the Flag field is enabled then *FL* is loaded with the ALU condition selected by the ALUcond field. Possible ALU conditions include overflow, carry, zero detect, and signed or unsigned comparison results. After the arithmetic operation is done, the Bsrc field determines whether the ALU result or the value of *FL* is stored in R2. Note that in Table 6, *FL* denotes the flag being computed in the current instruction and *FL'* indicates the previous value of *FL*. If no new value is loaded into the flag on the current instruction then $FL \equiv FL'$. Only micro instructions can effect the flag.

Shift micro instructions are interpreted similarly except that the seven bit ALUop field is replaced with a two bit Step field and shift control fields are added. The Step field provides an ALU no-op when only shifting and two forms of conditional add for implementing a multiplication algorithm or a restoring divide algorithm. The conditional steps work by forcing the A input of the ALU to zero if the condition (in parentheses in Table 6) is false. The Shift field chooses left, right, or no shift. The left shift input always comes from *FL'*. The right shift input is selected by the Sin field and can be *FL'* or the current ALU condition. If a shift micro instruction loads the flag, Flagin selects the source of the flag input: the ALU condition or the result of a shift. The Shift field determines which shifter provides the result.

Table 7 shows how some representative Forth primitives are implemented with the SC32 micro instruction. The final entry in the table illustrates how multiple Forth primitives can be packed into one SC32 instruction.

The interpretation of almost all of the instruction fields is independent of other fields. For example, the Next field allows *TOR* to be loaded into the program counter and fetch the next instruction. But to implement a return from subroutine instruction you must also specify popping the return stack in the Stack field. The interdependency between the Flag, Shift, and Flagin fields described above is the only exception.

There are a few restrictions on the execution of some of the instructions that are not evident from the above tables. The *PC* cannot be explicitly loaded by an instruction even though this possibility is implied by the R2 field in Table 2. The *PC* may only be used as the source of data in a store instruction. Not all of the ALU conditions in Table 6 can be guaranteed to have meaningful values if the ALU is doing a combinational logic operation. Only the 0, 1, Z, N, and their derivative conditions will be valid. All of the conditions are meaningful when the ALU is performing arithmetic.

Field	Action	Size
Bsrc	result driven on Bbus ALU, FL	1
ALUcond	ALU condition* 0, Z, N, C, V, NxorV, ¬C | Z, (NxorV) | Z, 1, ¬Z, ¬N, ¬C, ¬V, ¬(NxorV), ¬(¬C | Z), ¬((NxorV) | Z)	4
Cin	carry input 0, 1, *FL'*, ¬*FL'*	2
Flag	Flag control nop load	1
ALUop	*ALU* operation* 0, -1, b, a, ¬b, ¬a, a·b, a|b, a xor b, ¬(a·b), ¬(a|b), ¬(a xor b), b−a−¬Cin, a−b−¬Cin, a+b+Cin, b+Cin, a+Cin, ¬b+Cin, ¬a+Cin, b−¬Cin, a−¬Cin,	7
Shift	select shifter operation shift left shift right none	2
Sin	right shift source FL' ALUcond	1
Step	step operation* b+Cin b−a−¬Cin conditional a+b+Cin(FL') conditional a+b+Cin(¬FL')	2
Flagin	source of flag input ALUcond shift out	1
* legend: b≡R1, a≡TOS and ¬≡not, ·≡and, |≡or.		

Table 6. ALU Encoding

4. *Instruction Execution*

Almost all SC32 instructions are fetched and executed in two cycles (see Figure 2). However, since the next instruction is fetched while the current instruction is being executed, the net throughput is one instruction per cycle. Load and store instructions require an extra cycle to execute since accessing memory prevents an instruction fetch. The first cycle (which is identical to the normal execute cycle) is used to compute an address, while the extra cycle actually does the load or store.

Each cycle consists of two phases. In the first phase of the cycle, the operands are fetched from registers and placed in the ALU input latches. Concurrently, the address of the next instruction is sent to external memory. On the second phase, the ALU operates and the results are sent to the destination register. The new instruction is received and latched.

The only time the two phase execution is apparent to the programmer is when pushing or popping the stacks. Pushing or popping a stack does not take effect until the second phase. So

Primitive	SC32 Instruction R1 op TOS → R2; cc → FL; stackop; next
dup	TOS → TOS; ↓P
over	SOS → TOS; ↓P
>r	TOS → TOR; ↑P; ↓R
r>	TOR → TOS; ↑R; ↓P
1+	TOS+1 → TOS
0=	TOS; Z → FL → TOS
+	SOS+TOS → TOS; ↑P
<	SOS−TOS; NxorV → FL → TOS; ↑P
exit	Next=TOR; ↑R
over 0<	SOS; N → FL → TOS; ↓P

Table 7. SC32 Implementation of Some Typical Forth Primitives

an instruction that references *SOS* in R1, *TOS* in R2, and pops the parameter stack is referencing the same physical register on both phases (see definition of + in Table 7).

5. *Stack Caching*

A mechanism is provided to allow a stack to grow larger than the on-chip registers. A stack caching algorithm is implemented in the SC32 giving the programmer the illusion of arbitrarily large on-chip stacks. Since the instruction set allows access to the top four elements of either stack at any time and since a store instruction can pop a stack then write out the fourth element down, the algorithm must guarantee that the top five stack elements are always present.

We have observed that the stacks of running Forth programs stay near a certain depth for long periods of time while many small oscillations of the depth occur. The caching algorithm, using the on-chip registers as a window into the stack, attempts to adjust itself so that the window is centered on the average depth. The goal is to minimize the number of times a cache overflows or underflows.

The registers are used as a circular buffer (see Figure 3). Two sliding points mark the overflow and underflow positions of the buffer. A push causes the stack pointer to increment. If the stack pointer reaches the overflow mark, the register at the bottom of the window (one past the new stack pointer) must be pushed onto an external stack. The processor inserts two cycles to write

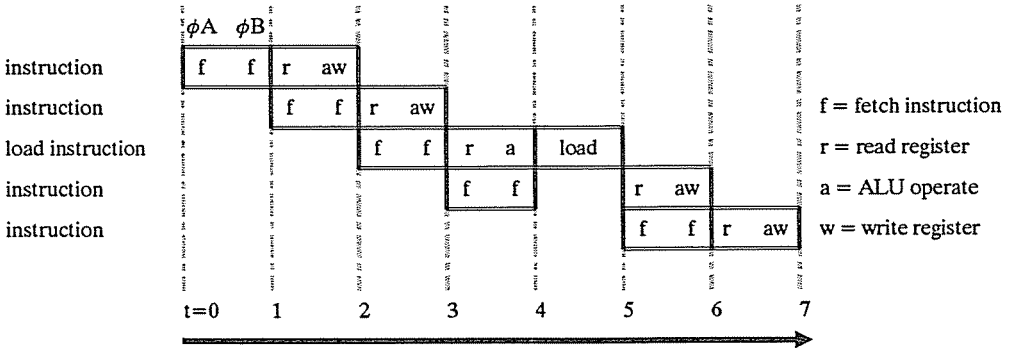

Figure 2. Instruction Execution

the register out to external memory and to adjust the overflow marker. In the first cycle a *UDR* dedicated to point to an external overflow area is decremented and the overflow/underflow markers are slid one register clockwise. In the second cycle, the register one past the stack pointer is written to the overflow area. On the first cycle of underflow a value is read from the overflow area into the register four positions below the stack pointer and the markers are slid one position counterclockwise. On the second cycle the dedicated *UDR* is incremented. *UDR0* is devoted to the return stack and *UDR1* to the parameter stack.

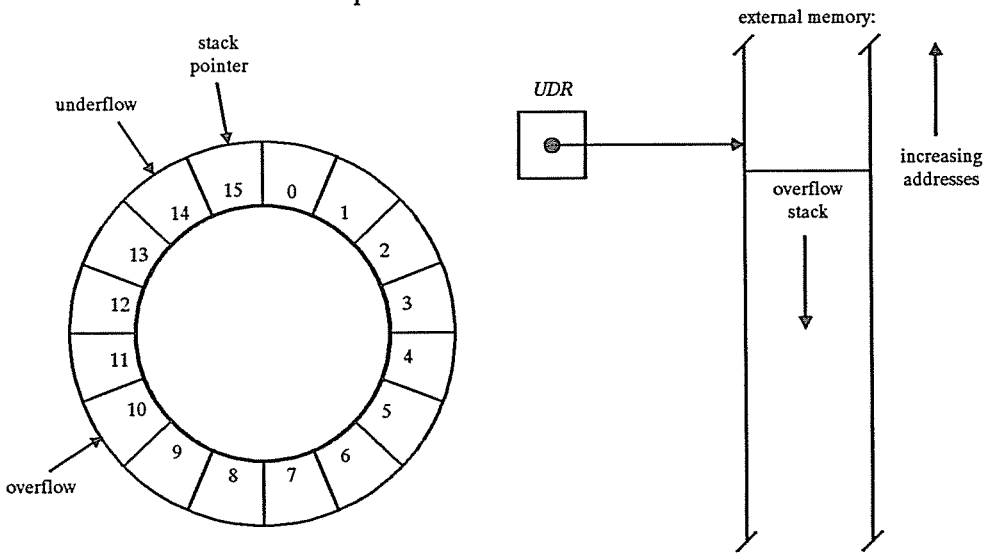

Figure 3. An Initialized Stack Cache

A cache enable bit, the stack pointers, and overflow markers are available in the *PSW*. The cache enable bit is cleared on reset but the stack pointers and markers must be initialized by the programmer. The format of the *PSW* is given in Table 8. The underflow marks are always twelve registers below the overflow marks. Figure 3 shows the configuration of the cache after a typical initialization.

PSW format	unused:14	ie:1	cache enable:1	parameter top:4	parameter overflow:4	return top:4	return overflow:4

Table 8. PSW Format

6. *Reset and Interrupts*

An external reset causes the processor to fetch and execute an instruction from memory location 0. A good instruction to place there is a subroutine call to the starting point of an initialization program. The reset clears the cache enable and the interrupt enable bits.

The SC32 has a single interrupt request pin. An interrupt response occurs when interrupt request is asserted and interrupts are enabled. The interrupt response is similar to the reset response except that memory location 1 is used. In addition, an interrupt acknowledge (INTACK) signal is asserted by the processor. A system designer may choose to use this signal so that the interrupting device, instead of the memory system, delivers the interrupt vector.

The interrupt enable bit (*ie*) is available to the programmer via the *PSW*. Writing to the *ie* is internally delayed by one cycle. This allows an interrupt service routine to reenable interrupts and then return from the interrupt so that a response to a pending interrupt is delayed until after

the return. On entering an interrupt service routine the return address plus one is stored on the return stack. It is the programmer's responsibility to decrement this address before returning from the interrupt.

The SC32 also has a Direct Memory Access (DMA) request pin. When an external device requests DMA, the processor tristates the address, data, and read pins and asserts a DMA acknowledge signal. The external device controls the bus until it releases DMA request.

7. Discussion

The SC32 was designed and implemented in six months. This short time frame limited the number of features that could be addressed and constrained the complexity of the design. Architectural features with proven utility were borrowed from our past design efforts and from current practice in conventional processor design. When neither history nor analysis showed a clearcut advantage for one decision over another, the simpler approach was usually chosen.

The SC32 inherits many features from FRISC 1 and 2. Measurements made during the design of FRISC 1 showed that subroutine call and return were the most frequently executed operations in Forth programs **[FRAE86]**. The design of FRISC 1 and 2 concentrated on a fast subroutine call and easy implementation of Forth's stack and arithmetic primitives. FRISC 1 and 2 have two instruction formats: a subroutine call and a user-defined microcode instruction. The msb of the instruction determines its type. A zero indicates that the remaining 31 bits are the address of a subroutine to call. The call executes in one cycle. A one in the msb indicates that the following 31 bits are a microcode word that directly controls the resources of the chip's data path. The microcode word can represent most Forth primitives (e.g. **dup**, **over**, **+**, **<**, **0=**, etc.) and the data path can execute most primitives in a single cycle. Primitives that must access memory take two cycles to execute. These include **branch**, **?branch**, **@**, **!**, and **(literal)**.

The Forth instruction frequency measurements done during the design of FRISC 1 showed that, after calls and returns, the most common instructions were loads, stores, and literals. These results are more in line with what is observed in conventional programming languages **[KATE85]**. Consequently, we were able to borrow ideas from many other processor designs where these issues have already been studied. In particular, our single register-indirect-plus-offset addressing mode is found in most RISCs **[HENN82A]**, **[PATT85]**. This addressing mode covers the most common array and record structure access operations. Register indirect addressing and absolute addressing (using the *Zero* register, another common RISC feature) are simply special cases of the one addressing mode. More complex, less frequently used addressing modes can be built using multiple instructions **[CHOW87]**. The SC32's register-indirect-plus-offset load and store instructions capture many Forth programming idioms in addition to Forth's traditional **@** and **!** (see Section 3).

Given the load instruction, it was relatively easy to design a load address instruction. This allows the most common literal values to be introduced into the data path in one cycle. Other instruction enhancements over FRISC 1 and 2 are a single cycle branch and a conditional add instruction. The conditional add can be used to construct a multiply step with two cycles per bit or a divide step with three cycles per bit.

Another improvement found in the SC32 is the Next field that selects the source of the address of the next instruction. Usually, Next specifies the *PC* but *TOR* can also be used. Thus, as with the Novix NC4016 **[GOLD85]**, concurrent execution of an instruction and a subroutine return is possible. A peephole optimizer can frequently combine return operations with the preceding primitive. Applying the optimizer to a large (12,000 line) Forth program resulted in the elimination of ~25% of the returns with this technique. The peephole optimizer also eliminates returns by converting a call-return pair into a branch. On the same program, ~50% of returns were removed in this way for a total of ~75% of all returns eliminated.

The SC32 has no pipeline other than the overlap of instruction fetch and execution. Deeper pipelines are common in RISC processors designed to execute conventional programming languages. Much effort has gone into developing hardware and software techniques that avoid the pipeline stalls caused by branch instructions **[MCFA86]**. These typically involve a delayed branch instruction and a compiler that can fill the delay slots. This issue would be even more critical in a pipelined Forth processor. An examination of typical Forth programs indicated that the control flow was changed (via calls, returns, or branches) very often: about once every three or four instructions **[FRAE86]**. Since it was not obvious how effective a compiler would be at filling all the delay slots, we decided on a shallow pipeline and a simple compiler.

One of the most important aspects of the design of a Forth processor chip is the management of the stacks. All three FRISCs have used stack caching. FRISC 1 and 2 used a naive cache management algorithm with cache overflow and underflow serviced by high priority interrupt routines. In the SC32, a much improved algorithm has been implemented in hardware.

Our stack caches are not true caches (a value held on chip is not a copy of a value in memory) but are top of stack buffers. In this respect they differ from the stack caches used on the AT&T Bell Lab's CRISP machine **[DITZ87]**. Our stack buffers are more closely related to the register window schemes used in some RISC processors **[TAMI83]**. Register window machines buffer recent procedure invocation frames whereas we buffer individual registers.

The two key design parameters of the stack cache are its size and the number of registers written on overflow and read on underflow. To choose the number of registers moved on overflow/underflow, we studied stack caches of 8, 16, and 32 registers. The number of registers moved on overflow/underflow was varied from one to the size of the cache minus five (since our instruction set allows the top five elements to be referenced within an instruction, they must always be in the stack cache). Each stack cache configuration was simulated, using stack depth traces obtained for seven Forth programs (see Table 9). The performance of each cache configuration was extremely sensitive to initial stack depth, so the initial depth was varied over the possible range. The worst case behavior was used to characterize each configuration. (A more complete description of this study should appear shortly **[HAYE89]**.)

flower:	A graphics program drawing a complex geometric figure.
meta:	The (meta) compilation of a new Forth system.
neural:	A back propagation neural network simulation learning xor.
traps:	A 50 rule expert system for spacecraft TRAjectory Preprocessing.
huff:	Huffman encode a text file.
fib:	Recursively compute the 24th Fibonacci number.
acker:	Recursive Ackerman's function.

Table 9. Stack Analysis Benchmark Programs

The simulations measured overflows/underflows per primitive executed, an implementation independent quantity. We would like to know the percentage of cycles spent on cache management (overhead) given a particular implementation. Therefore, cost models of different implementations were applied to the simulation results in a post processing phase. For example, Figure 4 shows the overhead in a 16 register parameter stack cache where each overflow/underflow is handled in hardware stalling the processor two cycles (it was assumed each primitive could be executed in one cycle). These results strongly indicate that writing one register on overflow and reading one register on underflow minimizes cache management overhead. The results for the return stack and for 8 and 32 register caches were similar. In fact, the one register conclusion held for all implementations that we studied!

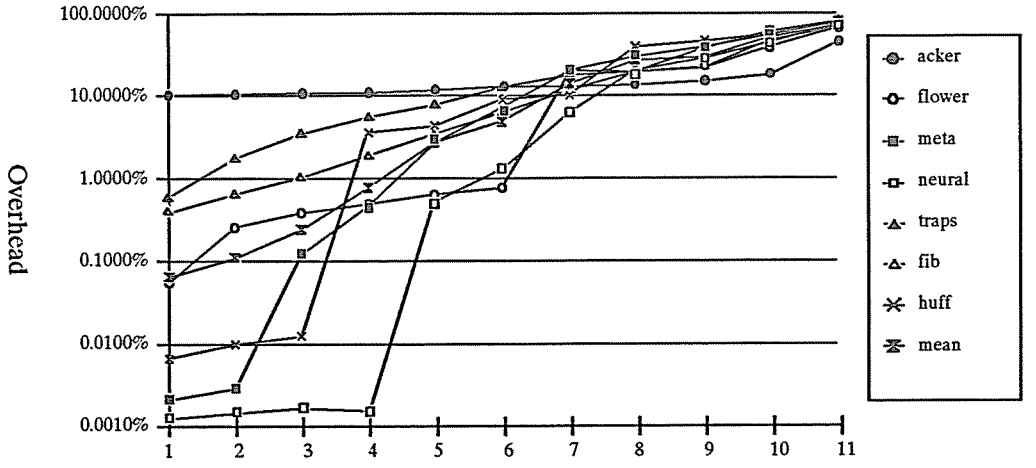

Figure 4 Stack Caching Overhead vs. Number of Registers Written/Read

These results convinced us to implement a one register overflow/underflow algorithm in hardware in the SC32. In Figure 4, the leftmost set of points (one register written/read) predict the overhead that should be seen in the SC32's parameter stack cache. The overhead is under 1% for all the benchmarks except acker. The stack depths for this recursive function vary in a wildly chaotic fashion. This is an atypical Forth program and was run to bring out the worst possible behavior of the stack caching algorithm. However, the overhead reached only 10% in each stack for a total of 20% overhead.

The choice of stack size is influenced by two conflicting demands: minimizing stack overflow/underflow overhead by having a large cache, and minimizing context switch times by having a small cache. The stack traces described above were used again to study the effect of intermittent context switches on the stack cache management. A "context switch" was introduced at intervals of 1000, 10000, and 100000 primitives. Context switching is done by pushing a stack fifteen times, letting the overflow mechanism write out the cache contents, then popping the stack fifteen times, letting the underflow mechanism load in a new context. No "cost" was assigned to the switch itself, but the effect of the switches on the number of overflows and underflows per primitive was calculated.

Figure 5 shows the cache management overhead in the parameter stack with context switching (assuming a one register move on overflow/underflow) versus the cache size. Each point represents the geometric mean of the overhead of all seven benchmarks. The curve labeled "infinity" is the no context switching case shown for comparison. As expected, the overflow/underflow overhead decreases with larger caches and increases with more context switching. However, beyond a certain point, larger caches offer diminishing returns. We conclude that a 32 register cache is best, but that 16 works almost as well when context switching is considered.

A number of features were deliberately excluded from the SC32 design. The SC32's intended application is advanced embedded systems. Memory management facilities are typically not needed in embedded systems and support for such is not provided. The SC32 does not support byte addressing; memory is addressed as 32 bit words. Providing only word addressing simplifies the instruction set and lets the processor run faster by avoiding the need for byte positioning multiplexors [HENN82B]. Finally, the SC32 has no floating point support. This was beyond the scope of what could be accomplished in six months.

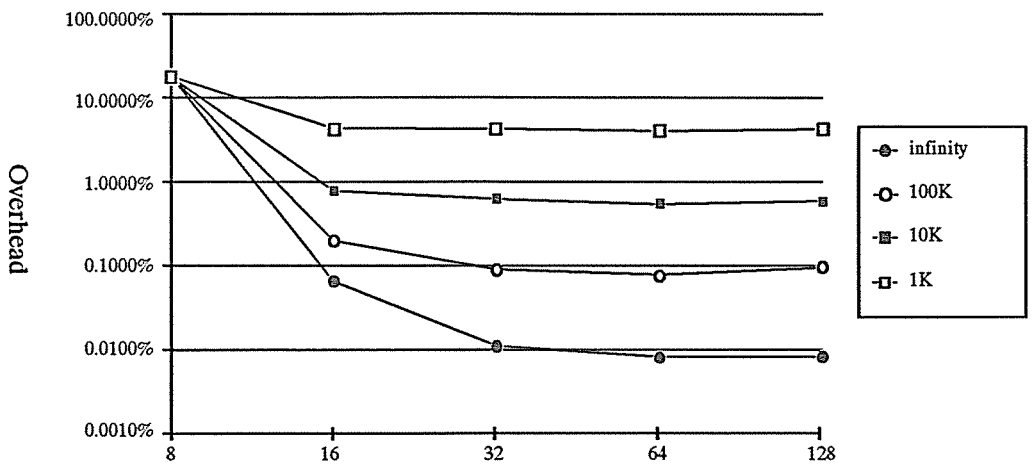

Figure 5. Stack Caching Overhead vs. Cache Size

8. *Results and Acknowledgements*

The SC32 architecture has been implemented in a 34,000 transistor CMOS chip. The chip features that use the largest number of transistors are the stack caches (~40% of all transistors), ALU and ALU condition logic (~20%), stack cache management (~10%), and instruction decode and control (~10%). The internal cycle time of the chip was dictated by the speed of the ALU plus a subsequent ALU condition being loaded into a register. The processor needs 35-55ns. external memories to run at 10MHz.

Chips have been fabricated by United Silicon Structures using a 2μm direct E-beam write-on-wafer process. Chips from the first fabrication run are fully functional and work at 10MHz. The part is packaged in an 84 pin PGA and consumes 650 mW.

This design drew heavily on ideas from earlier FRISC designs. Therefore, the contributions of Martin E. Fraeman, Robert L. Williams, and Thomas Zaremba are acknowledged here. Finally, we would like to thank the far sighted members of APL's Computer Architecture Thrust Panel and IR&D Committee for supporting this work. This work was done under Navy contract N00039-87-C-5301.

9. *References*

[CHOW87] Chow, F., Correll, S., Himmelstein, M., Killian, E., Weber, L. "How Many Addressing Modes are Enough?," *Proc. of the 2nd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, October 1987.

[DITZ87] Ditzel, D.R., McLellan, H.R., Berenbaum, A.D., "Design Tradeoffs to Support the C Programming Language in the CRISP Microprocessor," *Proc. of the 2nd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, October 1987.

[FRAE86] 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.

[GOLD85] Golden, J., Moore, C.H., Brodie, L. "Fast processor chip takes its instructions directly from Forth," *Electronic Design*, March 21, 1985, pp. 127-138.

[HAYE86] Hayes, J.R. "An Interpreter and Object Code Optimizer for a 32 Bit Forth Chip," *Proc. of the Eighth FORML Conference*, 1986.

[HAYE88] Hayes, J.R., Lee, S.C. "The Architecture of FRISC 3: A Summary," *Proc. 1988 Rochester Forth Conference.*

[HAYE89] Hayes, J.R. "Design Tradeoffs in a Top of Stack Cache," submitted for publication.

[HENN82A] Hennessy, J., Jouppi, H., Baskett, F., Gross, T., Rowen, C., Gill, J., "The MIPS Machine," *Proc. Compcon*, February 1982.

[HENN82B] Hennessy, J., Jouppi, H., Baskett, F., Gross, T., Gill, J., "Hardware/Software Tradeoffs for Increased Performance," *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, 1982.

[KATE85] Katevenis, M.G.H., *Reduced Instruction Set Computer Architectures for VLSI*, MIT Press, 1985.

[MCFA86] McFarling, S., Hennessy, J., "Reducing the Cost of Branches," *Proc. 13th Annual Int'l Symp. on Computer Architecture*, 1986.

[PATT85] Patterson, D.A., "Reduced Instruction Set Computers," *Comm. ACM*, 28,1, January 1985, pp. 8-21.

[TAMI83] Tamir, Y., Sequin, C.H., "Strategies for Managing the Register File in RISC," *IEEE Trans. on Computers*, C-32, 11, November 1983, pp. 977-989.

[WILL86] 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.

John R. Hayes received his B.S. degree in electrical engineering from the Virginia Polytechnic Institute and State University in 1982. He also received an M.S. degree in computer science from Johns Hopkins University in 1986. After joining the Applied Physics Laboratory of Johns Hopkins University in 1982, he wrote flight software in Forth for satellite based magnetometer experiments and for the shuttle based Hopkins Ultraviolet Telescope. He spent several years designing Forth language-directed microprocessors culminating in the SC32. He is currently applying the SC32 in a variety of projects. Other research interests include computer architecture and programming language design and construction.

Susan Ciarrocca Lee has a B.A. in Physics from Duke University (1973), an M.S. in Computer Science from the John Hopkins Evening College (1978), and an M.S. in Technology Management from the Whiting School of Engineering (1988). She has spent most of her career at the Johns Hopkins University Applied Physics Laboratory, where she has worked on both analysis software and real-time software for satellites, submarine tests, and biomedical devices. Ms. Lee became interested in Forth when working on embedded systems for both satellites and biomedical devices. The SC32 development resulted from a desire to capture both the excellent real-time software development environment of Forth and the speed of conventional, compiled languages.