SUSAN C. LEE and JOHN R. HAYES

DEVELOPMENT OF A FORTH LANGUAGE DIRECTED PROCESSOR USING VERY LARGE SCALE INTEGRATED CIRCUITRY

A 32-bit microprocessor has been developed for use in embedded computer systems. It supports an interactive programming environment on an embedded computer while providing the performance of compiled languages. The microprocessor hardware was designed to execute the Forth programming language directly.

INTRODUCTION

The Applied Physics Laboratory is heavily involved in the development of embedded computer systems. An embedded computer is a special-purpose computer residing in and controlling a piece of hardware such as a missile or telescope. One characteristic of an embedded system is the expense of software development. The hardware environment of an embedded system is usually inadequate to support its own software development, and an extra step must be added to the traditional edit/compile/debug cycle: the software is edited and cross-complied on a general-purpose computer and then downloaded to the target computer for debugging, which usually must be done with logic analyzers and in-circuit emulators. As the computational sophistication of embedded processors increases, the effectiveness of those techniques diminishes and the software development time increases.

The availability of new tools to support the quick development of custom very large scale integrated (VLSI) circuit chips suggested an alternative approach to the problem. We used Silicon Compiler Systems' Genesil silicon compiler to explore variations on a 32-bit architecture to implement the Forth language directly. Forth was chosen because of its excellence as an embedded-system development language. Since it is extremely small, it can be run on the target system, eliminating cross-compilation and downloading. The interactive nature of Forth allows debugging techniques, such as interrogating variables and checking program flow, to be used on the target system without recourse to in-circuit emulators and logic analyzers. Although embedded systems have been developed with Forth using commercial processors, the mismatch between the processor architecture and the Forth language causes 35% to 50% of the processor's time to be wasted in interpretation overhead. By tailoring our architecture directly to Forth, we can achieve significant improvements in processing speed. A 32-bit architecture was chosen to allow a large uniform address space.

A key feature of our approach was the use of the Genesil silicon compiler to implement the processor architecture as a VLSI circuit chip. The compiler allowed us to create candidate implementations and find their speed, size, and power requirements quickly. It also provided independence from implementation technology. The design is specified at a high level and can be compiled for a large number of possible silicon processing technologies and fabrication lines. As new fabrication processes are invented, they can be added to the compiler system, and the design can be recompiled.

The processor is being designed into ground support equipment for the Ocean Topography Experiment and Navy Radar Altimeter Program altimeters. It is also being used as the flight processor for a magnetometer experiment on Freja, a Swedish satellite. The processor has been licensed to a commercial vendor (Silicon Composers, Inc., Palo Alto, Calif.) and is being marketed under the name SC32.

IMPLEMENTING FORTH IN HARDWARE

Forth is an interactive, interpreted language; statements typed on the keyboard are translated into machine language by a program called the interpreter and are executed immediately. A Forth system can be used as a kind of calculator by typing statements like "2 + 2" in Forth syntax on the keyboard; the Forth interpreter translates the typed statement, executes it, and prints the answer on the terminal. For more complex problems, programs can be entered from the keyboard and then executed by typing their names and arguments.

Interpreted languages have an advantage over compiled languages in that the ability to test and debug programs interactively shortens the development cycle time. This is especially true with the Forth language, whose simple syntax allows it to be extended uniquely for each application. Unfortunately, because of the need for run-time interpretation, the execution speed of Forth on conventional computers suffers compared with that of compiled languages. Forth-oriented processor chips, by treating Forth as object code, eliminate the performance penalty of run-time interpretation. Consequently, interactive Forth programs running on the SC32 execute just

as fast as equivalent compiled programs on conventional microprocessors.

Forth uses a two-stack programming model: the parameter stack, which passes arguments to functions, and a control flow stack, which primarily holds subroutine return addresses and is called the return stack. Most Forth primitives (built-in functions like + and −) take operands from one or both stacks, push or pop the stacks (increment or decrement the stack pointers), and return a result to one of the stacks. For example, 2 + 3 is calculated in Forth by pushing 2 and 3 onto the parameter stack and executing +, which adds the first two stack elements and replaces them with the answer 5.

Forth is implemented on traditional processors using the approach shown in Figure 1. Because of the mismatch between Forth's stack model and the native processor, a layer of run-time interpretation is necessary. A tiny assembly language program called the inner (or address) interpreter is written for the bare processor. Forth's primitive stack operators, also written in assembly language, are implemented in the kernel layer. The top layer of a Forth system, the interactive outer interpreter, is written in Forth.

The Forth inner interpreter program on traditional processors uses a technique called threaded code.[1] The definitions of all high-level (nonprimitive) programs consist of a list of addresses of the definitions of the program's constituents (sometimes called threads, hence the term threaded code). The definitions of Forth primitives are in the traditional processor's assembly language. When a high-level program is executed, the inner interpreter program traces through the list of addresses. Each time the inner interpreter encounters an address in the list, it pushes the return address on the return stack and jumps to the address it found. If there is another address at that location, the process is repeated. The inner interpreter program nests down as deeply as necessary until a primitive operation defined in assembly language is found, and the assembly language is executed. The inner intepreter program then "threads" its way back to the starting point using the return addresses saved on the return stack. Not surprisingly, in Forth systems implemented on traditional processors, 35% to 50% of the system's time is consumed by the inner interpreter.

The SC32 eliminates this run-time overhead by eliminating the inner interpreter. The address of a primitive is replaced with the actual object code for the primitive. The addresses of other high-level routines are replaced with subroutine calls. At run time, a list of SC32 instructions is traced instead of a list of addresses. The inner interpreter has become the fetch–execute cycle of the processor.

Readers familiar with advanced Forth implementation techniques will realize that this scheme is "subroutine threaded code with in-line expansion." Theoretically, nothing precludes using this technique on conventional processors, but the mismatch between Forth and typical instruction sets would cause Forth programs to become much larger. For example, suppose several instructions are needed to implement the Forth stack primitive "dup" on a given processor. Each time dup

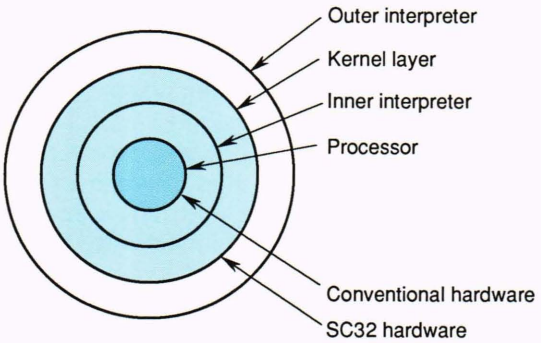


Figure 1. Hardware/software boundaries of the Forth virtual machine on conventional processors and on the SC32.

is required in a program, several instructions must be stored instead of the single address needed for threaded systems. The resulting object code could be significantly bigger than the size of a thread. The instruction set of the SC32 has been designed so that almost all Forth primitives are implemented with one, single-word instruction, so the resulting code is as small as that produced by a threaded system.

We gained much experience and insight into Forth-oriented architectures during the design of the SC32's predecessors, the Forth Reduced Instruction Set Computers 1 and 2.[2–4] Many elements of the earlier architectures are clearly visible in the SC32. It is an improvement over previous designs with more efficient load, store, literal, and branching instructions; better support for multiplication and division; and a better approach to stack caching.

The SC32 instruction set was explicitly designed to implement Forth. Each instruction executes in one cycle except for memory loads and stores, which require two each. Most of Forth's primitive operations can be represented with one instruction. For example, Forth's binary (two-operand) arithmetic operations, such as + and −, are single instructions and execute in one cycle. Binary logic operations like **and**, **or**, and **xor** and binary comparison operators like =, <, and > also execute in one cycle. In fact, any possible Boolean function of two variables is possible. The SC32 also has many single-cycle unary (single-operand) comparison operators and arithmetic instructions, such as increment, decrement, test for zero, and test for negative.

An SC32 instruction can access the top four items on either the parameter stack or the return stack, thereby allowing single-cycle implementation of many Forth stack manipulation operators (e.g., duplicate the top item on the stack, drop the top item on the stack, or transfer an item from the parameter to the return stack). This also provides for easy use of two sets of **do loop** indices.

The SC32 instruction set is more general than Forth's pure stack virtual machine, and sequences of more than one Forth primitive can frequently be implemented with one instruction. Arithmetic and comparison operators can often be combined with preceding stack manipulation operations. For example, **over over =** is a commonly used sequence of Forth primitives that copies the

top two elements on the stack and checks them for equality; this sequence of three primitives can be combined into a single SC32 instruction.

The combination of multiple Forth primitives into one instruction is especially useful with the SC32's load and store instructions, which provide a single, powerful addressing mode that covers the most common array and data-structure access operations. Fetching a variable at memory location 134 or fetching the eighth cell of a data structure (both require multiple Forth primitives) can be done with one SC32 instruction. Forth's load operator (written @ in Forth) is simply a special case of the SC32 general-purpose load instruction. A Forth store instruction (! in Forth) actually takes two instructions: one to do the store and another to clean up the stack; however, the stack cleanup can frequently be folded into a following instruction. The SC32 Forth compiler contains a "peephole optimizer," logic that searches for Forth primitive sequences that can be combined into a single SC32 instruction. Such optimization results in significantly faster execution.

The SC32 has single-cycle call, branch, and conditional branch instructions. The call and branch instructions directly implement Forth's subroutine nesting operation and **branch** operation. Forth's **if**, **while**, and **until** operations are implemented with two instructions: one to test the value on top of the parameter stack followed by a conditional branch. The test frequently can be combined with preceding operations, resulting in two-cycle test and branch operations. Similarly, Forth-83's **loop** and **+ loop** (increment counter and loop) operations 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 65,535 to be pushed on the stack in one cycle. The quick return provides a way to return from a subroutine 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 the delivery of stack operands to the processor. A consequence of executing one instruction every cycle is the need to fetch a new instruction every cycle. No spare bandwidth is left 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 the instructions and data. Special 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, a small price to pay for its advantages. Since the stacks are kept in the same address space as the instructions and data, only one address/data bus is needed to access them, resulting in an easily used 84-pin package. It also allows a stack to grow potentially as large as the address space of the processor (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.

MICROARCHITECTURE

Figure 2 is a diagram of a data path of the SC32. A data-path diagram depicts the microarchitecture of a processor, that is, the connectivity of the data handling and storage elements that are visible to the programmer. Forth's two-stack programming model is supported by two stack caches in the data path. The parameter and return stack caches consist of sixteen 32-bit registers each. Both caches have two read ports and one write port. The instruction set allows explicit programmer access to the top four locations of either stack. Cache overflow and underflow are handled transparently to the programmer. Forth's heavy use of subroutines is also directly supported. The instruction set and the arrangement of the data path allow subroutines to be called in one cycle and some returns to occur in zero cycles.

In addition to the stack caches there are four global utility registers called user-defined registers (UDR's). 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, to implement an additional stack or a frame pointer for a traditional language such as C.

The arithmetic logic unit (ALU) provides the usual 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 either the shift-out bit from one of the shifters or with any one of 16 ALU conditions. 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. The zero register is read-only and always returns the value zero. It is useful for constructing literals and addresses for loads and stores. The program counter (PC) register keeps track of the next instruction to be executed. There is also a processor status word (not shown in Fig. 2) that contains the state of the interrupt system and stack caches.

Three global buses provide communication between the resources described above. At the beginning of the execution of an instruction, B-bus delivers an operand to the ALU from a register resource (stacks, utility registers, zero register, etc.). The other ALU operand arrives on the T-bus and is always either the top of the parameter stack or a literal value from the instruction word. After the ALU operates, the result is sent to a destination register via the B-bus. The B-bus is connected to the off-chip data bus when doing load or store instructions. The A-bus addresses the external world. Normally, the program counter is driven onto the A-bus to fetch the next instruction. The top of the return stack can also drive the A-bus so that a return from a subroutine call can occur concurrently with the execution of

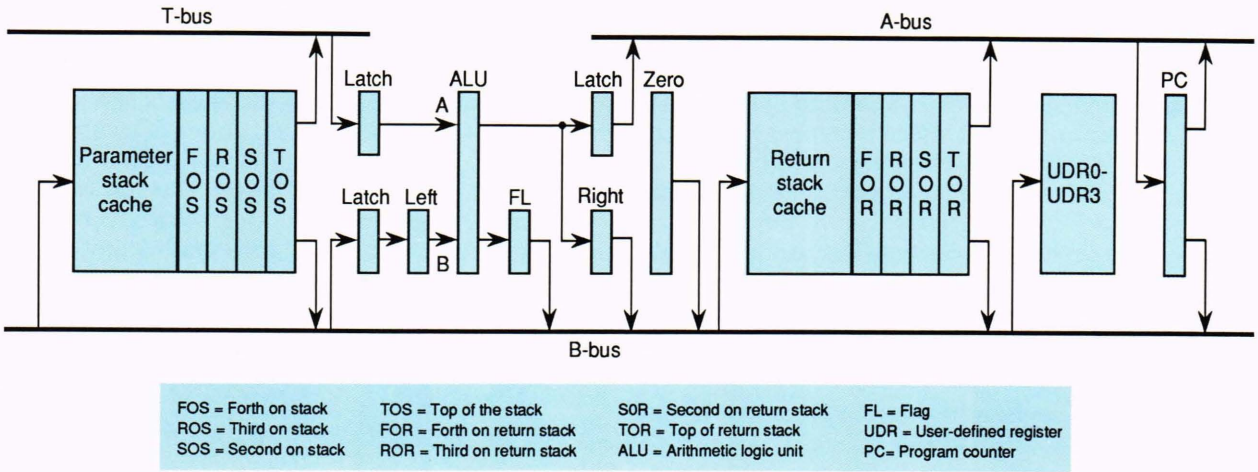


Figure 2. Data-path microarchitecture of the SC32.

many instructions. During load/store instructions, the ALU calculates an address that is subsequently driven onto the A-bus. Finally, The UDR's may also drive the A-bus during stack-cache management operations.

Although it is not shown in the figure, there is a path from the instruction register to the A-bus for branch or call instructions.

INSTRUCTION SET ARCHITECTURE

In keeping with the philosophy of the Reduced Instruction Set Computer, the SC32 instruction set has only eight instructions: three control flow instructions, four load/store instructions, and a single microcode instruction. Every instruction is encoded in a single 32-bit word, so that only a single cycle is needed to fetch any instruction. Each of the three instruction categories has a different format, shown in Figure 3. The three most significant bits of the instruction determine its type and the interpretation of the remaining 29 bits.

The three types of control flow instructions are call, branch, and conditional branch. The conditional branch is taken if the flag (set by a previous instruction) is 0. As shown in Figure 3, the subroutine address or branch destination is an absolute address embedded in the instruction. Embedding the destination in the instruction allows control flow to change in a single cycle, but it limits the program address space to 2^{29} words.

Figure 3 shows that the upper 16 bits of the micro and load/store instructions have the same format. In both, placing a 4-bit code into the R1 field selects one of the registers on the B-bus as the source for an ALU operation. The 4-bit code in the R2 field selects one of the registers on the B-bus as the destination register for the result. The code placed in the stack control field selects any combination of "pushing" and "popping" the parameter and return stacks. Finally, the "next" field determines whether the incremented program counter or the top of the return stack is used to provide the address of the next instruction.

The single SC32 microinstruction is the workhorse of the processor since it is used to implement most of Forth's primitive operations. All microinstructions consist of an ALU operation performed on data selected by the R1 field and the top of the parameter stack, with the result stored in the register selected by the R2 field. As shown in Figure 3, the low 16 bits of the instruction word are used as the ALU field, to select the ALU operation performed. For example, the Forth operation + adds the two top stack elements and replaces them with their sum. The operation is encoded in a microinstruction by placing the codes for "second on stack" in R1, the "top of the stack" in R2, "+ " in the ALU operation field, and "pop the parameter stack" in the stack field. The execution of the instruction then performs addition on the top of the stack (by definition) and the R1 register (second on stack), pops the parameter stack (throwing away the old top of the stack), and writes the result of the addition into the current top of the stack location (i.e., over the old second on stack value). The ALU field has two formats: one for doing arithmetic and logic operations, and one for doing shift, multiply, and divide steps, selected by the format select field.

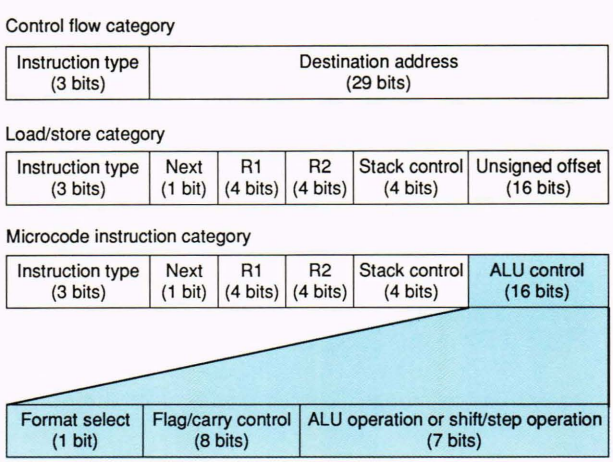


Figure 3. Thirty-two-bit instruction word formats for the SC32. ALU = arithmetic logic unit.

The four load/store instructions are load, store, load address low, and load address high. In these instructions, the ALU field of the microinstruction is replaced by an unsigned 16-bit number (offset) embedded in the low 16 bits of the instruction word. There is no need for an ALU field in a load/store instruction; instead, when a load/store is executed, addition is performed on R1 and the unsigned offset by definition. For the load/store instructions, the result of this addition becomes the address of the data to be transferred; for the load address low and load address high instructions, the result of the addition itself is the data. The operation of these instructions can be summarized as follows:

Load: the contents of the memory location (R1 + offset) → R2
Store: the contents of R2 → the memory location (R1 + offset)
Load address low: R1 + offset → R2
Load address high: R1 + (2^{16} × offset) → R2

Although it seems that only a single addressing mode, register indirect plus offset, is provided, other useful modes are obtained by choosing appropriate R1 and offset fields: setting the offset to zero produces a register indirect mode, and setting R1 to the zero register allows absolute addressing within the low 64K words of address space.

The load address instructions are "degenerate" loads in that an address is computed but no data are fetched. Instead, the address (the result of the addition of R1 and the offset) is saved in R2. The load address high instruction is similar to the load address low, except that the offset is shifted left 16 bits before being added to R1. The primary use for these two instructions is to allow literal values to be placed on the stack. Literals are numbers that are "hard-coded" into a program; for example, in the statement $y = x + 5$, 5 is a literal. Sixteen-bit literals can be pushed on the stack with a single load address low instruction by setting R1 to the zero register and placing the desired 16-bit literal in the offset field of the instruction word. Any 32-bit literal can be obtained by load address high followed by a load address low.

The first seven examples in Table 1 show uses of the SC32 load/store instruction category. The first two examples are implementations of @ (load) and ! (store). The next 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. The next examples show that indexing into an array or fetching a member of a record structure can be managed similarly. The load address low instruction allows small literals in the range 0 to 65,535 to be placed on the stack in one cycle, as shown by pushing the literal "1234" on the stack. Most literals found in programs are within this range. Larger literals, like the number "FEDCBA98" in Table 1, can be built in two cycles. The peephole optimizer handles all of these situations.

The last eight examples in Table 1 show how some representative Forth primitives are implemented with the SC32 microinstruction. The final entry illustrates how multiple Forth primitives can be packed into one SC32 instruction. The Forth **over** primitive gets the second on stack and pushes it on top of the stack. The **0<** primitive checks the top of the stack (now a copy of the former second on stack) and replaces it with a flag (−1 if the top of the stack is negative). Our processor can perform all these operations using one instruction: second on stack is selected as the input to the ALU (R1 field); the N (negative) ALU condition is selected to be loaded in the FL; and the flag field of the instruction word is set to force the FL, rather than the ALU result, to be placed on the B-bus. The parameter stack is pushed, creating a new top of the stack location. Since top of the stack is selected as the destination register (R2 field), FL is placed on the top of the stack. Thus, the effect of **over 0<** is achieved in one instruction.

INSTRUCTION EXECUTION

Almost all SC32 instructions are fetched and executed in two cycles (see Fig. 4). Because 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 loading or storing.

Each cycle consists of two phases, A and B. In the first phase, 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. In the second phase, the ALU operates and the results are sent to the destination register. The new instruction is received and latched.

The two-phase execution is transparent to the programmer except when pushing or popping the stacks, which is not done until the second phase. So an instruction that references second on stack in the R1 field, pops the parameter stack, and references the top of the stack in the R2 field is referencing the same physical register in both phases (see the definition of + in the previous section).

STACK CACHING

A stack-caching algorithm implemented in the SC32 gives 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 and then write out the fourth element down, the algorithm must guarantee that the top five stack elements are always present. The aim is to minimize the number of times the cache overflows or underflows.

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 depth occur. The caching algorithm, using the on-chip registers as a window into the stack, attempts to adjust the window so that it is centered on the average depth.

Table 1. Implementation of some typical Forth operations in the SC32 instruction set.

Operation	Forth code	SC32 instruction	
		ALU op.→destination	Stack operation
Load (replace memory address in TOS with memory address contents)	@	contents (TOS + 0)→TOS	none
Store register contents at memory address in TOS	!	TOS→address (TOS + 0) nop	Pop PS Pop PS
Load the contents of "avariable"	avariable @	contents (zero + 327)→TOS	Push PS
Load an element of "anarray"	over anarray + @	contents (SOS + 1234)→TOS	Push PS
Load the 9th element of a record	dup 9 + @	contents (TOS + 9)→TOS	Push PS
Push literal "1234" on stack	1234	zero + 1234→TOS	Push PS
Push literal "FEDCBA98" on stack	FEDCBA98	zero + FEDC000→TOS TOS + BA98→TOS	Push PS None
Duplicate the top of stack element	dup	nop SOS→TOS	Push PS
Copy the SOS on the top of stack	over	nop SOS→TOS	Push PS
Move the top of RS to top of PS	R>	nop TOR→TOS	Pop RS/Push PS
Move the top of PS to top of RS	>R	nop TOS→TOR	Push RS/Pop PS
Increment the top of stack	1+	TOS + 1→TOS	None
Add the top two numbers on stack	+	TOS + SOS→TOS	Pop PS
Test the top of stack for zero	0=	nop TOS; ALU cond Z→TOS	None
Check the SOS for negative	over 0<	nop SOS; ALU cond N→TOS	Push PS

Notes: TOS = top of the stack; PS = parameter stack; SOS = second on stack; RS = return stack.

The registers are used as a circular buffer (Fig. 5). 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 the register out to external memory and to adjust the overflow marker. In the first cycle, a UDR pointing 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 pointer in the UDR is incremented. UDR0 is dedicated to the return stack and UDR1 to the parameter stack.

A cache enable bit, the stack pointers, and overflow markers are available in the processor status word. The cache enable bit is cleared on reset, but the stack pointers and markers must be initialized by the programmer. The underflow marks are always 12 registers below the overflow marks. Figure 5 shows the configuration of the cache after a typical initialization.

EXTERNAL INTERFACE

An external reset causes the processor to fetch and execute an instruction from memory location 0. A good

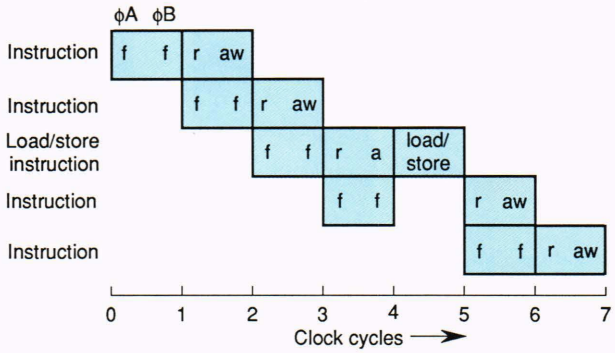


Figure 4. Instruction fetch and execution timing for microcode and load/store instructions. f = fetch instruction, r = read register, a = ALU operate, w = write register.

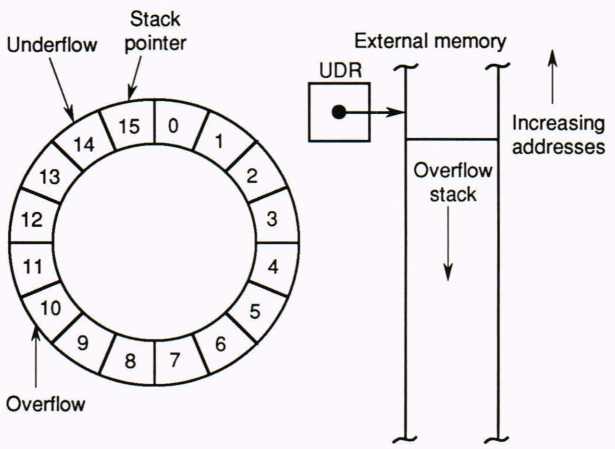


Figure 5. An initialized stack cache. UDR = user-defined register.

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 an 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 is available to the programmer via the processor status word.

The SC32 also has a direct memory access request pin. When an external device requests direct memory access, the processor tristates the address, data, and read pins and asserts a direct memory access acknowledge signal. The external device controls the bus until it releases the request.

DISCUSSION

The SC32 was designed and implemented in six months, a short period that 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 clear-cut advantage for one decision over another, the simpler approach was usually chosen.

Measurements made during the design of the Forth Reduced Instruction Set Computer 1 (FRISC 1) showed that subroutine call and return were the most frequently executed operations in Forth programs.[2] The design of FRISC 1 and 2 concentrated on a fast subroutine call and easy implementation of Forth's stack and arithmetic primitives. FRISC's 1 and 2 have two instruction formats: a subroutine call and a user-defined microcode instruction. The most significant bit of the instruction determines its type. A zero indicates that the remaining 31 bits are the address of subroutine to call, which executes in one cycle. A one indicates that the following 31 bits are a microcode word that directly controls the resources of the processor's data path. The FRISC 1 and 2 microcode instruction can represent most Forth primitives, and the data path can execute most primitives in a single cycle, although primitives that must access memory, including branches, loads, stores, and literals, take two cycles.

The Forth instruction frequency measurements made 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.[5] Consequently, we were able to borrow ideas for the SC32 design from many other processor designs for which those issues have already been studied. In particular, our single register-indirect-plus-offset address mode is found in most RISC's.[6,7] 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.[8] The SC32's register-indirect-plus-offset load and store instructions capture many Forth programming idioms in addition to Forth's traditional @ and !. Given the load instruction, it was relatively easy to design a load address instruction, allowing the most common literal values to be introduced into the data path in one cycle.

Other SC32 instruction enhancements 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 is the "next" field in the instruction word, which selects the source of the address of the next instruction. Usually, "next" specifies the program counter, but the top of the return stack can also be used. Thus, as with the Novix NC4016 microprocessor,[9] concurrent execution of an instruction and a subroutine return is possible. The peephole optimizer written for our compiler examines the primitive preced-

ing each return and frequently can combine return operations with the preceding primitive. Applying the optimizer to a large (12,000-line) Forth program resulted in the elimination of about 25% of the returns. The peephole optimizer also eliminates returns by converting call–return pairs into a branch. On the same program, about 50% of the returns were removed in this way for a total of about 75% of all returns being eliminated and a 5% to 10% improvement in execution speed.

Other RISC processors commonly use a technique called pipelining to improve their overall throughput. Pipelining is to processor execution as a "round" (e.g., Row, Row, Row Your Boat) is to singing. In a pipelined system, each instruction may require several stages (and clock cycles) to execute. A new instruction is introduced into the pipeline each clock cycle so that when one instruction is in the first stage of execution, the preceding instruction is in the second stage. Once the first instruction completes all the execution stages, each subsequent clock cycle completes the execution of another instruction. Unfortunately, if a branch instruction is executed, the pipeline must be emptied and refilled with instructions at the branch destination. This is known as a pipeline stall.

Deep pipelines (many stages) 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.[10] They 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.[2] It was not obvious how effective a compiler would be at filling all the delay slots, and we decided on a shallow pipeline and a simple compiler. Thus, the SC32 has no pipeline other than the overlap of instruction fetch and execution.

One of the most important aspects of the design of a Forth processor chip is the management of the stacks. All three FRISC's 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. A much improved algorithm has been implemented in hardware in the SC32.

Our stack caches are not true caches, because a value held on the chip is not a copy of a value in memory. Instead, they "buffer" the top of the stacks on the chip. In this respect, they differ from the stack caches used on the AT&T Bell Lab's CRISP machine.[11] Our stack buffers are more closely related to the register-window schemes used in some RISC processors.[12] Register-window machines buffer recent procedure invocation frames, whereas we buffer individual registers.

The two key design parameters of a 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:

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 encodes a text file
Fib: recursively computes the 24th Fibonacci number
Acker: recursive Ackerman's function

The performance of each cache configuration was extremely sensitive to initial stack depth, so the depth was varied over the possible range. The worst-case behavior was used to characterize each configuration.

The simulations measured overflows/underflows per primitive executed, a quantity that is independent of implementation. We want to know the percentage of cycles spent on cache management (the overhead), given a particular implementation, so cost models of different implementations were applied to the simulation results in a post-processing phase. For example, Figure 6 shows the overhead in a 16-register parameter stack cache; it was assumed that each primitive executes in one cycle, but that each overflow/underflow stalls the processor for two cycles. The 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!

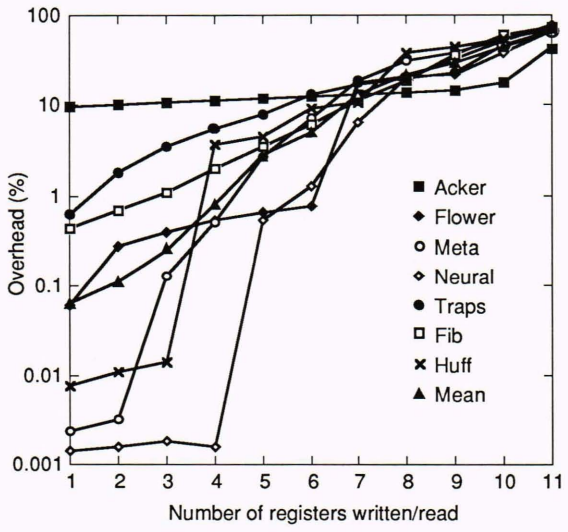


Figure 6. Stack-caching overhead versus number of registers written or read.

The top of stack cache described here was analytically modeled by Hasagawa and Shigei.[13] Their analysis assumed that the stack depth follows a random walk and predicts that the number of cache overflows and underflows will be minimized by writing out half of the registers on overflow. The cache overflow and underflow behavior revealed by our study is quite different from that predicted by a random-walk model. Cache overflow is minimized by writing one register. Apparently, the top of stack movement is more patterned than a random walk. This is quite reasonable. For example, consider that the variations in stack depth occurring within a program loop are repeated each time through the loop. Small oscillations about a particular depth are produced for long periods. Since repetition is what computers do best, we should expect stack depth to exhibit such patterned oscillations.

These results convinced us to implement a one-register overflow/underflow algorithm in hardware in the SC32. In Figure 6, the leftmost set of points (one register written/read) predicts 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 chaotically. This atypical Forth program was run to bring out the worst possible behavior of the stack-caching algorithm, but the overhead reached only 10% in each stack, for a total of 20%.

The choice of stack size is influenced by two conflicting demands: minimizing stack overflow/underflow overhead by having a large cache, and minimizing context switching times by having a small cache. The stack traces described above were used again to study the effect of intermittent context switches on stack-cache management. A "context switch" was introduced at intervals of 1000, 10,000, and 100,000 primitives. Context switching is done by pushing a stack 15 times, letting the overflow mechanism write out the cache contents, and then popping the stack 15 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 7 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. Beyond a certain point, larger caches offer diminishing returns. We concluded that a 32-register cache is best but that 16 works almost as well when context switching is considered.

Some features were deliberately excluded from the SC32 design. Its intended application is advanced embedded systems. Memory-management facilities typically are not needed in embedded systems and their support is not provided. Byte addressing is not supported, and memory is addressed as only 32-bit words. Providing only word addressing simplifies the instruction set and lets the processor run faster by avoiding the need for byte positioning multiplexers.[14] Finally, the SC32 has no floating point support; it was beyond the scope of what could be accomplished in six months.

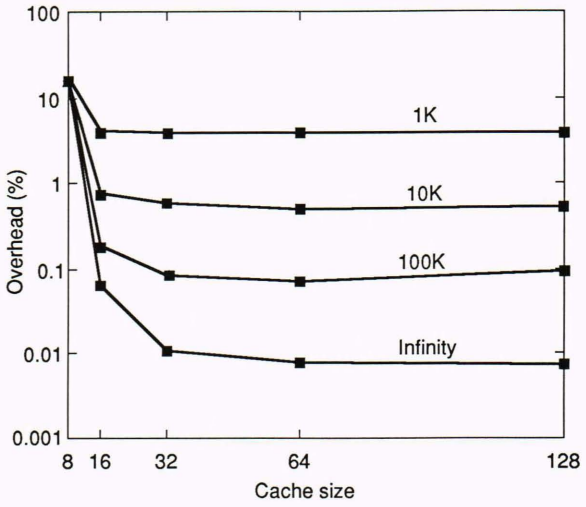


Figure 7. Stack-caching overhead versus cache size with context switching.

RESULTS

The SC32 architecture has been implemented in a 34,000-transistor 2-μm complementary metal-oxide semiconductor chip. The chip features that use the largest number of transistors are the stack caches (about 40% of all transistors), ALU and ALU condition logic (about 20%), stack-cache management (about 10%), and instruction decode and control (about 10%). The internal cycle time of the chip was dictated by the speed of the ALU plus a subsequent ALU condition code being loaded into a register. The processor needs 35- to 55-ns external memories to run at 10 MHz.

The prototype chips were fabricated by United Silicon Structures, which has a Direct E-beam Write On Wafer process. The new technology is especially appropriate for fabricating prototype chips. Since the silicon surface is patterned directly with an electron beam, no masks need to be made. Eliminating the mask-making step saves money and results in a faster design turnaround. We contracted the company to guarantee that we receive 15 working parts. A number of foundries are willing to satisfy this guarantee for designs created on the Genesil silicon compiler. Chips from the first fabrication run are fully functional and work at 10 MHz. The part is packaged in an 84-pin pin grid array and consumes 650 mW.

REFERENCES

[1] Ritter, T., and Walker, G., "Varieties of Threaded Code for Language Implementation," *BYTE* **5**, 206–227 (1980).
[2] Fraeman, M. E., Hayes, J. R., Williams, R. L., and Zaremba, T., "A 32 Bit Architecture for Direct Execution of Forth," in *Proc. Eighth FORML Conf.* (1986).
[3] Hayes, J. R., "An Interpreter and Object Code Optimizer for a 32 Bit Forth Chip," in *Proc. Eighth FORML Conf.* (1986).

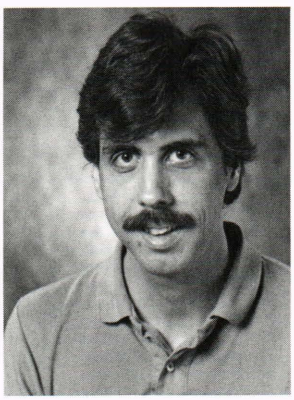
[4] Williams, R. L., Fraeman, M. E., Hayes, J. R., and Zaremba, T., "The Development of a VLSI Forth Microprocessor," in *Proc. Eighth FORML Conf.* (1986).
[5] Katevenis, M. G. H., *Reduced Instruction Set Computer Architectures for VLSI*, MIT Press, Cambridge (1985).
[6] Hennessy, J., Jouppi, H., Baskett, F., Gross, T., Rowen, C., et al., "The MIPS Machine," in *Proc. Compcon* (1982).
[7] Patterson, D. A., "Reduced Instruction Set Computers," *Comm. ACM* **28**, 8–21 (1985).
[8] Chow, F., Correll, S., Himmelstein, M., Killian, E., and Weber, L., "How Many Addressing Modes Are Enough?" in *Proc. 2nd International Conf. on Architectural Support for Programming Languages and Operating Systems* (1987).
[9] Golden, J., Moore, C. H., and Brodie, L., "Fast Processor Chip Takes Its Instructions Directly from Forth," *Electron. Des.*, 127–138 (21 Mar 1985).
[10] McFarling, S., and Hennessy, J., "Reducing the Cost of Branches," in *Proc. 13th International Symp. on Computer Architecture* (1986).
[11] Ditzel, D. R., McLellan, H. R., and Berenbaum, A. D., "Design Tradeoffs to Support the C Programming Language in the CRISP Microprocessor," in *Proc. 2nd International Conf. on Architectural Support for Programming Languages and Operating Systems* (1987).
[12] Tamir, Y., and Sequin, C. H., "Strategies for Managing the Register File in RISC," *IEEE Trans. Comput.* **C-32**, 977–989 (1983).
[13] Hasagawa, M., and Shigei, Y., "High-Speed Top-of-Stack Scheme for VLSI Processor: A Management Algorithm and Its Analysis," in *Proc. 12th International Symp. on Computer Architecture*, pp. 48–54 (1985).
[14] Hennessy, J., Jouppi, H., Baskett, F., Gross, T., and Gill, J., "Hardware/Software Tradeoffs for Increased Performance," in *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems* (1982).

ACKNOWLEDGMENTS—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 farsighted members of APL's Computer Architecture Thrust Panel and IR&D Committee for supporting this work. The work was done under Navy contract N00039-87-C-5301.

THE AUTHORS

SUSAN C. LEE was born in San Diego in 1952. She received a B.S. in physics from Duke University in 1973 and joined APL the same year. In 1978 she received an M.S. in computer science from The Johns Hopkins University G. W. C. Whiting School of Engineering. She received an M.S. in technology management from the G. W. C. Whiting School of Engineering in 1987.

At APL, Mrs. Lee has worked in the Space Department on the SAS-C, TIP, AMPTE, and TOPEX satellites. She also worked in the Submarine Technology Magnetics Group on several at-sea exercises. Her primary work area is software systems engineering.



JOHN R. HAYES was born in Washington, D.C., in 1960. He received a B.S. degree in electrical engineering from the Virginia Polytechnic Institute and State University in 1982 and an M.S. degree in computer science from The Johns Hopkins University in 1986. After joining the Computer Engineering Group of APL's Technical Services Department in 1982, he wrote flight software for satellite-based magnetometer experiments and for the Hopkins Ultraviolet Telescope. He spent several years designing Forth language-directed microprocessors culminating in the SC32. Mr. Hayes is currently applying the SC32 to ground support equipment for the TOPEX and SALT radar altimeters. Other research interests include computer architecture and programming language design and construction.