DESIGN TRADEOFFS IN STACK COMPUTERS

A PERSONAL EXPERIENCE

PHILIP KOOPMAN, JR. - WEXFORD, PENNSYLVANIA

When I started designing stack processors for WISC Technologies in 1985, little had been published about the architectural requirements of Forth engines. A substantial amount of architectural measurement had been performed on previous stack-based processors (in particular the Xerox Mesa architecture), but the behavior of single-stack processors for executing conventional languages is not representative of the types of things Forth processors do. When I started, all I knew was that Forth programs did a *lot* of subroutine calls, but beyond that I was groping in the dark. Here I hope to describe some of the history behind the development of the WISC and 32-bit RTX processors in terms of discoveries, blunders, and serendipity. Along the way, I will talk about the various requirements for implementing a high-speed Forth engine, and will describe the motivations underlying the design of Harris' 32-bit RTX architecture.

THE HARDWARE-FRENZY PHASE

The first phase of my continuing journey to stack-computer enlightenment was characterized by a frenzy of designing, building, debugging, and programming Forth hardware.

The WISC CPU/16

The WISC CPU/16 was my first stack computer design (and, for that matter, my first computer design of any type). The "WISC" stands for Writable Instruction Stack Computer. It was implemented entirely in 74LSxxx series TTL components, wire-wrapped on a single IBM-PC plug-in board. We produced a printed circuit board version once the design was shaken out. The design decisions for the CPU/16 were made in favor of simple and inexpensive prototyping first and foremost. This led to the decision to use a microcoded design, with RAM chips for a writable control store instead of a hardwired design.[1] A block diagram of the CPU/16 is shown in Figure One.

The design had 256 elements for each stack, and 256 opcodes with eight possible micro-instructions per opcode. Most instructions took three micro-cycles to execute, with subroutine calls and returns tying up the data bus to the exclusion of other operations. Figure Two shows the two instruction types supported: subroutine call and opcode. Thus, the importance of Forth's subroutine call was incorporated, but the rest of the design was dictated primarily by the constraints of fitting everything onto a single board while still using standard TTL components.

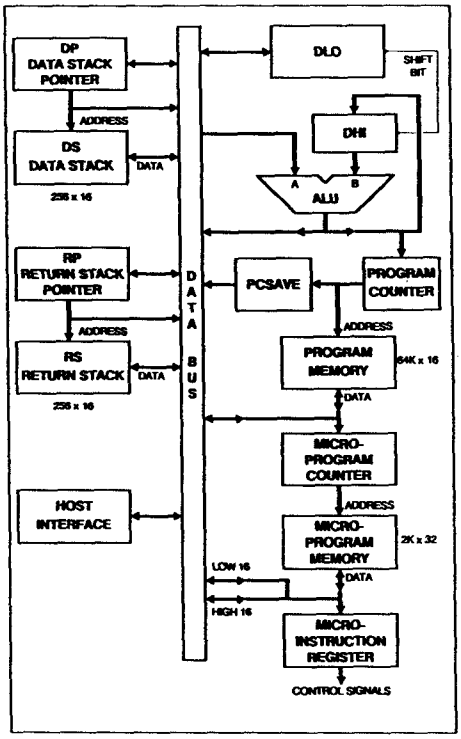

Figure One. WISC CPU/16 block diagram.

The RISC vs. CISC battle was about to take a new turn...

The Novix NC4000 chip had been introduced shortly before the WISC CPU/16 was built. A principle difference between the two designs (other than the fact that the Novix was a single chip compared to the CPU/16 discrete implementation) was that the Novix was a hardwired processor, while the CPU/16 was microcoded. The simplistic microcode implementation techniques used on the CPU/16 caused it to take an average of three micro-instructions for each opcode (at a cost of three clock cycles). At similar clock speeds (which translated into similar program memory speeds), one would have expected the NC4000 to outperform the CPU/16 by a factor of three to one.

But that didn't happen. Instead, the 4.77 MHz CPU/16 was much slower than a 5 MHz NC4000 on programs that used simple operations, but competitive (although, probably, not quite as fast) on programs that used more complex operations. This was because complex operations,

1. This decision was perhaps influenced by the fact that I did not possess an EPROM programmer, and that available programmable logic for use in synthesizing random logic was very modest in capabilities—and I didn't have a programmer for that either.

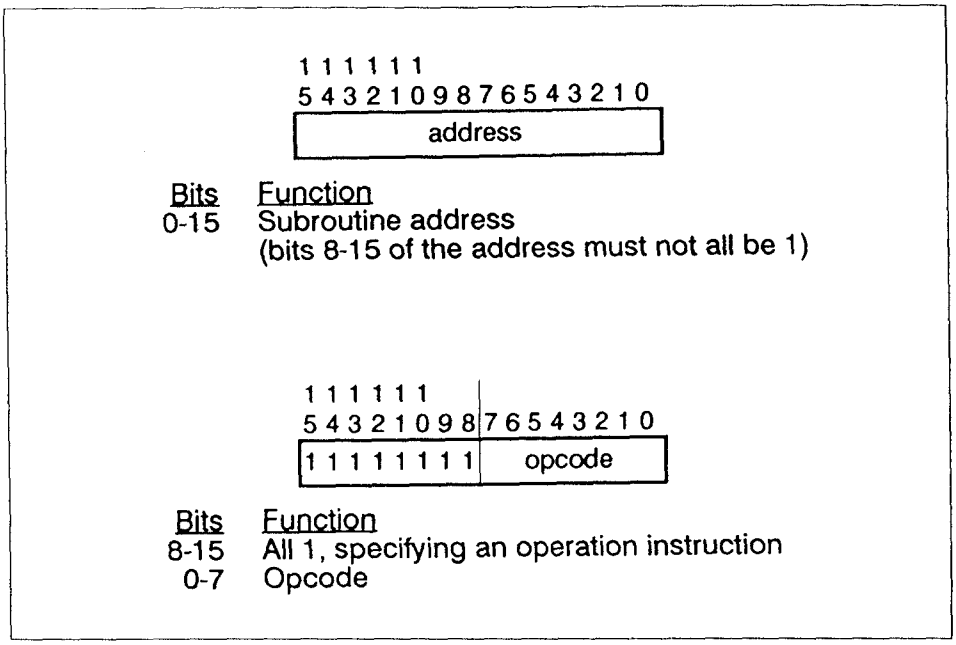

Figure Two. CPU/16 instruction formats.

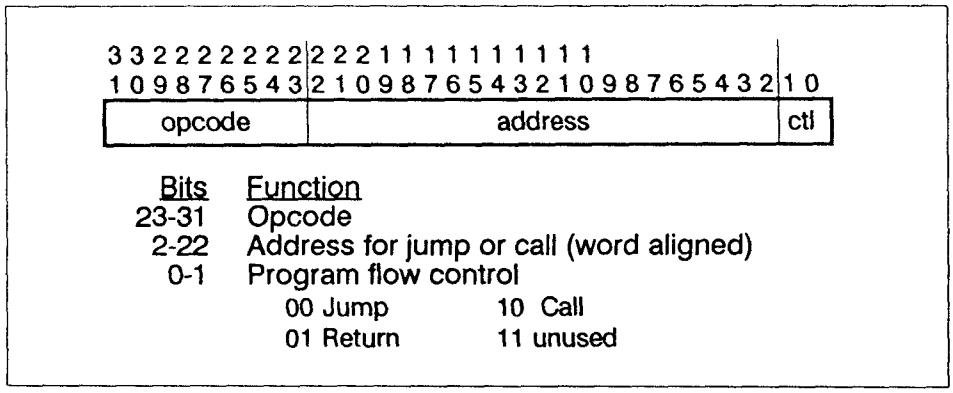

Figure Three. CPU/32 instruction format.

such as double-precision math and multi-element stack manipulations, were implemented in microcode in fewer clock cycles than the equivalent sequences in NC4000 assembly language. The execution speed for a mix of Forth primitives was just under one million typical Forth operations per second (including complicated operations such as multiply and double-precision math in a typical instruction mix).

As a result of my CPU/16 experience, I think microcoded techniques are inappropriate for a 16-bit Forth processor in most cases. Primarily, this is because the requirements for 32-bit wide microcode cause a single-chip implementation to be too large to be competitive with a hardwired approach. Also, the use of a microcoded approach does not provide many additional benefits when the processor is restricted to a 16-bit instruction format. However, the experience showed that something interesting was possible—microcoded machines could, perhaps, be competitive with hardwired machines with similar functions. This was because flexibility of operation and a high semantic content in each instruction could make up for a lack of raw speed. In other words, the RISC vs. CISC battle was about to take a new turn in the arena of stack computers.

The Monster/32

WISC Technologies produced a single prototype of a 32-bit computer that was seen by a very few people at the 1986 Rochester Forth Conference. In his book *The Mythical Man-Month* (Addison-Wesley, 1982), Fred Brooks describes what he calls the "second system syndrome." In this syndrome, the designer of a system saves up scores of neat ideas that can't be implemented in the first system because of time and money constraints. When the designer gets another crack at a similar problem (the second system), all these ideas are thrown in, usually with disastrous results.

The Monster/32 was my second system. The only truly good idea that was included was the decision to make it a 32-bit machine. Some of the ideas were reasonably good, but poorly executed. One idea was the inclusion of extra registers around the ALU. This eliminated congestion caused by having to save and restore the top-of-stack register when using the ALU for other calculations. Another idea was the addition of separate hardware to increment subroutine return addresses independent of the ALU.

The worst ideas had to do with the micro-instruction format and the use of multiple counters for addressing program memory. The 64-bit micro-instructions had a large number of interesting features, including the capability to specify a variable length for each micro-cycle. None of these features turned out to be very useful. The complexity of the micro-instruction format did result in almost impenetrable microcode that was very difficult to write and debug.

The Monster/32 was constructed using eight wire-wrapped boards in an S-100 card cage (but without using the S-100 bus in the usual manner). The wire-wrapping exercise itself taught me an important lesson about the value of simplicity, and wore out my first electric wire-wrap gun.[2] The system was eventually operational for a period of two weeks, and successfully ran a Forth system. The folks who saw it operate at the Rochester Forth conference never did ask why the attachment cable to the IBM PC host was only a foot long. There was an incredible noise problem in the host interface, and any longer cable wouldn't work reliably.

It became clear that, for a number of reasons, my first 32-bit design was a flop. Fred Brooks, again in *The Mythical Man-Month*, asserts that you should always be prepared to "throw one away." So we did.

2. Based on this experience, I rate battery-powered wire-wrap guns at about two miles of wire per gun.

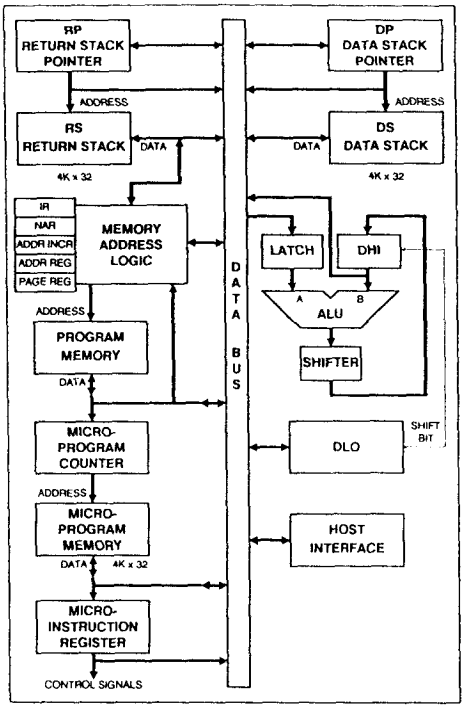

Figure Four. WISC CPU/32 block diagram.

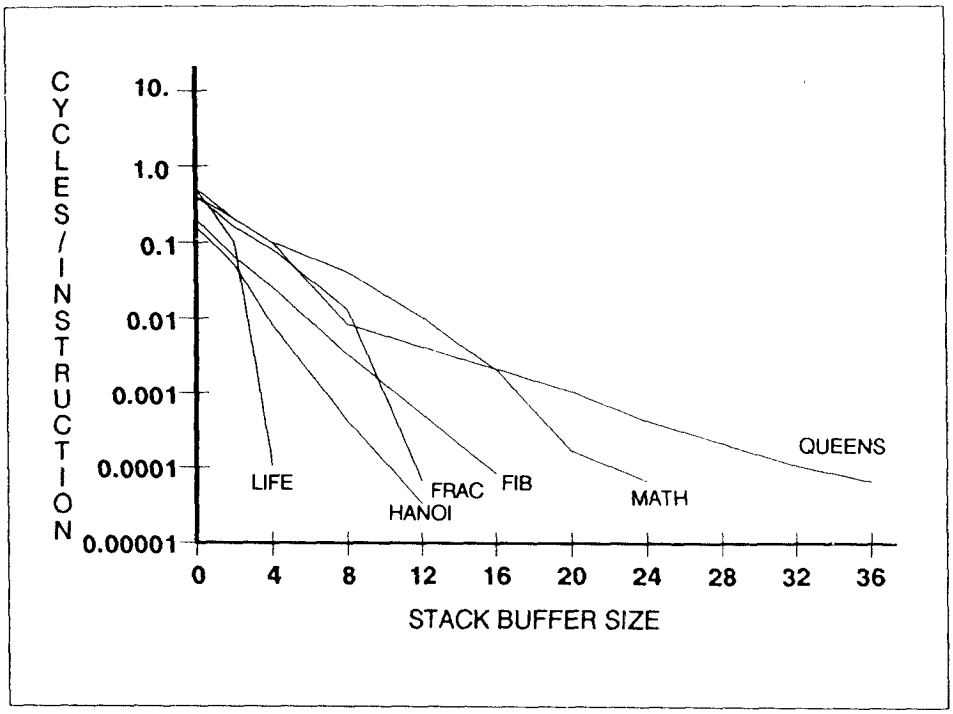

Figure Five. Return stack spilling overhead vs. stack buffer size.

The CPU/32

I began to distill the Monster/32 experience, and to decide what formed the true essence of an efficient WISC system. The CPU/16 had been arbitrarily constrained to simplicity, whereas the Monster/32 had been allowed to grow almost limitlessly. While there were a few good ideas to be salvaged, overall my immensely complex 32-bit design was a waste of good silicon. I began to see what I had missed in the realm of hardware design, despite my extensive experience with Forth: within limits, simpler *is* better.

At the same time, I began to combine several ideas that had been collecting in the back of my mind. One of them was that CPU cycle times can be made much faster than affordable memory speeds. Another was that taking advantage of concurrency in operations is a traditional way of speeding up computers that I had not exploited very well in previous designs. The last major idea was that, since microcoded stack machines only need eight or nine bits to specify an opcode, much of my 32-bit instruction memory was being wasted as unused bits in opcode-type instructions.

The answer to all my collected concerns hit like a bolt of lightning one day.[3] There were enough bits left over in an opcode instruction to also hold a large address, so why not make *every* instruction have both an opcode and a subroutine call? This had the effect of reducing program size, as well as providing for simultaneous operation of subroutine calls and opcodes. Thus, the resulting machine allowed control flow (subroutine calls and returns) to proceed in parallel with data manipulations (data stack operations), allowing two separate operations to be accomplished on each instruction. In other words, it offered the ideal situation for a Forth programmer: subroutine calls for free. Of course, in order to have a complete set of machine operations, a subroutine return format was required, which also combined an opcode with the return operation.

Not every instruction was a subroutine call or return, so there was a need for an instruction that incremented the program counter as well. In my quest to simplify the hardware, I made another discovery: the program counter was unnecessary. By using a jump instruction format instead of an increment-PC instruction format, I could have every instruction point to the next instruction to be executed (even if it was just the next sequential instruction). This reused the logic that performs subroutine calls, with a modification to suppress the push of the return address onto the return stack. The instruction format of the CPU/32 is shown in Figure Three.

Other enhancements to the CPU/32, based on experiences with the Monster/32 and the limitations of the CPU/16, included using a latch between the bus and the ALU to facilitate single-cycle exchange of data between the DHI register and the Data Stack. The microcode format was trimmed back to 32 bits, which makes microcode simple enough to be easily comprehensible, and saves a large amount of memory space. A block diagram of the CPU/32 is shown in Figure Four.

Another important insight in the design of the CPU/32 was the balance achieved between program memory speed and processor speed. RISC processors strive to execute one instruction per clock cycle. That implies that memory must be cycled as quickly as the clock in order to provide a steady stream of instructions. In a simple and streamlined processor, that means that programs must reside in fast memory. Usually, the required memory chips are so expensive that even high-end RISC systems must use them sparingly as cache memories. Many Forth applications have traditionally been in the areas of real-time

3. I don't remember just how the idea came to me. My best ideas usually come during my morning shower. However I was not electrocuted, so this one probably did not.

control. Many real-time control applications cannot afford the unpredictability of cache memory. Many others can't afford the cost of even a single bank of fast memory chips for any purpose. So, taking advantage of the fact that a microcoded machine can have a higher instruction semantic content (*i.e.*, it can accomplish more work per instruction), I designed the CPU/32 to execute an instruction every *two* micro-cycles, with each memory bus cycle taking two clock cycles. Assuming that both micro-cycles of every instruction are well employed, this allows twice the processing power for a given memory speed than an approach of one instruction per clock cycle.

The CPU/32 was originally built on reused S-100 boards from the Monster/32, with 74ALSxxx logic and some 74Fxxx logic for speed-critical sections. The use of "F" logic caused enough noise problems that the wire-wrapped version never ran at speed, so we produced a printed circuit board version before debugging was completed. This five-board version eventually ran at a 6 MHz micro-cycle rate, and executed approximately three million Forth operations per second.

The RTX 32P

The finished CPU/32 was demonstrated at the 1987 Rochester Forth Conference. At that conference, Harris Semiconductor was promoting its RTX 2000 processor, a redesign of the NC4000. They were intrigued by the possibilities for the CPU/32 as a 32-bit member of the RTX family. So, in July of 1987, I visited Melbourne Florida and transferred the schematics of the CPU/32 into their standard cell design system. In 31 days, the design was entered and verified with the help of one Harris engineer.[4] The product of this effort was, in January of 1988, an implementation that was functionally identical to the three printed circuit boards of the CPU/32 core processor, reduced to two chips operating at an 8.3 MHz micro-cycle rate. The two chips were the data chip (with the ALU, data stack, and half the microcode memory) and the control chip (with the memory addressing logic, the return stack, and the other half of the microcode memory).

The reason for a two-chip set instead of a single-chip processor implementation was to allow maximum flexibility with the finished system. 2K words of microcode memory were included on-chip, since 256 opcodes seemed to be more than I could possibly use.[5] When asked how big the stacks should be, I replied, "Gee, how much will you give me?" So, the chips ended up with 512 elements by 32 bits each for data and return stacks. This resulted in three things: it allowed Harris to make the biggest chip they have ever attempted, it made for a poor yield, and it produced chips which have logic on one quarter and memory in the other three. But, all these results were in keeping with the experimental nature of the project.

THE ANALYSIS PHASE

After the successful production of the CPU/32, I began to define and build a commercial version of the architecture for inclusion in the RTX product family. This exercise involved optimizing the architecture to fit the design constraints of CMOS chip technology as well as evolving the architecture to improve performance and better address the needs of the marketplace.

In the summer of 1987, I foolishly agreed to simultaneously refine the architecture for Harris and write a book about stack computer architecture. I did survive the summer, and found that the synergy between the two tasks was amazing. The book required me to think about measuring and describing the essence of stack machines. The design task required me to think about efficiency and architectural refinement. By the end of the summer, I had reached a number of conclusions about tradeoffs in stack machine design.

Stack Size

One of the big unknowns in producing the RTX 32P was how big to make the stacks. Before, I had been limited either by the need to keep chip count low or by standard high-speed memory chip sizes. On the RTX 32P, I guessed at 512 stack elements.

I guessed wrong. Simulations of several Forth programs show that many programs never used more than four or five stack elements. Of those that used more stack elements, all showed a small variability in stack size across reasonably large periods of time. In order to reduce hardware costs, it is advantageous to exploit this behavior and reduce on-chip stack sizes to the minimum possible.

An interesting line of thought to pursue is to assume that on-chip stacks are so expensive that they will be smaller than required. Also assume that there is some mechanism (say, a finite state machine that monitors stack overflows and underflows) that will copy elements to and from memory as required. The question to ask, then, is how much does this copying cost in terms of program performance degradation? Figure Five shows the results of a simulation for the return stack on a number of programs. The vertical axis indicates the amortized costs of stack spills in terms of wasted memory cycles per instruction executed in the course of the program. Notice that this axis has a logarithmic scale. The horizontal axis specifies the size of the on-chip stack buffer. The amazing thing is that, for a stack size of 16 elements, the cost is less than one percent. For a stack size of 16 to 32 elements, the cost reduces to essentially zero. Data stack behavior is similar.

The right answer, then, to how big stacks should be is 16 or 32 elements, no more. In the case of a multitasking environment, it is advantageous to have a partitioned stack that allocates 16 or 32 stack elements for each task in order to eliminate context-switching overhead.

Hardwired vs. Microcoded Performance

With the design of the RTX 32P, the hardwired control vs. microcoded control issue became ripe for detailed study. The RTX 2000 and the RTX 32P represent two processors designed to accomplish similar tasks using similar technology. One is hardwired, the other microcoded. The question is, which is faster?

I collected statistics on instruction execution frequency for Forth programs. But, I didn't simply gather numbers for the obvious primitives such as DUP, +, and SWAP. Instead, I took an IBM PC Forth compiler that was optimized to the point that anything worth speeding up was written in assembly language. This became my set of Forth "primitives"; that is, the basic building blocks used by real Forth code in real programs. Not surprisingly, these primitives included many double-precision operations (including "2-type" stack opera-

4. That includes the weekend I took off to visit Walt Disney World.

5. Of course this means that they were completely filled with mostly worthless junk almost immediately.

tions), and slow instructions such as multiply and divide. After I had measured the instruction execution frequencies for several programs, I multiplied the frequency times the number of clock cycles required for each of the RTX 2000 and RTX 32P processors. I assumed that RTX 2000 programs were operating on 16-bit data, and that RTX 32P programs were operating on 32-bit data. The result was surprising.

Despite the fact that most instructions on the RTX 2000 execute in a single clock cycle and that all instructions on the RTX 32P execute in two or more clock cycles, the RTX 32P required only ten percent more clock cycles than the RTX 2000 to do the same amount of work. In other words, clock-for-clock, the two processors did about the same amount of work. Part of the reason for the RTX 32P's good performance was the fact that its microcoded opcodes mapped well onto the high-level Forth operations used in real programs. Another part of the reason was that many of the subroutine calls counted as instructions, but were executed "for free" by the RTX 32P when combined with opcodes in the same machine instruction. Note that, although the program execution speed is similar, the RTX 32P accomplishes the same amount of work in half the memory accesses as the RTX 2000, since it accesses memory every two clock cycles. This difference allows it to use much slower memory for comparable processing speeds.

The result of this comparison is that it is not clear that the RISC approach of hardwired instructions and single-clock-cycle execution offers a compelling benefit over microcoded designs in terms of program execution speed for stack machines. This means that designing a 32-bit processor with hardwired control may result in suboptimal use of available memory bandwidth. For reasons previously stated, this should not be interpreted as meaning that 16-bit Forth chips should be designed with microcoded control—the area costs are just too high, and the lack of bits in the instruction format to support simultaneous opcode and subroutine call execution makes the potential payoff too low.

C—The Realities of the Marketplace

Forth is Good. But, Forth doesn't always Sell. The fact is, C is becoming the language of choice in many application areas, including real-time control. Also, architectural features required to support C go a long way towards supporting Ada for the military market. So, the RTX family is migrating to a position in which C is the primary language for many users. Forth then becomes the "assembly language" for the system, used for optimizing critical routines.

Aside from minor quirks of C (such as signed and unsigned characters, requiring optional signed byte extension on byte fetches), the only important C structure that is incompatible with Forth-based stack machines is the stack frame. C semantics *assume* that anything in the stack frame is addressable as a normal memory element. Furthermore, C stack frames grow too big to fit into any reasonably sized on-chip stack buffer. So, a stack processor must have some efficient method of supporting a stack frame. At a minimum, this means having a dedicated frame pointer on-chip, as well as the capability for using frame-pointer-plus-offset addressing. The RTX 2000 design incorporated a movable User Area pointer that can fulfill this requirement (an improvement over the NC4000, which had a fixed User Area location). The RTX 32P did not have this capability, but you can be assured that the commercial 32-bit RTX chip will.

For Forth users, the frame pointer can provide unexpected benefits. Many Forth programmers have advocated the use of local variables of some sort as a way of improving code organization and readability. A frame pointer mechanism makes an ideal implementation vehicle for a local variable stack, as well as providing a clean interface between C procedures and Forth subroutines.

Conclusions

I've described some of the history behind the sequence of processors leading up to the 32-bit RTX chip now in development. Along the way, I've tried to give some insight into why the processors have been designed the way they have, and into stack machine design issues in general. While the information has been presented as a personal history, it should provide some idea of the essential elements of designing stack computers.

In the real world, design of a good architecture is seldom done entirely through the sole use of wisdom and knowledge, and is never done right on the first try. Happenstance, and the background and education of the designer have much to do with the process. More important than the ability to get it right the first time is the ability to recognize mistakes, try new ideas, and retain the best of the old while incorporating the best of the new.

I would like to take this opportunity to acknowledge the involvement of two people without whom this history could not have taken place. Glen Haydon provided insight, encouragement, and financial support for the WISC Technologies processors. Dave Williams has been personally responsible for the acceptance and survival of the RTX 32-bit technology at Harris Semiconductor.

Philip Koopman Jr. is a senior scientist at Harris Semiconductor .