Architectural Opportunities For Future Stack Engines

Philip Koopman, Jr.

Harris Semiconductor
2525A Wexford Run Rd.
Wexford, PA 15090

Abstract

The next generation of stack computer designs must address the changing realities of the marketplace. Embedded real time control will become the primary application area, forcing design tradeoffs that are different from those made by other processors. Important issues that must be addressed by stack computer designers include: support for the C programming language, living with memory bandwidth limitations, and making the best possible use of large numbers of transistors.

Introduction

Today's marketplace will no longer accept a stack-based processor that is sold as a general-purpose CPU designed to run only Forth software. Times are changing. RISC processors have matured, and are starting a second generation. C is invading the domain of embedded control. Processor speeds are now faster than affordable memory. And, transistors are cheap enough that the compactness of a stack processor is not always an overwhelming advantage.

It is quite possible for stack-based processors to capture a significant share of the crowded CPU market. However, they won't do it by following strategies used in earlier days. Problems of competition from RISC systems, adequate support for C, faster memory speeds, and the availability of plentiful transistors are all changing the rules of the game. Future stack processors must address these issues, or be relegated to niches where ability to run Forth software quickly is a compelling advantage.

Embedded Real Time Control as an Application Area

Stack-based processors are not used in the major PCs or workstations. That is just as well, because today's stack CPUs make tradeoffs away from performance as general purpose processors in favor of the requirements of embedded real time control. With the dominance of first- and second-generation RISC processors for new workstation and PC designs (except those systems shackled to software compatibility with previous CPU generations), it would be a mistake to compete head-on with RISC and CISC processors. Instead, most applications for stack processors will be based on embedded real time control.

Common requirements for embedded real time control applications include strict limits on size, weight, power, cooling, reliability, and cost while demanding a high performance that has both determinacy and predictability **[KOOP90a]**. Stack processors are able to meet these requirements at low cost and with high performance **[KOOP90b]**. Therefore, we shall focus on embedded real time control as the application area of interest for the rest of this discussion.

Since embedded real time control favors design tradeoffs made to reduce overall system complexity, future stack machine designs will continue to support increased system integration and maximum performance with minimum hardware resources. Most importantly, common hardware-intensive tradeoffs used in RISC processors to increase speed will be avoided by stack processor designers. For example, support for dynamically managed cache memories will continue to be omitted. Also, Harvard architectures that increase memory fetch and store speed at the cost of doubling the amount of required processor pin bandwidth will be avoided.

Architectural Support for C Compilers

Assembly language used to be the premier language of embedded systems. However, that is changing. Today, one of the first questions asked by customers (even those currently using assembly language) is "how well does it run C?" The question is *not* "does it have a C compiler?" — the existence of C support is taken for granted. Any stack processor that does not have good support for C is simply not a contender.

At the architectural level, the C execution environment must be supported efficiently. As a minimum, this requires a dedicated frame pointer register that points to C activation records (*i.e.*, a register pointing to the memory-resident stack used for local variable storage by C). This frame pointer register is used frequently to read and write memory using base register plus offset addressing. Therefore, a quick way to read and write memory must be provided that uses the frame pointer plus a short literal value as the memory address.

Conventional wisdom dictates the use of a reasonably large register file for efficient execution of C programs. This wisdom is based on the fact that current C compiler technology uses intermediate code based on pseudo-register assignments. Also, advanced compiler technology (developed primarily for RISC processors) is able to use efficiently a large register file to reduce memory traffic to and from the activation record area in memory. So, one way to support C is to add a register set to the processor. The problem with this is that such registers cannot be as efficient as registers on a RISC machine, since stack machines do not typically allow for more than one literal that could be used as a register number in any instruction. Therefore, a register-to-register add that can be performed in one clock by a RISC machine could take three clock cycles on a stack machine with an auxiliary register set. At the same time, these registers reduce the advantages of stack computers by increasing the context to be saved on context switches.

I favor a different approach to C support for stack machines. Rather than use existing compiler technology (which was developed for register-rich RISCs), we must develop new compiler technology that keeps values on the data stack whenever possible. Preliminary experiments at Harris show that analysis similar to that performed by optimizing RISC compilers can be used to compile C code into code that resembles hand-written Forth programs for RTX processors. How efficient this code is compared to Forth remains to be seen, and it will no doubt be quite a while before this compiler technology is ready for general use. Nonetheless, the prospect of a C compiler that can optimally use the resources of a stack processor has great appeal compared to burdening an efficient stack architecture with a retrofitted register set.

There are several other minor issues for supporting C. Most Forth systems use the value -1 as a true flag, whereas C specifies a value of 1. Also, most Forth systems have used unsigned byte operators (*e.g.*, `C@`, `C!`) whereas the default for C is sign-extended characters. Stack architectures and compilers must somehow find compromises on these issues that do not significantly reduce system performance.

It is not likely that a stack processor can be faster at C than a RISC processor in the general case. The challenge is to give stack processors better C performance than a RISC system *of comparable cost and complexity*, and then excel on issues of interrupt performance, context switching speed, code size, better integration level, increased performance when using Forth as assembly language for inner loops, and application-specific concerns.

The Memory Bandwidth Problem

Memory bandwidth limitations seldom receive the attention they deserve. Increasing clock speeds cause problems with memory response times in two ways. At a first level, faster clock speeds mean that memory chips must be faster to have the same cycle time as the processor. As memory speeds edge down toward the 10 ns threshold, a secondary effect takes place: delays for driving addresses and data over pins become significant. With a 4 MHz part (250 ns memory cycle time), a 10 ns delay for driving an address bus is a minor consideration. With a 40 MHz part (25 ns memory cycle time), a 10 ns delay accounts for 40% of the bus cycle time! Great pains have been taken to optimize RISC processors to require a new instruction from memory on every clock cycle, putting relentless demands on the memory subsystem for instructions and data.

RISC processors are dealing with the memory bandwidth problem in two ways. First, they use a memory hierarchy that places most data in relatively slow and inexpensive DRAM chips. A comparatively small amount of high-speed cache memory (small for a RISC — typically 64K bytes) is then used to reduce average access time to memory. These cache chips must be a bit faster than the CPU's operating frequency in order to deliver one datum per clock cycle. This hierarchical approach reduces the amount of fast memory needed, and therefore system cost, at the expense of giving up execution time determinacy.

The second way RISC processors deal with the memory bandwidth problem is by using brute force. Harvard architectures, which use separate data paths to instruction and data caches, double the bandwidth to cache memory in exchange for better performance. With or without a Harvard architecture, the cache memory chips must be fast enough to keep up with the processor, and chip-to-chip signal propagation delays must be kept to a minimum. Therefore, some newer RISC chips use ECL RAM chips and pin driving circuitry. ECL is known for being extremely fast in both transistor switching speed and pin driving speed. It is also known for consuming large amounts of power and chip area.

Embedded applications are not likely to use ECL memories anytime soon. They are simply too expensive, power-hungry, and hot. Therefore, RISC manufacturers are introducing stripped-down versions of their processors that run on TTL-level SRAM and DRAM chips. In many cases, the SRAM chips used will be slower than cache-grade SRAM chips because of cost constraints. These stripped-down RISC processors then have a severe memory bandwidth problem. And, to make matters worse, any on-chip cache memory that would alleviate the problem must be disabled to gain determinacy for many applications.

Stack processors will also use TTL-level SRAM and DRAM chips for embedded control. Therefore, the question becomes one of how stack processors can gain a speed advantage with limited memory bandwidth. One simple technique that works on any stack processor is to make more effective use of a statically allocated bank of fast memory. The smaller program size of a stack processor increases the likelihood that an entire application will fit into a single bank of fast SRAMs, whereas a RISC processor would need to keep much of the program in slower DRAMS. Also, even for larger programs, quick subroutine calls allow stack processors to keep critical code sections in fast memory while other sections are in slow memory.

With the coming of 32-bit stack processors, another method becomes available for improving the amount of work done given a limited memory bandwidth. A 32-bit instruction is typically big enough to hold two or more stack machine instructions. The RTX 4000 design performs two separate operations with most instruction formats **[KOOP89]**. The ShBoom design holds between one and four operations in each instruction word **[PTAC90]**. A stack processor that executes more than one operation for each instruction word fetch can be optimized to perform multiple clock cycles per memory access, whereas RISC systems are optimized for a single clock cycle per memory access. Thus, *if memory bandwidth is the limiting factor to system performance*, it is possible for a stack machine to execute significantly more instructions per second than a RISC processor.

Making Use of Cheaper Silicon

Recently we have seen the introduction of several RISC and CISC processors with more than one million transistors on a chip. These gargantuan CPUs are quite expensive compared to a stack processor, yet they remain popular. Why?

The key to the maximum acceptable size of a chip is the cost of the CPU relative to total system cost. Large, expensive CPU chips are acceptable in a workstation design where CPU costs account for only one-tenth of the system cost. In such systems, CPU costs are dwarfed by the cost of DRAM, which typically accounts for one-half the system cost **[HENN90]**. In embedded systems, memory requirements and the costs for other components are often smaller, and CPU costs are often a significant portion of total system costs. Therefore, CPU die size and cost becomes an important issue in embedded control systems, favoring the use of small, inexpensive CPU chips.

On the other hand, there is no doubt that an increased number of transistors will become available for stack computer chips in the future. Because defect rates are exponential with die size, there is a knee in the yield curve below which smaller die size is not particularly advantageous for improving yield. For very small chips, the number of bonding pads and packaging costs can easily become dominant factors in chip cost. If useful functions can be added to the CPU chip while keeping the die size small enough, significant system cost savings can be realized.

The best features to add to an embedded control CPU chip are *not* simply brute-force techniques to increase speed. The best features reduce overall system complexity while improving system performance. The best additions improve the integration level of the system. Adding timers, counters, I/O devices, and memory control logic to the CPU chip can reduce the number of the components on the system significantly. Application-specific versions of a CPU can replace custom hardware on a circuit board with on-chip logic, often reducing complexity while increasing system speed. These techniques are already being used by the Harris RTX 2000 processor family, and will be emphasized even more in the future.

As a last resort, memory can be added to the CPU to improve available memory bandwidth for executing programs. This is not the first thing to use chip area for, however, because RAM takes up a lot of chip space, and the effectiveness of on-chip RAM is limited if the entire application program will not fit on the CPU chip. One attractive alternative is to provide on-chip ROM for application-specific versions of a CPU. ROM is much denser than RAM, so reasonable amounts of ROM can be placed on a 16-bit processor chip. This ROM would either execute an entire application program, or contain a toolbox of routines that execute quickly.

Conclusions

Stack computers still have a bright future in embedded real time control applications. There are several important developments required to make this potential a reality. The performance of C on stack machines must be improved through a combination of architectural adjustments and better compiler technology. Architectures must be tuned to take advantage of the restricted memory bandwidth available on low-end systems. And, newly available "cheap silicon" must be used wisely to improve overall system cost-effectiveness and performance. Current and future Harris RTX chips incorporate features to exploit advantages in all these areas to maintain their edge for embedded real time control.

References

[HENN90] Hennessy, J., Patterson, D., *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, San Mateo CA, 1990.

[KOOP90a] Koopman, P., "Design constraints on embedded real time control systems," Majithia (Ed.), *Systems Design & Networks Conference, Microprocessor Track*, May 8-10, 1990, Santa Clara, CA, pp. 71-77.

[KOOP90b] Koopman, P., "Modern stack computer architecture," Majithia, (Ed.), *Systems Design & Networks Conference, Microprocessor Track*, May 8-10, 1990, Santa Clara, CA, pp. 153-164.

[KOOP89] Koopman, P., VanNorman, R., "RTX 4000," *Proceedings of the 1989 Rochester Forth Conference*, pp. 84-86.

[PTAC89] P.T. Acquisitions, *ShBoom Microprocessor Applications Manual*, System Insights, Austin TX, 1989.