MODERN STACK COMPUTER ARCHITECTURE

Philip J. Koopman Jr.
Senior Scientist
Harris Semiconductor
2525A Wexford Run Rd.
Wexford, PA 15090

ABSTRACT

A new generation of stack processors based on the Forth abstract machine has recently been developed for embedded real-time control applications. These processors are optimized for minimum system complexity, small program size, fast but consistent processor performance, and excellent response to external events. The Harris RTX 2000 is described as an example of a typical stack processor.

INTRODUCTION

Hardware-supported Last In First Out (LIFO) stacks have been used on computers since the late 1950's. Originally, these stacks were used to increase the execution efficiency of high level languages such as ALGOL. Since then they have fallen in and out of favor with hardware designers, eventually becoming incorporated as a secondary data handling structure in most computers.

With the introduction of VLSI microprocessors, conventional methods of computer design have been questioned once again. Complex Instruction Set Computers (CISCs) have evolved into complicated processors with comprehensive instruction sets. Reduced Instruction Set Computers (RISCs) have challenged this approach by using simplified processing cores to achieve higher raw processing speeds for some applications. However, the next generation of RISC processors, based on superscaler instruction dispatching technology, promises in its own way to be every bit as complex as CISC designs.

Once more the time has come to consider stack machines as an alternative to other design styles. For some application areas they offer a better set of speed and complexity tradeoffs than RISC or CISC register-based processors. New stack machine designs based on VLSI design technology provide additional benefits not found on previous stack machines. A key benefit is the ability to include substantial stack memories on the same chip as the CPU, thereby eliminating the need to consume precious memory bus cycles shuffling stack elements.

The first successful application area for stack machines has been in real-time embedded control environments. Stack machines offer processor complexity that is much lower than that of CISC machines, and overall system complexity that is lower than that of either RISC or CISC machines. Stack machines provide extremely fast subroutine calling capability and superior performance for interrupt handling and task switching. They execute programs at competitive speeds without reliance on techniques such as caching and scoreboarding that result in non-deterministic execution times in real-time applications. When put together, these traits create computer systems that are fast, nimble, and compact.

This paper starts with a discussion of the RTX 2000 stack processor as a concrete example of a modern stack architecture. It then discusses the architectural properties of stack processors in depth, including comparisons to RISC and CISC characteristics. and why they are well suited to real-time embedded control applications.

THE RTX 2000 AS AN EXAMPLE

Modern stack architectures are distinguished from earlier stack architectures, such as the Burroughs family of processors (Earnest 1980) by a number of architectural features. The older stack architectures are based on the execution environment used with block-structured languages such as ALGOL and Pascal. Thus, they have a single stack that is largely resident in program memory. Furthermore, while they often perform computations on a push-down stack, traditional activation record structures on this same stack are used for subroutine linkage.

The newer stack architectures have a distinctly different heritage. They are primarily based on the Forth abstract machine computation model (Kogge 1982, Moore 1980). This computation model includes a requirement for two stacks: a data stack and a return stack. The data stack is used both for expression evaluation and for subroutine parameter passing. The return stack is used for subroutine return address storage and for loop counter storage. When these new architectures execute languages that require stack frames for procedure linkage, they use a separate frame pointer that addresses a data structure in memory independent of the two hardware stacks. While the emphasis on older stack machines was on providing a very large stack buffer in program memory, newer stack machines provide on-chip buffers for both stacks, greatly reducing the memory bandwidth bottleneck and increasing execution speed. The Forth computational model also places a great emphasis on the ability to perform subroutine calls quickly, which results in hardware implementations typically implementing single-cycle subroutine calls and returns.

Several Forth-based stack architectures have been developed since 1985. Among the commercial implementations are the: Novix NC4016 (Miller 1987), MISC M17 (MISC 1988), WISC CPU/16 (Koopman 1987), WISC CPU/32 (now called the Harris RTX 32P) (Koopman 1988), Johns Hopkins FRISC 3 (now called the Silicon Composers SC32) (Hayes et al. 1987), and the Harris Semiconductor RTX 2000 family (Harris Semiconductor 1988). Other architectures are discussed by Koopman (1989). It is anticipated that more architectures will be announced in the next year or two.

Before discussing the advantages of newer stack architectures, it is useful to have an example architecture for discussion. The Harris RTX 2000 is a 16-bit stack processor that is an architectural descendent of the Novix NC4016, the first single-chip Forth computer. The current implementations of the RTX 2000 use 1.5 micron feature size standard cell CMOS technology and run at a 10 MHz clock rate.

Figure 1 shows that the RTX 2000 has characteristics typical of a Forth-derived stack processor. The RTX 2000 has two on-chip hardware stacks: the data stack and the return stack. The sizes of these on-chip stacks varies among implementations, but ranges from 64 words for each stack in the RTX 2001A to 256 words each in the RTX 2000. Studies have shown that this is an ample amount of storage for most applications (Koopman, 1989).

The ALU section contains a 2-element buffer for the top elements of the data stack. The T register holds the Top data stack element, and N (Next) register holds the second-from-top data stack element. There is also a special MD register for support of multiplication and division as well as an SR register for fast integer square roots. The ALU may perform operations on the T register and any one of the N, MD, or SR registers.

The I register is used as a buffer for the top element of the Return Stack. Since Forth keeps loop counters as well as subroutine return addresses on the Return Stack, the Index register can be decremented to implement countdown loops efficiently.

The UBAR (User area Base Address Register) may be used as a frame pointer for addressing memory-resident activation record frames in support of conventional languages such as C. On different RTX 2000 family implementations there are also various functional blocks for byte swapping within 16-bit words, interrupt control, stack overflow/underflow control, timers, counters, memory page registers, a separate off-chip data

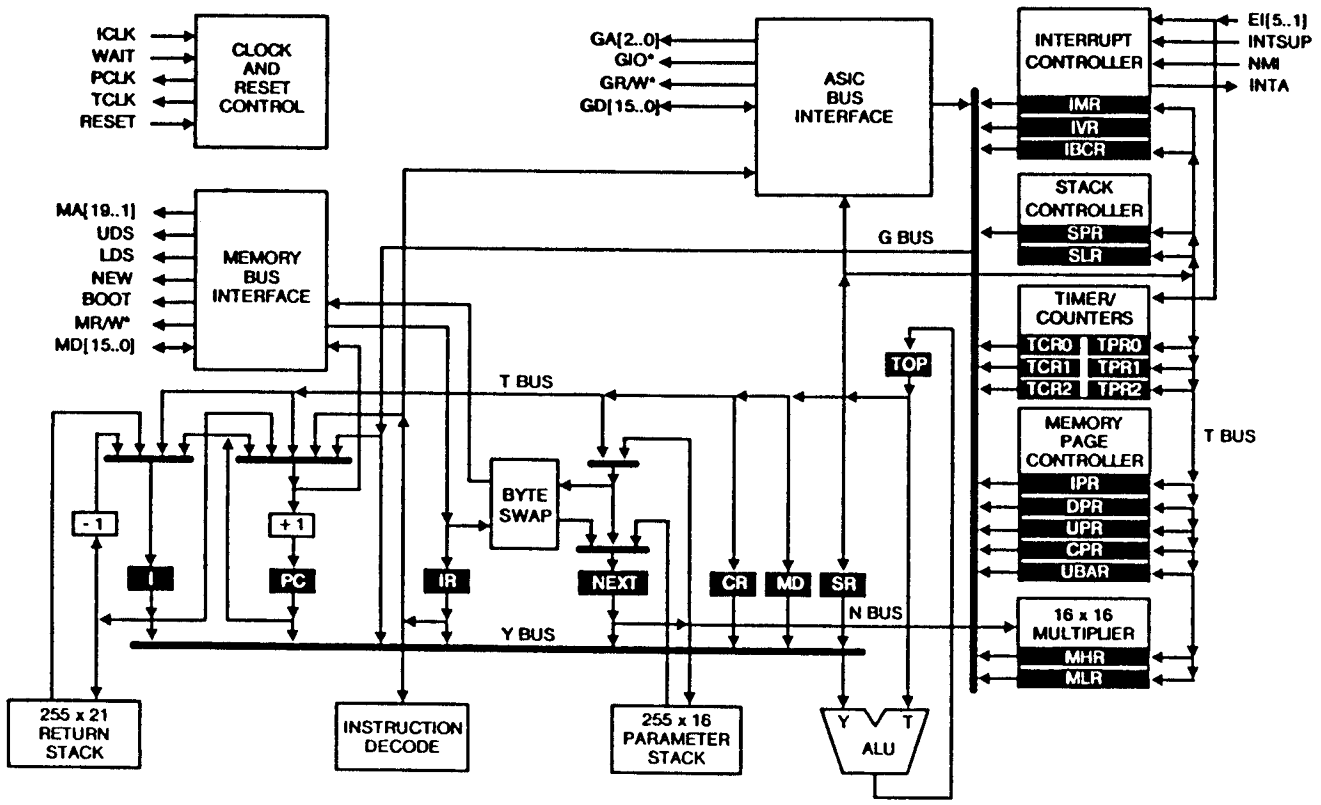


Figure 1.
Harris RTX 2000 processor block diagram.

path for I/O transfers, and a hardware multiplier.

A key to the RTX 2000 design is that it uses a partially decoded 16-bit instruction format. If the highest bit of the instruction is 0, then the instruction is a subroutine call with a 15-bit target address. Otherwise, the instruction is in one of a few formats in which each group of bits controls a resource of the CPU. The RTX 2000 is able to execute almost all instructions in a single clock cycle, including subroutine calls. Memory loads and stores take two clock cycles, since there is only a single memory data bus that is shared between instruction and data accesses. The RTX 2000 is also able to execute subroutine return operations in parallel with most instructions (*i.e.*, subroutine returns are "free"), because there is a separate bit dedicated to indicating subroutine return operations in most instruction formats.

AN OVERVIEW OF EMBEDDED CONTROL REQUIREMENTS

In order to properly evaluate the architectural qualities of stack computers, we must first set the context for comparison. Stack computers are primarily used in embedded real-time control applications. The requirements of these applications are very different from the requirements for engineering workstation applications and general-purpose mainframe programs upon which most popular benchmarks are based.

Most embedded systems place severe constraints on the processor for size, weight, power, cooling, performance, reliability, and cost. This is because the processor is just a component of a larger system, which has its own operating requirements and manufacturing constraints. This means that a need for high speed performance cannot always be solved just by buying the latest high-speed CPU.

Size and Weight

Size and weight restrictions can greatly limit the complexity of the hardware which can be brought to bear on solving a particular embedded control problem. A typical embedded controller may have a size budget of a few cubic inches, and a weight budget of a few pounds (including the power supply).

The problem with size and weight restrictions is that high performance processing systems tend to be larger and heavier than slower systems. At the CPU level, a processor that has a large number of pins takes up scarce printed circuit board area. At the system level, a design that needs cache memory controller chips and large amounts of memory takes even more printed circuit board area.

The key to the size and weight issue is to keep the component count small. The smallest systems are those which do not require external support chips (especially cache memory support), and which allow small, efficient encodings of programs.

Power and Cooling

The processor complexity can affect the power consumption of the system. High power consumption increases the size and weight requirements of the power supply. Since essentially all power consumed by a computer is eventually given off as heat, increased power consumption results in increased cooling requirements.

The amount of power used by the processor is related to the number of transistors and pins on the processor chip. Processors that rely on exotic process technology for speed usually require more power than is available in a tightly constrained embedded system. Processors that need large numbers of high speed memory devices likewise can exceed a power budget.

The key to the power and cooling issue is to minimize system complexity to reduce the number of transistors in the system that can consume power. Also, the integration level should be kept as high as possible to minimize the number of pins in the system.

Computing Performance

Computing performance in a real-time embedded control environment is not simply an instructions-per-second rating. While raw computational performance is important, other vital factors include interrupt response characteristics, context switching overhead, and I/O performance. Because real-time tasks are characterized by frequent, often unpredictable events, the performance of a processor at handling interrupts is crucial to its suitability for real-time processing. Since a control application usually involves a reasonable amount of communication with the outside world (such as reading sensors and activating control circuits), good I/O performance is also important.

A key consideration for more difficult real-time control applications is predictability. For real-time control with hard deadlines, a designer must be able to predict with absolute certainty the running time of the piece of code responding to an external event. If a system has elements of statistical performance, such as cache memory, the designer must be extremely pessimistic about the performance of these features, and plan on the worst case.

Reliability

Embedded processing applications are notorious for extreme operating conditions, especially in automotive and military equipment. The processing system must deal with vibration, shock, extreme heat and cold, and perhaps radiation. In remotely installed applications, such as spacecraft and undersea applications, the system must be able to survive without field service technicians to make repairs.

The general rule to avoiding problems caused by operating environments is to keep the component count and number of pins as low as possible. Also helpful is keeping the system as cool as possible to inhibit aging of the system's components.

Cost

The cost of the processor itself may be very important to low- and medium-performance systems, especially consumer electronics products. Since the cost of a chip is related to the number of transistors and to the number of pins on the chip, low complexity processors have an inherent cost advantage.

In high-performance systems, the cost of the processor may be overwhelmed by the cost of the multi-layered printed circuit boards, support chips, and high-speed memory chips. In these cases, overall system complexity must be reduced to keep system costs down. It may be attractive to use a standard cell design with a CPU macro-cell and custom peripheral cells, if this reduces overall system complexity.

ARCHITECTURAL INSIGHT INTO STACK MACHINES

Total system performance includes not only raw execution speed, but also total system cost and system adaptability when used in real world applications. The execution speed component of system performance includes not only how many instructions can be performed per second on straight line code, but also speed in handling interrupts, context switches, and performance degradation due to factors such as procedure calls.

Program Size

A popular saying is that "memory is cheap." Anyone who has watched the historically rapid growth in memory chip sizes knows the amount of memory available in a system can be expected to increase dramatically with time. Yet, somehow the problems that computers are called on to solve seems to keep up with the supply of memory, so in many systems memory space is still a limiting factor. Further aggravating the situation is the widespread use of high level languages for all phases of programming. This results in bulkier programs, but of course improves programmer productivity.

The amount of program memory available for an application is fixed by the economics of the actual cost of the memory chips and printed circuit board space. It is also affected by mechanical limits such as power, cooling, or the number of expansion slots in the system (limits which are also based on economics). Small program sizes reduce memory costs, component count, and power requirements, and can improve system speed by allowing the cost-effective use of smaller, higher speed memory chips. Embedded microprocessor applications are very sensitive to the costs of printed circuit board space and memory chips, since these resources form a substantial proportion of all system costs (Ditzel *et al.* 1987).

The traditional solution for a growing program size is to employ a hierarchy of memory devices with a series of capacity/cost/access-time tradeoffs. Typical members of this hierarchy are on-chip register files, cache memory, DRAM program memory, and hard disk for long-term storage.

A problem with cache memory is that it must be big enough to hold enough program fragments long enough for eventual reuse to occur. Another problem is that the non-determinism introduced by cache misses can create problems when dealing with time-critical sequences in a control system. For these two reasons, cache memory with acceptable performance for RISC and many CISC processors must often be larger than can be afforded for a particular system.

Davidson and Vaughan (1987) suggest that RISC computer programs can be up to 2.5 times bigger than CISC versions of the same programs (although other sources, especially RISC vendors, would place this number at perhaps 1.5 times bigger.) They also suggest that the RISC computers need a cache size that is twice as large as a CISC cache to achieve the same performance. Furthermore, a RISC machine with twice the cache of a CISC machine will still generate twice the number of cache misses (since a constant miss ratio generates twice as many misses for twice as many cache accesses), resulting in a need for higher speed main memory devices as well for equal performance. This is corroborated by the rule of thumb that a RISC processor in the 10 MIPS (Million Instructions Per Second) performance range needs 128K bytes of cache memory for satisfactory performance, while high end CISC processors typically need no more than 64K bytes.

Stack machines have much smaller programs than either RISC or CISC machines. Stack computer programs can be 2.5 to 8 times smaller than CISC code (Harris 1980, Ohran 1984, Schoellkopf 1980), although there are some limitations to this observation having to do with the instruction set and programs executed. This means that a RISC processor's cache memory may need to be bigger than a stack processor's entire program memory to achieve comparable average memory response times.

Small program size on stack machines not only decreases system costs by eliminating memory chips, but can actually improve system performance. This effect happens by increasing the chance that an instruction will be resident in high speed memory when needed, possibly by using the small program size as a justification for placing an entire program in fast memory. Also, placing an entire program in high-speed memory greatly improves consistency of program execution by eliminating the possibility of cache misses entirely.

How can it be that stack processors have such small memory requirements? There are two factors that account for the extremely small program sizes possible on stack machines. The more obvious factor, and the one usually cited in the literature, is that stack machines have small instruction formats. Conventional architectures must specify not only an operation on each instruction, but also operands and addressing modes. For example, a typical register-based machine instruction to add two numbers together might be: **ADD R1,R2** . This instruction must not only specify the **ADD** opcode, but also the fact that the addition is being done on two registers, and that the registers are **R1** and **R2** .

On the other hand, a stack-based instruction set need only specify an **ADD** opcode, since the operands have an implicit address of the current top of stack. The only time that an operand is present is when performing a load or store instruction, or pushing an immediate data value onto the stack.

A less obvious, but actually more important reason for stack machines having more compact code is that they efficiently support code with many frequently reused subroutines, often called threaded code (Bell 1973, Dewar 1975). While such code is possible on conventional machines, the execution speed penalty is severe. In fact, one of the most elementary compiler optimizations for both RISC and CISC machines is to compile procedure calls as in-line macros. This, added to most programmers' experience that too many procedure calls on a conventional machine will severely degrade program performance, leads to significantly larger programs on conventional machines.

Stack oriented machines are built to support procedure calls efficiently. Since all working parameters are always present on a stack, procedure call overhead is minimal, requiring no memory cycles for parameter passing. On most stack processors, procedure calls take one clock cycle, and procedure returns take zero clock cycles in the frequent case where they are combined with other operations.

Processor and System Complexity

When speaking of the complexity of a computer, two levels are important: processor complexity and system complexity. Processor complexity is the amount of logic (measured in chip area, number of transistors, etc.) in the actual core of the processor that does the computations. System complexity considers the processor embedded in a fully functional system which contains support circuitry, the memory hierarchy, and software.

CISC computers have become substantially more complex over the years. This complexity arises from the need to be very good at all their many functions simultaneously. A large degree of their complexity stems from an attempt to tightly encode a wide variety of instructions using a large number of instruction formats. Added complexity comes from their support of multiple programming and data models.

The complexity of CISC machines is partially the result of encoding instructions to keep programs relatively small. The goal is to reduce of the semantic gap between high level languages and the machine to produce more efficient code. Unfortunately, this may lead a situation where almost all available chip area is used for the control and data paths (for instance the first three generations of the Motorola 680x0 and Intel 80x86 products). Additionally, an argument made by RISC proponents is that CISC designs may be paying a performance penalty as well as a size penalty.

The concept behind RISC machines is to make the processor faster by reducing its complexity. To this end, RISC processors have fewer transistors in the actual processor control circuitry than CISC machines. This is accomplished by having simple instruction formats and instructions with low semantic content. The instruction formats are usually chosen to correspond with requirements for running a particular programming language and task, typically integer arithmetic in the C programming language.

This reduced processor complexity is not without a substantial cost. Most RISC processors have a large bank of registers to allow quick reuse of frequently accessed data. These register banks must be multi-ported memory (allowing simultaneous accesses at different addresses) to allow fetching both source operands and storing a result on every cycle. Furthermore, because of the low semantic content of their instructions, RISC processors need much higher memory bandwidth to keep instructions flowing into the CPU. This means that substantial on-chip and system-wide resources must be devoted to cache memory to attain acceptable performance. Also, RISC processors characteristically have an internal instruction pipeline. This means that special hardware or compiler techniques must be employed to manage the pipeline.

Different RISC implementation strategies make significant demands on compilers such as: scheduling pipeline usage to avoid hazards, filling branch delay slots, and managing allocation and spilling of the registers. While the decreased complexity of the processor makes it easier to get bug-free hardware, even more complexity shows up in the compiler. This makes compilers complex as well as expensive to develop and debug. The reduced complexity of RISC processors comes with an offsetting (perhaps even more severe) increase in system complexity.

Stack machines strive to achieve a balance between processor complexity and system complexity. Stack machine designs realize processor simplicity not by restricting the number of instructions, but rather by limiting the data upon which instructions may operate: all operations are on the top stack elements. In this sense, stack machines are "*reduced operand set computers*" as opposed to "reduced instruction set computers."

Limiting the operand selection instead of how much work the instruction may do has several advantages. Instructions may be very compact, since they need specify only the actual operation, not where the sources are to be obtained. The on-chip stack memory can be single ported, since only a single element needs to be pushed or popped from the stack per clock cycle (assuming the top two stack elements are held in registers.) More importantly, since all operands are known in advance to be the top stack elements, no pipelining is needed to fetch operands. The operands are always immediately available in the top-of-stack registers. As an example of this, consider the T and N registers in the RTX 2000 design, and contrast these with the dozens or hundreds of randomly accessible registers found on a RISC machine.

Stack machines are extraordinarily simple: 16-bit stack machines such as the RTX 2000 typically use only 20 to 35 thousand transistors for the processor core. In contrast, the Intel 80386 chip has 275 thousand transistors and the Motorola 68020 has 200 thousand transistors. Even taking into account that the 80386 and 68020 are 32-bit machines, the difference is significant.

Stack machine compilers are also simple, because instructions are very consistent in format and operand selection. In fact, most compilers for register machines go through a stack-like view of the source program for expression evaluation, then map that information onto a register set. Stack machine compilers have that much less work to do in mapping the stack-like version of the source code into assembly language. Forth compilers, in particular, are well known to be exceedingly simple and flexible.

Stack computer systems are also simple as a whole. Because stack programs are so small, exotic cache control schemes are not required for good performance. Typically the entire program can fit into cache-speed memory chips without the complexity of cache control circuitry.

In those cases where the program and/or data is too large to fit in affordable memory, a software-managed memory hierarchy can be used: frequently used subroutines and program segments can be place in high speed memory, while infrequently used program segments are placed in slow memory. Inexpensive single-cycle calls to the frequent sections in the high speed memory make this technique very effective, yet provide complete determinism in program execution for embedded control systems.

Stack machines, therefore, achieve reduced processor complexity by limiting the operands available to the instruction. This does not force a reduction of the number of potential instructions available, nor does it cause an explosion in the amount of support hardware and software required to operate the processor.

Processor Performance

Processor performance is a very tricky area to talk about. Untold energy has been spent debating which processor is better than another, often based on sketchy evidence from questionable benchmarks. While some progress has been made on workstation benchmarks, embedded control benchmarks are still in their infancy.

The problem of finding exact performance measures is not going to be resolved here. Instead, we shall concentrate on a discussion of some reasons why stack machines can be made to go faster than other types of machines on an instruction-by-instruction basis, why stack machines have good system speed characteristics, and what kinds of programs stack machines are well suited to.

Instruction Execution Rate

Most first-generation RISC processors strive for an instruction execution rate of one instruction per processor clock cycle. This is accomplished by pipelining instructions into some sequence of instruction address generation, instruction fetch, instruction decode, data fetch, instruction execute, and data store cycles as shown in Figure 2a. This breakdown of instruction execution accelerates overall instruction flow, but introduces a number of problems. The most significant of these problems is management of data to avoid hazards caused by data dependencies. This problem comes about when one instruction depends upon the result of the previous instruction. This can create a problem, because the second instruction must wait for the first instruction to store its results before it can fetch its own operands. There are several hardware and software strategies to alleviate the impact of data dependencies, but none of them completely solves it.

Stack machines can execute programs as quickly as RISC machines, perhaps even faster, without the data dependency problem. It has been said that register machines are more efficient than stack machines because register machines can be pipelined for speed while stack machines cannot. This problem is caused by the fact that each instruction depends on the effect of the previous instruction on the stack. The whole point is, however, that stack machines *do not need* to be pipelined to get the same speed as RISC machines.

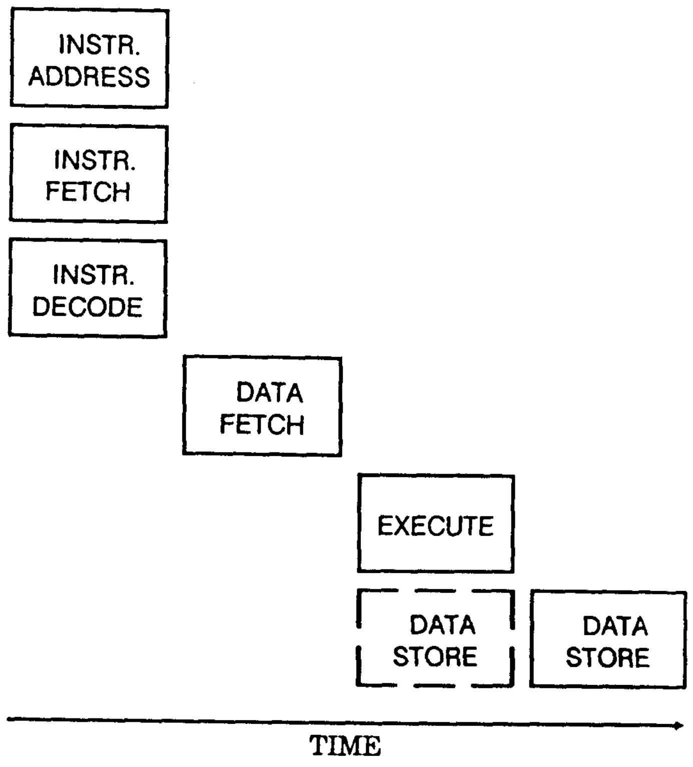
Consider how the RISC machine instruction pipeline can be modified when it is redesigned for a stack machine. Both machines need to fetch the instruction, and on both machines this can be done in parallel with processing previous instructions. For convenience, we shall lump this stage in with instruction decoding.

In the next step of the pipeline, the major difference becomes apparent. RISC machines must spend a pipeline stage accessing operands for the instruction after (at least some of) the decoding is completed. A RISC instruction specifies two or more registers as inputs to the ALU for the operation. A stack machine does not need to fetch the data; they will be waiting on top of the stack when needed. This means that the stack machine can dispense with the operand fetch portion of the pipeline.

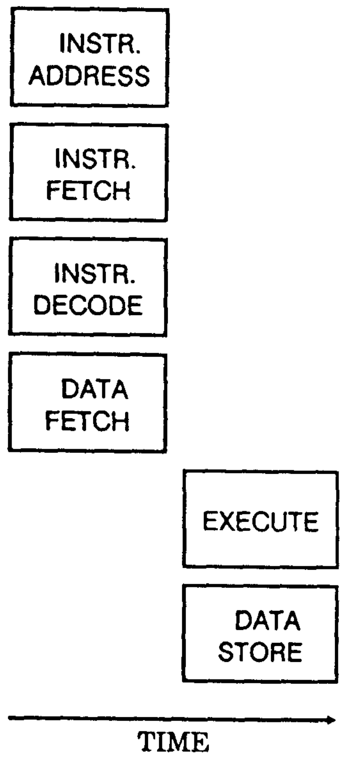
The instruction execute portion of both the RISC and stack computers are probably about the same since the same sort of ALU can be used by both systems.

The operand storage phase takes another pipeline stage in some RISC designs, since the result must be written back into the register file. This write conflicts with reads that need to take place for new instructions beginning execution, causing delays or the need for a triple-ported register file. Conversely, the stack machine simply deposits the ALU output result in the top-of-stack register and is done. An additional problem is that extra data forwarding logic must be provided in a RISC machine to prevent waiting for the result to be written back into the register file if the ALU output is needed as an input for the next instruction. A stack machine always has the ALU output available as one of the implied inputs to the ALU.

Figure 2b shows that stack machines need only a two-stage pipeline: instruction fetch and instruction execute. Figure 2a shows that RISC machines need at least three pipeline stages and perhaps four to maintain the same throughput: instruction fetch, operand fetch, and instruction execute/operand store. Also, we have noted that there are several problems inherent with the RISC approach, such as data dependencies and resource



a) pipelined RISC computer



b) stack computer

Figure 2.
Instruction execution phases for RISC and stack computers.

contention, that are simply not present in the stack machine.

What this all means is that there is no reason that stack machines should be any slower than RISC machines in executing instructions, and there is a good chance that stack machines can be made faster and simpler using the same fabrication technology.

System Performance

System performance is even more difficult to measure than raw processor performance. System performance includes not only how many instructions can be performed per second on straight-line code, but also speed in handling interrupts, context switches, and system performance degradation because of factors such as conditional branches and procedure calls. Approaches such as the Three-Dimensional Computer Performance technique (Rabbat *et al.* 1988) are better measures of system performance than the raw instruction execution rate.

RISC and CISC machines are usually constructed to execute straight-line code as the general case. Frequent procedure calls can seriously degrade the performance these machines. The cost for procedure calls not only includes the cost of saving the program counter and fetching a different stream of instructions, but also the cost of saving and restoring registers, arranging parameters, and any pipeline bubbles that may occur. The very existence of a stack computer structure called the Return Address Stack should imply how much importance stack machines place upon flow-of-control structures such as procedure calls. Since stack machines keep all working variables on a hardware stack, the setup time required for preparing parameters to pass to subroutines is very low, usually a single instruction.

Interrupt handling is much simpler on stack machines than on either RISC or CISC machines. On CISC machines, complex instructions that take many cycles may be so long that they need to be interruptible. This can force a great amount of processing overhead and control logic to save and restore the state of the machine within the middle of an instruction. RISC machines are not too much better off, since they have a pipeline delays the time between the occurrence of the interrupt and the time at which the interrupt instruction actually executes. Many RISC designs also have a large number of registers that need to be saved and restored in order to give the interrupt service routine resources with which to work. It is common to spend several microseconds responding to an interrupt on a RISC or CISC machine.

Stack machines, on the other hand, can typically handle interrupts within a few clock cycles. Interrupts are treated as hardware invoked subroutine calls. There is no pipeline to flush or save, so the only thing a stack processor needs to do to process an interrupt is to insert the interrupt response address as a subroutine call into the instruction stream, and push the interrupt mask register onto the stack while masking interrupts (to prevent an infinite recursion of interrupt service calls). Once the interrupt service routine is entered, no registers need be saved, since the new routine can simply push its data onto the top of the stack. As an example of how fast interrupt servicing can be on a stack processor, the RTX 2000 spends only 4 clock cycles between the time an interrupt request is asserted and the time the first instruction of the interrupt service routine is actually executed.

Context switching is perceived as being slower for a stack machine than other machines. However, experimental results presented by Koopman (1989) show that this is not the case. In fact, context switching costs for stack machines are approximately the same as CISC machines, and less than for most RISC machines. Furthermore, simple software techniques such as cooperative multitasking (Koopman 1990) can reduce the context switching costs to essentially zero.

A final advantage of stack machines is that their simplicity leaves room for algorithm-specific hardware on customized microcontroller implementations. For example, the Harris RTX has an on-chip hardware multiplier. Other examples of application-specific hardware for semicustom components might be an FFT address generator, A/D or D/A converters, or communication ports. Features such as these can significantly reduce the parts count in a finished system and dramatically decrease program execution time.

Program Execution Consistency

Advanced RISC and CISC machines rely on many special techniques that give them statistically higher performance over long time periods without guaranteeing high performance during short time periods. System design techniques that have these characteristics include: instruction prefetch queues, complex pipelines, scoreboarding, cache memories, branch target buffers, and branch prediction buffers. The problem is that these techniques cannot guarantee increased instantaneous performance at any particular time. An unfortunate sequence of external events or internal data values may cause bursts of cache misses, prefetch queue flushes, and other delays. While high average performance is acceptable for some programs, predictably high instantaneous performance is important for many real-time applications.

Stack machines use none of these statistical speedup techniques, yet achieve good system performance. As a result of the simplicities of stack machine program execution, stack machines have a very consistent performance at every time scale, which is especially important for control applications.

CONCLUSIONS

The new generation of stack computers embodies a distinct set of design tradeoffs that result in systems optimized for minimum complexity, fast program execution, ability to quickly service interrupts, small memory requirements, and consistent performance. These tradeoffs are different from those made by processors targeted at the workstation or personal computer markets, and make stack computers very well suited for embedded real-time control applications.

REFERENCES

Bell, J. (1973) Threaded code. *Comm. of the ACM*, June 1973, **16**(6) 370-372

Davidson, J. & Vaughan, R. (1987) The effect of instruction set complexity on program size and memory performance. In: *Proc. of the Second Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS II), Palo Alto CA, 5-8 October 1987*, pp. 60-64

Dewar, R. (1975) Indirect threaded code. *Comm. of the ACM*, **18**(6) 330-331, June

Ditzel, D., McLellan, H. & Berenbaum, A. (1987) Design tradeoffs to support the C programming language in the CRISP microprocessor. In: *Proc. of the Second Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS II), Palo Alto CA, 5-8 October 1987*, pp. 158-163

Earnest, E. (1980) Twenty years of Burroughs high-level language machines. In: *The Proc. of the Int. Workshop on High-Level Language Computer Arch., 26-28 May 1980, Fort Lauderdale FL*, pp. 64-71

Harris, N. (1980) A directly executable language suitable for a bit slice microprocessor implementation. In: *Proc. of the Int. Workshop on High-Level Language Computer Arch., Fort Lauderdale FL, 26-28 May 1980*, pp. 40-43

Harris Semiconductor (1988) *RTX-2000 Real Time Express Microcontroller Data Sheet*, Harris Corporation, Melbourne FL

Hayes, J., Fraeman, M., Williams, R. & Zaremba, T. (1987) An architecture for the direct execution of the Forth programming language. In: *Proc. of the Second Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS II), Palo Alto CA, 5-8 October 1987*, pp. 42-49

Kogge, P. (1982) An architectural trail to threaded-code systems. *Computer*, **15**(3):22-32, March

Koopman, P. (1987) The WISC concept. *Byte*, **12**(4) 187-194, April

Koopman, P. (1988) 32 Bit RTX Chip Prototype, *Journal of Forth Application and Research*, 5(2) 331-335

Koopman, P. (1989) *Stack Computers: the New Wave*, Ellis Horwood, Chichester, UK

Koopman, P. (1990) "Heavyweight Tasking", *Embedded Systems Programming*, **3**(4) 42-52, April

Miller, D. (1987) Stack machines and compiler design. *Byte*, **12**(4) 177-185, April

MISC (1988) *MISC M17 Technical Reference Manual*, MISC Inc., Aurora, Colorado

Moore, C. (1980) The evolution of Forth, an unusual language. *Byte*, **5**(8), August

Ohran, R. (1984) Lilith and Modula-2. *Byte*, **9**(8) 181-192, August

Rabbat, G., Furht, B., & Kibler, R. (1988) Three-dimensional computer performance. *Computer*, **21**(7) 59-60, July

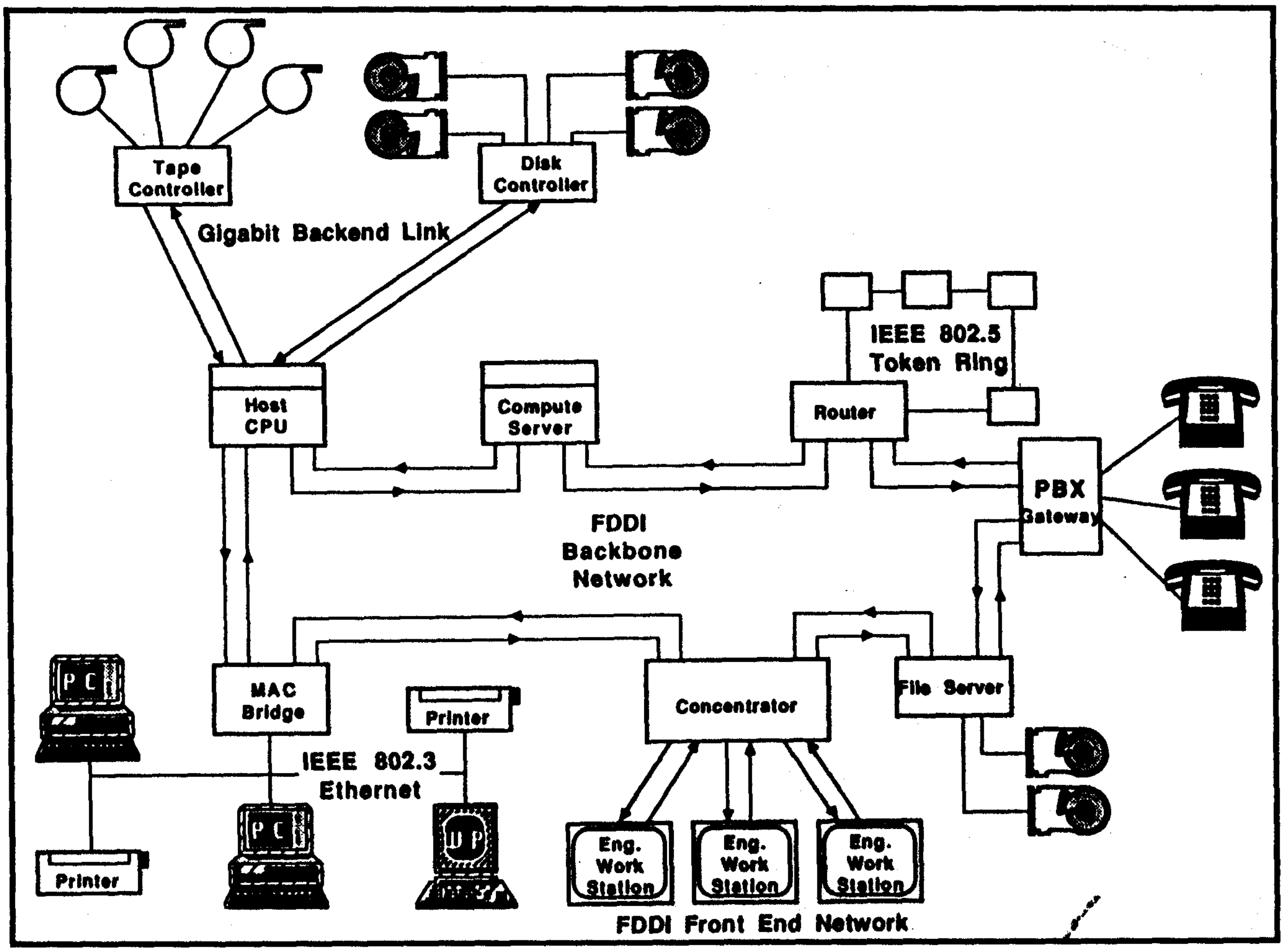
Schoellkopf, J. (1980) PASC-HLL: A high-level-language computer architecture for Pascal. In: *Proc. of the Int. Workshop on High-Level Language Computer Arch., Fort Lauderdale FL, 26-28 May 1980*, pp. 222-225

Systems Design & Networks Conference

The Conference on Computer Systems, Peripherals and Networks

Santa Clara Convention Center, Santa Clara, CA

Conference Proceedings Microprocessor Track



EDITOR
Kenneth Majithia

MAY 8-10, 1990
SANTA CLARA, CA

Sponsored By:
EDCON &
SFBAC OF IEEE

In Cooperation with:
Electronic Design
(a Penton Publication) and ACM

MAPLE PRESS