Harris Real Time Express™ (RTX™): A New Concept For Real-Time Control

Contact: Dave Williams
Market Development Manager
(407) 729-4629

HARRIS

TECHNICAL
BACKGROUND
INFORMATION
Revised May 1988

Harris Real Time Express (RTX): A New Concept For Real-Time Control

WHAT ARE THE REQUIREMENTS FOR "REAL TIME?"

"How fast is fast enough for real time?" is a complicated question because of the application specific nature of real time computing. For example, an acceptable real time response for a transaction processing system might be a half-second response to a query. For an avionic control system, however, a half-second response to an external event is likely to be too slow. For the former, conventional data processing type processors are clearly an effective solution to real time requirements, but in many applications -- data acquisition, process control, robotics, local area network controllers, digital signal processing -- the response within a critical time period is extremely important. As a result, real time can only be defined within the context of the end application. It is driven by the processing requirements to meet time critical external events.

Many computers need fast instruction execution speeds. However, for real time applications interrupt latency and context switch times are important specifications in addition to the raw instruction execution speed of a processor. Recent advances in hardware and software have reduced response times for interrupts and context switches down an order of magnitude for many processors, to the many tens of microseconds or many hundreds of microseconds, respectively. The RTX 2000 brings these response times down another order magnitude to 400 nanoseconds for interrupt response time,and two microseconds for context switch time, while achieving an instruction execution rate over 10 MIPS.

A key consideration for real-time processing is predictability. Most general purpose computers have features to improve average instruction execution times. Features such as pipelines, caches, on-chip registers, and optimizing compilers all contribute to an improvement in average execution rate. But, they also contribute to more uncertainty with respect to critical timing predictability. As a result, external logic, such as DMA controllers or I/O processors, must be provided to support all but the most routine interface requirements with the external world. The RTX 2000 significantly contributes to moving more hardware into software as a result of high instruction execution rate, rapid responses to external events and predictable timing of instruction execution.

HIGH SPEED MICROCONTROLLERS - A NEW ALTERNATIVE TO CONVENTIONAL MICROPROCESSORS FOR REAL TIME

Most conventional microprocessors are optimized for an office automation computer or computer-aided workstation computing environment. Significant complexity is added to these machines in order to support the memory management and general purpose data processing requirements of these applications. Microcontrollers, on the other hand, take advantage of the increased circuit density made possible by advances in silicon processing to enhance the device for controller or other specific applications. This has produced a large variety of devices that are useful for specific applications as well as microcontrollers for more general purpose use. However, microcontrollers have traditionally been much slower than microprocessors and have not been optimized to support real time applications.

Microcontrollers can be broadly classified as either application generic or application specific. The generic products, such as the RTX 2000, provide solutions for a broad variety of applications. The RTX 2000, however, with its unique ASIC Bus, provides the capability of optimizing the solution by simplifying the partitioning between hardware and software through the use of external ASIC peripherals. The obvious extension to this philosophy is to incorporate those peripherals on the ASIC Bus within the IC. Therefore, application specific devices (both standard product or customer specific) can be developed as a natural extension to the core processor.

The emergence of high speed 16-bit microcontrollers provides a new opportunity to replace hardware with software. A major limitation to the ability to replace hardware with software has been the performance constraints of software solutions. The RTX 2000's substantial increase in performance provides new opportunities to replace random logic with software providing significant improvements in flexibility and time to market. For example, because of the very high speed of the RTX 2000, a full duplex UART can be emulated in software and requires less than 1% of the processor's bandwidth to perform the functionality of a 1500 gate UART.

RTX 2000 - PERFORMANCE THROUGH SIMPLICITY/ PARALLELISM

The RTX 2000 achieves performance through simplicity. The chip is designed for simplicity. It has no pipeline; no microcode sequencer; and no microcode. All instructions except memory access instructions execute in one cycle (memory access instructions execute in two). The RTX 2000 minimizes address calculation delays by incorporating a simplified memory paging mechanism, and eliminates the complexity of multiple addressing modes and memory management. The RTX 2000 is a stack machine. Stacks facilitate the evaluation of expressions and minimize the control overhead needed to organize data. The stack uses only a single pointer register to keep track of and access its data. A stack machine not only uses a stack for temporary data storage, but executes all operations on data from the stack. The ALU thus finds all of its data in a pre-defined location, and can get that data without an address specification. In addition, no addresses need to be compiled for stack access. The RTX 2000 also has a hardware return stack which handles subroutine return addresses. This stack can also be used for temporary data storage as well.

The RTX 2000 instruction set is sub-divided into six instruction classes, with each section controlling a hardware operation. Like horizontally micro coded bit slice architecture instructions, multiple operations can be compacted and coded within a single OP code to execute in parallel. Four separate buses for the data stack, return stack, memory and ASIC Bus and operate in parallel, significantly increasing instruction execution efficiency. For example, the OP code BE68 (8 G@ ;) simultaneously references all four address spaces. It fetches a 16-bit data value from the ASIC Bus, pushes it onto the data stack, forces a subroutine return, which pops a 20-bit return address from the return stack and fetches the next instruction all within a single clock cycle.

In a conventional processor, high-level structured programs are converted from groups of procedures with stack-oriented local variables to machine code. A considerable change in the look and feel of the program takes place as high-level language operations are transformed into groups of primitive operations. While the complex machine instructions that may support such stack operations (such as frame pushes and pops) and even fetch a variable (given a frame pointer and an offset) the paradigm switches from variables and frames (in a high-level language) to registers and memory pointers in machine code. The means of passing information between many high-level language procedures is the stack. The way of passing information between conventional machine language instructions is through registers or discrete memory locations. The fundamental mechanisms are completely different. Furthermore, conventional machines do not support efficient subroutine calls. Many clock cycles are required for managing the internal operations.

The RTX 2000 uses simple, and fast hardware to execute high semantic content instructions that closely reflect the structure of the program. Performance is not penalized for organizing programs into small, compact, understandable procedures. This results in compact program structures that are composed of hierarchically arranged solutions to sub-problems. Thus programs can be simultaneously optimized for small memory space, fast execution speed, and low development costs. This allows the hardware/software environment to deliver cost effective solutions to the users problems.

SUBSTITUTING PARALLELISM FOR PIPELINING IMPROVES SPEED WHILE IMPROVING REAL TIME RESPONSIVENESS

Pipelining is a common architectural strategy to increase the speed for conventional processors. For example, portions of a processor concentrate on fetching instructions, fetching operands, computing values, computing next addresses, and storing results. This method is a very efficient mechanism to increase speed for sequentially executing programs at a relatively small cost of added hardware complexity. However, pipelining impacts the timing of instruction response to subroutine calls, interrupts, and context switching, and the speed increases achieved by pipelining can be lost for highly structured programs.

The RTX 2000 abandons the use of pipelining as a means to increase circuit speed, and substitutes parallelism. The parallelism comes from exploiting two fundamental characteristics of the RTX architecture. The dual stack Quad Bus architecture provides an efficient mechanism to increase simultaneous operations. Harvard architectures have become increasingly popular in many applications because they effectively double the bandwidth of a microcomputer bus system. However, this is at the expense of increased interface requirements to separately address and interface both the data and program memory spaces.

In the case of the RTX 2000, parallelism is achieved by having on-chip stack memory for both parameters and return addresses, as well as an interface to main memory and the ASIC Bus for hardware acceleration. Since the stack buses are on-chip, I/O restrictions are eliminated. Also, since stacks are implicitly addressable without requiring address fields in the instruction, the number of functions that can be included in each word is increased. This leads to a significant improvement in performance due to the increased amount of work that is done within every instruction execution.

Another important difference between a conventional RISC machine and a CISC machine is the large semantic gap between high-level language source code and its corresponding machine code. This results in creating a large number of machine instructions for every high-level language instruction. The RTX 2000 differs dramatically from that mode of operation by having a single instruction correspondence to most FORTH instructions and, in fact, can pack up to three FORTH instructions in a single word. This high semantic content in each instruction greatly improves the effective operating speed of the processor.

Another limitation of pipelining in a heavily structured or asynchronous real time environment is the inefficiency and uncertainty of emptying and refilling the pipeline when branches or interrupts occur. The RTX 2000, by eliminating the pipeline, significantly improves the utilization of all available memory cycles and reduces the timing uncertainty of instruction execution.

STACK BASED PROCESSORS - OPTIMUM FOR REAL TIME STRUCTURED PROGRAMS

Conventional computers are optimized for executing programs made up of streams of serial instructions. Conversely, modern programming practices stress the

importance of non-sequential control flow, and small procedures. The result of this hardware/software mismatch in today's general purpose computers is a costly sub-optimal compromise. In fact, the very philosophy of conventional RISC architectures is to implement only the most used instructions by studying software programs which have been based on existing architectures. This is a self-perpetuation of the programming style dictated by register based Von Neumann machines. The RTX 2000 takes a radically different approach by optimizing the instructions set to the requirement of a particular high level language (FORTH) which is well suited to real time control. However the architecture is also well suited to other high level languages. The objective is to promote the use of a highly structured programming style for real time, without the usual performance penalties. The RTX 2000 provides an efficient procedure for subroutine call through the use of the stack to store the parameters and a highly efficient subroutine call itself. A subroutine call takes only one clock cycle and the return can take zero clock cycles (since it can be implemented within the last instruction of the subroutine sequence).

RTX - PROVIDING FLEXIBLE PARTITIONING BETWEEN HARDWARE AND SOFTWARE

While semicustom ICs provide an attractive performance and integration alternative to standard microprocessors, they are not amenable to dealing with change. Off the shelf programmable ICs, with their support hardware and software tools, offer far greater flexibility. Consequently, when designers face requirements for both high performance and adaptability, some are finding that a mixture of application specific ICs and standard products is the optimum approach. The ability to partition the task between hardware and software is an important element in achieving total system performance. The RTX 2000 significantly increases the speed of solving hardware problems in software. In some applications, however, only hardware is fast enough. The ability to make an efficient partitioning between what is implemented in hardware and what is implemented in software is a key element in achieving system cost and performance objectives. The RTX ASIC Bus is a unique approach to assist developers in partitioning hardware and software efficiently. Because the memory bus and ASIC Bus run concurrently and the ASIC Bus can be operated on directly by RTX instructions, hardware accelerators can easily be added internally or externally to speed up processing functions.

RTX vs. CONVENTIONAL RISC PROCESSORS

Traditionally, most embedded control processing functions have been performed by general purpose microprocessors such as the 8086, 80286 or 68000 family. With the emergence of RISC computers, many manufacturers are claiming that they are well suited for embedded control applications. RISC processors are generally optimized for the requirements of 32-bit workstations and super microcomputers. While they are clearly capable of providing computing power for embedded applications, they have many specialized features that relate to supporting the UNIX environment and the specific requirements of computer aided engineering. While these processors have a small simple instruction set to achieve speed, they are hardly simple devices, most having in excess of 200,000 transistors, extensive memory management, and extensive pipelining. In addition, they require complex compilers to create efficient optimized code. As a result, the programmer loses visibility into the actual operations of the machine, thus creating a difficult to design environment for time critical functions. Since the machine is dependent on the operation of this optimizing compiler, programming in assembly language for those time critical applications is prohibitive.

While on-board memory management is useful for computer aided engineering functions in a multi-user environment, most realtime applications run logical to physical, not using virtual addressing at all. There's usually no need for virtual addressing in realtime applications. While the protection features of a memory management unit can be handy during program development, they are usually not required once the finished code is running.

Most RISC machines make extensive use of registers. As an example, the AMD 29000 has 192 general purpose registers on board. While each task may not use all registers, swapping out an extensive set of registers during a context switch creates an excessive latency problem which is often unacceptable. Allocation of a fixed number of registers to each task becomes a confining condition for the compiler.

Several characteristics of FORTH facilitate a simple scheme for context switching. Conventional architectures are not fast at context switch because they use a large number of registers. Saving or restoring a FORTH task in an RTX context switch takes little time because FORTH uses as few as three registers. The RTX 2000 core contains only eight registers so that a complete context switch to store all of the registers can be done very quickly.

RTX - EFFICIENT OPERATION FOR MULTI TASKING REENTRANT PROGRAMS

Although all real time operating systems are multi-tasking, not all multi-tasking operating systems are real time. Unix, for example, takes far too long to answer interrupts and make a context switch to suit real time applications. Its file structure suits program development, but not online control. Unix does not use reentrant code. If 16 users invoke an editor, then Unix loads 16 copies of the editor, thereby consuming large amounts of memory. Further, it has only rudimentary facilities for inter-task communication synchronization.

Another advantage of a stack oriented machine is the capability to efficiently support reentrant code. Reentrant code is useful in real time systems for two reasons: first, it saves space, because many tasks can use the same reentrant code simultaneously. The fastest real time systems must keep all code in memory -- a practice that puts a premium on compact coding style. Second, reentrant code exactly suits multi-tasking since you can interrupt the process using reentrant code at any point in the code segment, and later resume the process with no adverse effects. FORTH produces code that is inherently suitable for reentrant programs. Other languages require discipline on the part of the programmer. Making a routine reentrant simply means that all variables must reside in an area private to the task using the code, not in the code itself. The penalty for using reentrant code can be increased overhead or more CPU cycles for conventional processors, because read and write operations are indirect rather than immediate. Because FORTH promotes an object oriented programming style, reentrant programs are more manageable and comprehensible, especially in a multi-tasking environment. Of all the languages commonly used for real time control, only FORTH offers a straight forward programming facility for building classes of objects.

FORTH - WELL SUITED FOR REAL TIME OPERATING SYSTEMS

FORTH was originally developed for real time applications, and from its inception it has included features designed to simplify handling multiple concurrent tasks. Harris plans to offer several real-time operating system solutions for the broad variety of application requirements that the RTX 2000 will serve. FORTH, Inc.'s polyFORTH, for example, which will be offered for the RTX 2000, includes a multitasking, multiuser real time OS which has been widely used in industrial and aerospace applications. polyFORTH uses a proprietary multitasking algorithm which has been benchmarked at between four and twenty times faster than other real time OSs on CPUs of the 68000 and 8086 families (I&CS, October, 1987).

The secret to polyFORTH's speed is a "non-preemptive" multitasking algorithm. This means that a task relinquishes control of the CPU under well-defined, predictable circumstances, instead of having control taken away unpredictably when a time interval is up or an external event requires handling by a higher priority task.

In systems using preemptive multitasking, a task may be suspended when it is partially through updating a variable, for example. To avoid this, mailboxes or shared variables may only be accessed through system calls, which resolve potential conflicts at some cost in overhead. In polyFORTH, however, such a situation cannot arise, and so a shared memory region may be used to contain data of interest to two or more tasks with no OS overhead involved.

The polyFORTH multitasker is a simple round robin, and normally tasks don't have priorities. The non-preemptive round robin algorithm ensures optimum performance to all tasks. A task relinquishes the CPU whenever it performs any kind of I/O operation (including "virtual I/O" such as writing into screen memory). When the operation is complete, the task will wake up on its next turn. In most real-time applications there is so much I/O being performed that this is sufficient to guarantee rapid turnaround. There are commands available to "tune" CPU-intensive operations, if necessary. A real-time clock helps you to monitor how long tasks have to wait to wake up, and do whatever tuning is appropriate.

Another way in which polyFORTH ensures that all functions are performed as fast as necessary is by having no OS overhead whatever on interrupt servicing. Event response in polyFORTH is a two-layer process: interrupts are serviced instantly, with the hardware vector going straight into the application service routine. This service routine handles the most time-critical operations (reading a value and storing it in memory, for example), then notifies the task responsible for the interrupting device that the event has occurred. The task will "wake up" on its next pass through the round robin and handle the more complex aspects of processing.

The combination of low-overhead task management with instantaneous interrupt servicing provides the ability to handle extremely high data rates and complex real time applications with ease.

RTX COMPILERS FOR OTHER LANGUAGES

While the RTX 2000 directly executes FORTH, efficient compilers for other languages can also be developed. The architecture for the RTX 2000 is well suited for providing facilities for the efficient implementation of other languages such as "C", PROLOG and ADA. "C" compilers also use stacks to create local variables and to pass run time parameters among functions. "C" breaks tasks up into functions. A "C" program consists essentially of a series of functions with one beginning function specified as main ( ). It is straight forward to implement a "C" language run time allocation stack using the RTX 2000's fast access user memory locations as pseudo-registers. Data can be accessed by allocating one of the RTX 2000 pseudo-registers (first 32 words of memory) to create a pointer into the stack. This stack can be used to file clusters of information called frames. Offsets into the stack are addressed to fetch as 2-part addresses. A routine to find data first asks which frame and then which element in the frame to fetch. A stack frame is a miniature segmented memory with a 2-part address.

Stack frames store information on entry to functions. They permit temporary storage of variables and parameters so that subsequent routines can run without interfering with

other functions, variables and parameters. The RTX instruction set pseudo-register operations support the capability to develop an efficient "C" compiler. Since instructions execute at a high clock speed, the use of pseudo register enables micro code-like performance of custom "C" run time stack instruction sequences.

RTX 2000 - WELL SUITED FOR REAL TIME ARTIFICIAL INTELLIGENCE

The recent flurry of activity in commercial expert system development has all but bypassed the real time computing community. Although it is desirable to incorporate expert systems in some real time computing applications, the amount of computing over symbols (reasoning) required by an expert system is difficult to implement in real time. General real time symbolic processing has remained an AI research problem, however a number of applications have been implemented which use FORTH. FORTH provides an integrated environment for both conventional data processing and AI with a FORTH Prolog compiler. This technique has been applied in a diesel electric locomotive repair expert system, an orbiting spacecraft command and control system, a spacecraft trajectory processing data error detection system, and a real time polysomographer sleep disorder diagnosis system. Other applications include utilization of expert system capabilities for such applications as radar and sonar processing, image compression and analysis, etc. A Prolog compiler is being developed which provides a set of high-level artificial intelligence programming tools (i.e., inference engine, language parser, etc.) built from FORTH primitives to take advantage of the high run time execution speed offered by FORTH. In this implementation, the Prolog interpreter is imbedded in a FORTH environment. Real time algorithms stored in the knowledge base through this mechanism can subsequently be executed on a logic driven basis by the expert system.

RTX ALGORITHMIC COPROCESSORS IMPROVE TOTAL SYSTEM PERFORMANCE

Designers of both microprocessors and peripheral interfaces are struggling to minimize the traffic jam their own success has created. New techniques in bus management, heavy use of specialized memories like caches and FIFO buffers, and a new reliance on distributed I/O processing are all strategies being used to resolve microcomputer I/O bottlenecks. As these techniques are more widely used they are changing the architecture of silicon components, and they are equally affecting the practice of microcomputer system design. The RTX 2000 provides a significant capability to attack the I/O processor and coprocessor requirements to improve overall system performance by distributing the processing. Of significance is the development of a shared memory to provide an efficient interface between an RTX coprocessor and host. The ability to have a stored program controller as a coprocessor provides the capability to compute complete algorithms rather than just a single operation at a time. This provides a major improvement in the system performance due to elimination of heavy bus overhead and the inefficiency of tying up the host processor to service the coprocessor. Such applications as Local Area Network controllers and virtual disks, demand that memory pages be swapped over the network or disks for remote file service, but the increased frequency with which packets arrive and depart can cause a workstation to approach its memory bandwidth limitations. Another example is when a microcomputer forms the backend of the signal processing system. Data may be required for the digital signal processing algorithms at very high rates. Maintaining data flow through the system may demand that the system microprocessor continue running at these high speeds until all data is processed.

The push to make individual components run faster soon runs into complications. The utilization of coprocessing controllers within the microcomputer environment is a technique which can significantly increase performance of the overall system by reducing the amount of bus interface time for the host processor. A powerful strategy for breaking up bottlenecks is to move the I/O driver code from the CPU to the peripheral controllers. An obvious example is in serial concentrators where the processor monitors the number of serial lines, accumulating data until the entire block has been received. In a multi-user UNIX system for example, this function usually requires a serial concentrator to perform basic UNIX line editing functions since these operations must occur before the end of line character comes in from the terminal.

The strategy of removing the responsibility for device and housekeeping off the CPU and on to the peripheral controller dictates that the device controllers become programmable to pick up many of the tasks formerly executed by the device driver software in the host. Moving these tasks to the controller not only removes slow dumb tasks from the central resource, but also spares the central processor the hail of interrupts, context switches, commands and status bytes that are a necessary side effect of centralized device drivers. This not only maximizes I/O speeds, but it also simplifies the user's programming requirements. The I/O controller concept provides a system design that emphasizes distributed processing with a high degree of concurrency and parallelism. In addition, this architecture provides an environment that reduces the data movement within the system.

Currently, I/O processors of this class are based on bit slice architectures in order to provide the performance necessary to manipulate data on the fly. A standard product microprocessor such as the RTX 2000, with a high level programming language, helps users interface standard I/O ports to

the host processor. The result is an architecture optimized for high performance, but with a high degree of modularity and user programmability, saving developers both time and expense. The utilization of I/O processors also helps avoid arbitration between devices producing input and output data streams. Since the RTX 2000 has all of the provisions to support direct memory access management, the processor has the capability to incorporate smart DMA, which relieves the host of this burden.

REAL TIME SOFTWARE DEVELOPMENT IS MORE COMPLEX THAN CONVENTIONAL PROCESSING

The characteristics of real time software systems that set it apart from conventional data processing applications are that real time systems must:

- Respond to real world stimuli.
- Within a finite period of time.
- By directly manipulating hardware resources.
- As a set of concurrent asynchronous processes.
- With a high degree of reliability.

The RTX family contains both software development tools as well as fast hardware to operate in this environment. It promotes an interactive programming environme@t which has four primary attributes:

- A set of highly integrated tools.
- Which are low cost and simple to use.
- Which do not burden the target system with unnecessary complexity.
- Which promote use of the underlying structure of the program as an organizing tool.

STRUCTURED PROGRAMMING SUPPORT REDUCES DEVELOPMENT/DEBUG COSTS FOR REAL TIME

Hardware that is fundamentally based on the concept of modularity and programmer interactiveness will lead to changes in programming style that will better support efficient software development. The expense of using a software programmable microcontroller to solve a problem consists of not only the money for hardware, but also the development costs of creating and debugging the code. Previously the cost of solving problems with computers was dominated by hardware costs, but as hardware costs have plunged, software costs have grown by leaps and bounds. This is nowhere more evident than real time control. Real time control applications tend to be significantly more complex than conventional programming and, ironically, offer the least amount of high-level language support for those time critical functions. The developer is caught in the dilemma of trying to write most of the program in a high-level language support such as C or ADA, only to have to merge in assembly language programming for the most time-critical applications. Worse, during the debug process there is a poor correlation between what is written in the high-level language and what appears in the machine language, because the compiler has dramatically altered the programming style of the program. The compiler modifies content by providing optimization including the unrolling of loops into in-line code, and expanding the lowest level procedures as macros within the calling routines.

RTX - MAINTAINING KNOWLEDGE OF HARDWARE OPERATION WHILE EXECUTING A HIGH LEVEL LANGUAGE

Since the RTX creates a virtual FORTH machine, the correlation between the high-level FORTH language and the operations executing in the FORTH engine are closely coupled. Therefore, the uncertainty with regard to what the compiler is doing to the source language of the real time system is significantly reduced. Also, by providing symbolic capabilities within the development system and operating completely within the high-level language, software and hardware debug and integration is significantly simplified.

Optimizing compilers obscure the correspondence between source code and compiled object code. Programs written in a high-level language that need to meet specific response time specifications require the programmer to switch on the compilers optimizer and then debug optimized code. Among the many techniques optimizing compilers employ is register allocation by coloring. Coloring keeps the most commonly used values and registers at all times. The compiler examines the entire subroutine to determine which local variables and parameters are used most often in a routine. It allocates them to registers. Further, the register allocator can use data flow analysis to find the lifetime of each variable. Using this information it can increase the number of variables that get stored in registers by using the same register for several variables in the same subroutine.

A software developer must be very familiar with the operation of the compiler in order to understand the implications of what is happening in real time software. The designer cannot be assured of just where program variables are as when programming in assembler language and making the variable assignments explicit. A key feature of writing in FORTH and executing in FORTH on the RTX 2000 is the close correlation between the high-level language and the actual execution of the machine. Real time programs which must respond in a critical time period can therefore be more easily designed since the complete operation of the machine is understood.

RTX IMPROVES SOFTWARE PRODUCTIVITY FOR REAL TIME

According to the Defense Science Board Task Force on military software, most software productivity gain has been brought about by three factors. First is utilization of a high level language. The removal of awkwardness of machine

language has been shown to increase software productivity by a factor of 10. While the benefits of a high level language have been available for low performance functions, many real time applications heretofore have often been programmed in assembly language. Even with high level languages, slow turnaround, edit, compile, link, load, and debug cycles contribute to a loss of mental continuity in the development and debugging of software. An integrated, interactive development environment has been shown to result in improvements in productivity from two to five. Finally, compatibility of files, formats and interfaces among the various tools has been demonstrated to increase productivity by a factor of two. These are precisely the benefits of FORTH that have been incorporated into the RTX family development tools.

REAL TIME DEBUG DEMANDS SPECIAL CONSIDERATIONS TO OPERATE AT FULL SPEED

In a typical software development environment (the host and target method), programmers use a general purpose host such as a DEC VAX, a CAE workstation, or an IBM PC AT to generate application code. After they write the code, programmers must transport it to the target for final testing. Unfortunately, for most embedded microprocessor development the target system does not offer enough programming and test resources to support the total development process. In order to debug the hardware, an In Circuit Emulator (ICE) is the most traditional vehicle for testing programs written for embedded processors. While an ICE is a powerful tool for software and hardware integration, its high cost may be difficult to justify for an application specific device. Of more importance, the cabling requirements and timing constraints for a 10-MIP processor provide a significant performance limitation in the@feasibility of debugging the system at full speed for realtime applications. The Harris approach provides a real time debugging monitor in the target system and is an excellent alternative to ICE. Because the RTX is fully static and eliminates pipelining, the problem of getting the hardware to work initially can be solved through single step operations using low cost logic analyzers and the host development system. Then the full capabilities of the integrated hardware/software development system can be used to provide a cost effective and high performance approach to achieving hardware/software integration.

RTX - OPTIMIZED FOR RAPID TIME TO MARKET

Time to market is becoming an increasingly important criteria for the makers of electronic systems. The problem has been intensifying as product lifetimes have progressively shortened. Makers and users of semicustom chips feel the pressure even more because of the need to generate prototypes before a system can be debugged and demonstrated, much less marketed. The RTX 2000 provides a cost effective means of implementing a core based processor in a semicustom environment. Because it is a standard product with an ASIC Bus designed to accommodate application specific peripherals, prototypes can be rapidly developed with a standard product. This significantly lowers the risk of development for a complex microcontroller based on a core processor. Because the ASIC Bus can be integrated on-chip, future versions can incorporate not only the core processor, but the external peripheral logic and memories all on one chip.

In almost every project -- military or commercial -- the faster the prototype is up and running the more likely the system will meet the market window. Shorter prototyping time means getting to market faster, with less risk. Realtime systems tend to have complex relationships with external hardware. A hardware prototyping vehicle is a useful mechanism to validate real time performance and is complimentary to the use of simulation.