Star Award

RTOS Acceleration Using Instruction Set Customization

Institution: **Centre for High Performance Embedded System (CHiPES), Nanyang Technological University (NTU)**

Participants: **Muhamed Fauzi Bin Abbas, Ku Wei Chiet**

Instructor: **Professor Thambipillai Srikanthan**

Design Introduction

As embedded system designs become increasingly more complex, the use of real-time operating systems (RTOS) becomes essential to meet time-to-market pressures and to contain non-recurring engineering costs. However, an RTOS consumes precious CPU cycles in return for the services it provides. Further, the RTOS is typically treated as a pure software entity and is subject only to minor adaptations. In this project, we leverage work done by our research group towards instruction set customization for RTOS acceleration using dedicated hardware and apply it to the Nios® II processor. We present our design, findings, and results in this paper.

In this project, we implemented RTOS acceleration by customizing the Nios II instruction set. As part of the process, we identified RTOS operations that are executed frequently and converted them into custom instructions (CIs), thereby collapsing a series of instructions into fewer operations and reducing the RTOS CPU overhead.

By reducing the time consumed by system tasks, greater processing time is made available for user applications. The system becomes more responsive, and this effect is further noticeable if the CPU is clocked at a lower clock frequency, where the same number of clock cycles constitutes a higher percentage of system overhead. By allowing more time for user tasks, RTOS acceleration benefits applications that are run on the system without changing the applications. This acceleration provides a drop-in method for improving system performance and responsiveness.

The effect of our work is very noticeable in systems that run a large number of tasks. For our project we use the Nios II/s CPU and μC/OS-II.

We decided to use the Nios II processor because of the following factors:

- The Nios II processor is a soft-core CPU that allows instruction set customization. In our opinion, using instruction set customization for RTOS acceleration is extremely attractive because the instruction execution is serialized with respect to other instructions, retaining the sequential nature of the original function being accelerated. However, all parts of the custom instruction can execute in parallel when the instruction is executed. Finally, this instruction can be accessed either in assembly language or as an intrinsic function from a high-level language such as C or C++, making integration with the software RTOS simpler.

- SOPC Builder makes it extremely easy to add custom instructions to the CPU core. In the future, we plan to explore the use of the C-to-Hardware Acceleration (C2H) Compiler for this work. This implementation would potentially make it easy to create new RTOS customization instructions because the RTOS itself is typically written in C.

At the same time, it seems that the Nios II processor is typically targeted at medium complexity applications (about a 100-MHz clock with multiple tasks). We found that in this configuration, RTOS overheads can be very significant in systems with a large number of tasks. These types of systems stand to benefit significantly from our work. However, as we shall demonstrate later in the paper, our proposed design can be easily parameterized to cater to different workloads and requirements.

Function Description

For this project, we used µC/OS-II running on the Nios II/s processor. By analyzing the RTOS, we determined that the two most frequently executed portions are:

- *Timer tick routine*—The timer tick routine executes at the system clock frequency, which typically ranges from 100 Hz (10 ms) to 1,000 Hz (1 ms). Because all RTOS operations are expressed relative to this parameter, the system clock frequency has a direct impact on the system resolution. This routine runs as part of the interrupt service routine (ISR) in response to the timer tick, therefore, it is executed very frequently, and acceleration results in significant savings.

- *Scheduler*—The scheduler is called every time the status of one or more tasks changes in the system. Further, some portion of the scheduler is called with interrupts disabled. Any improvement in these areas results in reduced interrupt latency. In general, the scheduler is called as the final operation of many system calls and acceleration improves the system call processing time.

µC/OS-II handles 64 tasks with unique task priorities. The task priority is the same as the task ID.

Time Management Module

The timer services offered by µC/OS-II consist of timeouts and delays. The timer services' basic unit of measurement is the system timer tick, which is generated as a periodic interrupt by a hardware timer in the system. Timer management affects the following areas:

- All delays and timeout requests are expressed in number of ticks. µC/OS-II, which has a 16-bit internal timer delay variable, supports delays of up to 65,535 ticks.

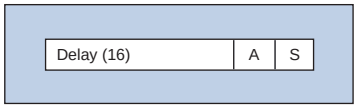
- When a timer tick occurs, the tick ISR decrements the waiting task's delay variable. If the variable reaches 0, the task can be added to the ready list.

- If the task is suspended when its delay value reaches 0, the task delay value is reloaded with 1 to delay it by another tick. This process continues until the task is resumed.

The timer module's basic architecture comprises a 16-bit counter to store the delay value for each supported task. Additionally, two extra bits are stored for every task: one bit stores whether the delay

counter is active and the other stores whether the task is suspended. Figure 1 shows the basic unit for each task, which is replicated for the number of tasks expected in the system.

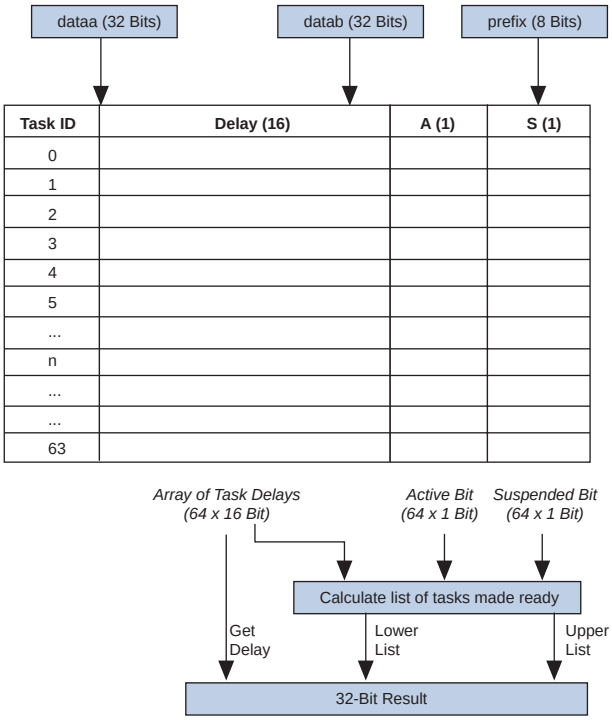
Figure 1. Timer Management Basic Unit Architecture



Name	Width	Function
Delay	16	Delay counter.
A	1	Active. Indicates whether the counter is in use.
S	1	Suspended. Indicated whehter the task is suspended.

Figure 2 shows the architecture for a full implementation with 64 units. The instruction usage is described next.

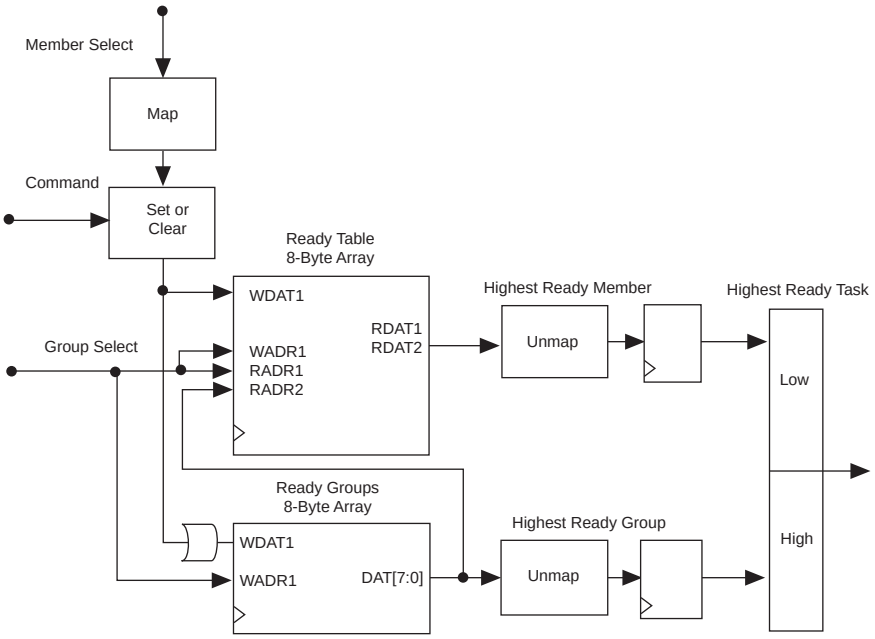
Figure 2. Custom Instruction for 64 Tasks



Scheduler Module

In an associated project in our research group, the basic scheduler was accelerated using instruction set customization [1]. Figure 3 shows the architecture diagram for that implementation. We implemented the scheduler so that it could be instrumented and tested in the same manner as the other work done in this project.

Figure 3. Scheduler Custom Instruction Architecture Block Diagram



Performance Parameters

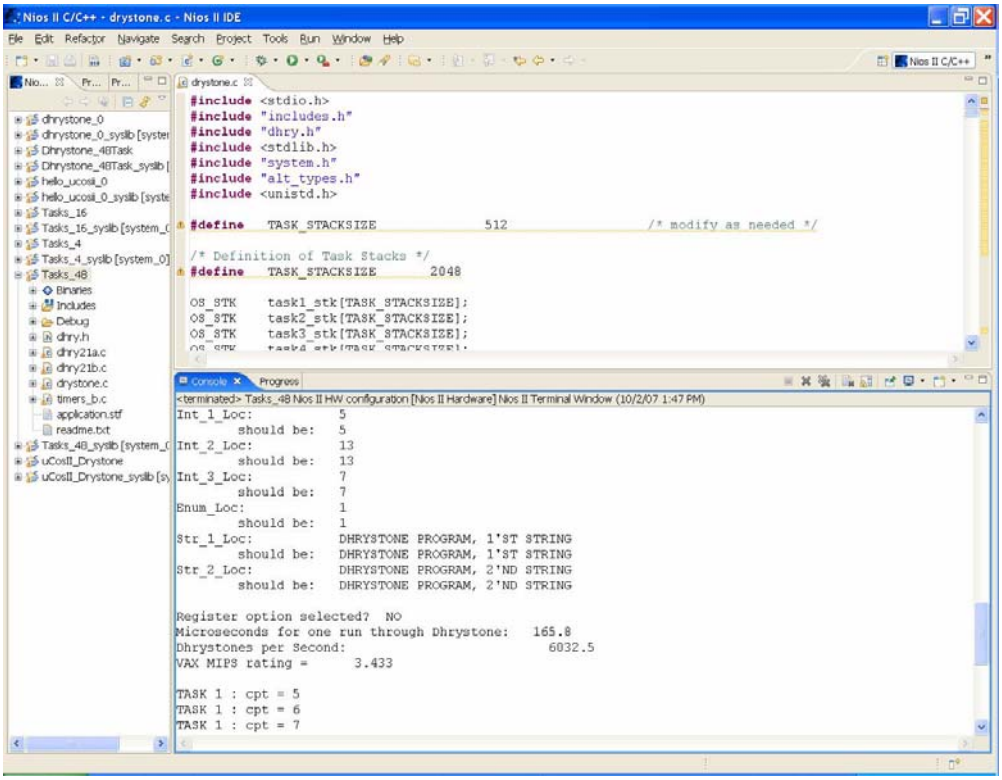
The design uses 3,748 registers and 41% of the logic elements (LEs) available in the Cyclone® II device on the Development and Education (DE1) board, as shown in Table 1. Details about the hardware consumption are discussed later in this section.

Table 1. Design Resource Usage

Resource	Description
Flow status	Successful - Tue Oct 02 09:52:15 2007
Quartus II version	7.1 Build 156 04/30/2007 SJ Full Version
Revision name	DE1_SD_Card_Audio
Top-level entity name	DE1_SD_Card_Audio
Family	Cyclone II
Device	EP2C20F484C7
Timing models	Final
Met timing requirements	Yes
Total logic elements	7,749 / 18,752 (41%)
Total combinational functions	7,278 / 18,752 (39%)
Dedicated logic registers	3,631 / 18,752 (19%)
Total registers	3748
Total pins	287 / 315 (91%)
Total virtual pins	0
Total memory bits	47,104 / 239,616 (20%)
Embedded multiplier 9-bit elements	4 / 52 (8%)
Total phase-locked loops (PLLs)	1 / 4 (25%)

Figure 4 shows the debug messages sent from the DE1 board to the PC console through the JTAG connection. The messages show how we benchmarked the system using the Dhrystone benchmark program.

Figure 4. Debug Messages



The custom instructions need to be carefully added into specific parts of the RTOS and are called from multiple places in the system. Therefore, there is an overall system performance impact from our project. In this section, we report the main results and improvements by using custom instructions for RTOS acceleration.

Performance Improvement of Specific Functions

Although the custom instructions are used in many places in µC/OS-II, their main impact is seen in a few specific functions.

Timer Tick Custom Instruction

For the timer tick custom instruction, the main impact is in the timer tick ISR (OSTimeTick). At every system timer interrupt (typically every 1 or 10 ms), the timer tick ISR is run in response to the timer tick interrupt.

The timer tick ISR has 2 main operations:

- The wait count of each task is decremented by 1 by iterating through the task control blocks for all tasks present in the system.
- For unsuspended tasks whose wait count becomes 0, the scheduler is called to update the task's ready list.

The timer tick custom instruction only affects the decrement operation. Because the kernel iterates through the waiting tasks, the function's execution time directly depends on the number of tasks in the system. Table 2 shows the results for the OSTimeTick module. The performance improvement

(calculated as a percentage) is shown in brackets. We tabulated the results with 4 tasks (light load), 16 tasks (medium load), and 48 tasks (high load) in the system.

Table 2. Timer Tick Performance

CPU Types (# of Tasks)	Software *(1)*		Hardware (with Custom Instructions) *(2)*	
	st *(3)*	lt *(4)*	st	lt
4 Tasks	584	612	107 (81.60%)	183 (70.10%)
16 Tasks	921	2525	249 (72.90%)	298 (88.20%)
48 Tasks	4,057	4,844	219 (94.60%)	260 (94.60%)

Notes:

(1) The Software column indicates performance when only the pure software RTOS is executed (i.e., without any custom instruction support).
(2) The Hardware (with Custom Instructions) column indicates performance when the RTOS has been accelerated using the custom instruction.
(3) st refers to the typical shortest time spent in the execution of the relevant portion.
(4) lt refers to the typical longest time spent in execution.

As shown in Table 2, the performance improvement is in the range of 70% to 81%, even for systems with a light load. As the number of system tasks increases, the software RTOS starts to show very high overhead. In contrast, the RTOS supported by the custom instruction scales better and less time is consumed in the ISR, showing more than 90% improvement.

Scheduler Instruction

The scheduler instruction significantly affects many areas of the RTOS. The main impact is noticed in OS_Sched. As with the timer tick, the performance depends on the number of system tasks. Table 3 shows the performance data (see Table 2 for the column definitions). The typical improvement is 40%.

Table 3. Scheduler Instruction Performance

CPU Types (# of Tasks)	Software		Hardware (with Custom Instructions)	
	st	lt	st	lt
4 Tasks	295	423	140 (52.50%)	251 (40.66%)
16 Tasks	287	440	154 (46.34%)	269 (38.86%)
48 Tasks	286	437	154 (46.15%)	268 (38.67%)

Dhrystone Benchmark

While the performance improvement in individual functions is impressive, it is not representative of the performance improvement that a system will actually experience because that depends on the custom instruction usage in the actual system. However, because these instructions are part of the RTOS, some improvement can be measured without actual applications.

The Dhrystone benchmark is typically used to measure the CPU performance with respect to the amount of work it performs. In recent years, Dhrystone use for comparing CPUs has diminished. However, it is still a viable measure of the amount of work that the CPU can perform.

In our case, we are measuring the performance of the same CPU with different amounts of hardware to offload certain tasks. We are trying to measure the ability of the same CPU to do work when supported by the custom instructions. On the same CPU, the Dhrystone benchmark is affected only by the number of CPU cycles that are available for the software to execute. Therefore, we can use the Dhrystone as a

benchmark to measure of the amount of work that can be performed as a result of adding RTOS custom instructions.

Table 4 shows the Dhrystone benchmark for the system running at 50 MHz with the Dhrystone task running at medium priority. Results are reported for the following four system types with a different number of system tasks because the number of tasks impacts the RTOS primitive execution times:

- *Original RTOS*—The system without any custom instructions.
- *Scheduler modified*—The system includes the scheduler custom instruction.
- *Timer modified*—The system includes the timer tick custom instruction.
- *Combined modified*—The system includes both custom instructions.

The numbers are reported for the system with timer interrupts occurring at 100 Hz (10 ms) and 1,000 Hz (1 ms). The corresponding percentage improvement is shown in parentheses. As required by the Dhrystone benchmark, we did not apply a compiler optimization. As Table 4 shows, the performance is improved by nearly 54%.

Table 4. Dhrystone Tests without Compiler Optimization

# of Tasks	Original RTOS		Scheduler Modified		Timer Modified		Combined Modified	
	100 Hz	1000 Hz	100 Hz	1000 Hz	100 Hz	1000 Hz	100 Hz	1000 Hz
16	3.995	3.257	4.001 (0.15%)	3.305 (1.47%)	4.023 (0.70%)	3.587 (10.13%)	4.045 (1.25%)	3.722 (14.28%)
48	3.486	2.153	3.491 (0.14%)	2.202 (2.28%)	3.592 (3.04%)	3.194 (48.35%)	3.604 (3.38%)	3.316 (54.02%)

Although we collected the data in Table 4 without compiler optimization, we feel that it is not representative of commercial systems. Table 5 presents the same numbers when compiler optimization (O2) is applied. Although the differences are not as large as before, we can see that the Dhrystone benchmark does improve by up to 9.4%. Therefore, the system can perform almost 10% more work when RTOS acceleration is applied.

Table 5. Dhrystone Tests with Compiler Optimization

# of Tasks	Original RTOS		Scheduler Modified		Timer Modified		Combined Modified	
	100 Hz	1000 Hz	100 Hz	1000 Hz	100 Hz	1000 Hz	100 Hz	1000 Hz
16	4.014	3.787	4.023 (0.22%)	3.806 (0.50%)	4.019 (0.12%)	3.85 (1.66%)	4.026 (0.30%)	3.908 (3.20%)
48	3.536	3.159	3.538 (0.06%)	3.18 (0.66%)	3.56 (0.68%)	3.4 (7.63%)	3.566 (0.85%)	3.456 (9.40%)

Hardware Resource Requirements

The base hardware was designed to support 64 system tasks. Table 6 shows the resources (logic cells and registers) required for the scheduler and the timer.

Table 6. Hardware Requirements

Optimization	Logic Cells	Registers
Scheduler	592	80
Timer	3,657	1,184

The hardware data is for a custom instruction that supports all 64 system tasks. However, it is possible to obtain comparable performance while consuming less hardware based on application-specific parameters because the Nios II processor is a soft-core processor and we can create the CPU to match the specific application requirements. The system has two main optimization areas:

- µC/OS-II is extremely customizable and we can restrict the number of tasks that are present in the system. Depending on the application requirements, we can support fewer tasks. Similarly, we can restrict the number of tasks that require hardware. For example, if the system needs to support 16 tasks, we only need 16 counters in the timer custom instruction. This method allows us to achieve the same performance without consuming too much hardware.

- In software, it is convenient to use entities that are multiples of 8 bits. Therefore, the µC/OS-II timer wait value is 16 bits. A system that ticks at a rate of 1,000 Hz waits for a maximum of approximately 65 seconds. A system that ticks at a rate of 100 Hz waits for a maximum of 655 seconds. For a system that typically has small waits, we can reduce the timer tick custom instruction counter width. For example, a 12-bit counter allows a wait of more than 4,000 ticks (4 seconds at 1,000 Hz or 40 seconds at 100 Hz). µC/OS-II already supports delays longer than the time supported by the resolution of the counter. By adapting that area, we can use a smaller hardware counters width, thereby reducing the hardware requirements.

Table 7 shows the hardware resources required for different numbers of tasks and counter resolution for the timer tick custom instruction. The hardware required to support 16 tasks at 12-bit resolution is approximately 20% of the custom instruction (23% logic cells and 20% registers) that supports all 64 tasks at 16-bit resolution.

Table 7. Hardware Results

Tasks	Timer Resolution (bits)	Logic Cells	Registers
64	16	3,657	1,184
64	12	2,834	928
24	16	1,301	432
16	16	917	304
16	12	865	240

This result is possible because the Nios II processor is a soft-core CPU and can be easily customized to closely match the CPU to the application requirements. The custom instructions can be easily parameterized, i.e., SOPC Builder can generate the custom instruction with the correct bit-width and task support the designer requires.

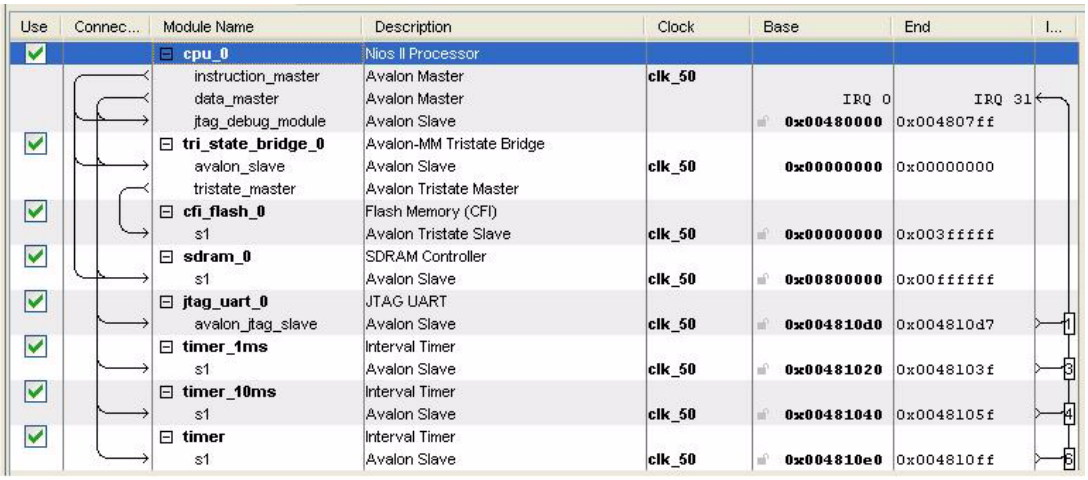
Design Architecture

We constructed our basic Nios II platform with µC/OS-II obtained from the Nios II Integrated Development Environment (IDE) Project Wizard. We developed the custom instruction for the time

management and scheduler modules and then we modified the µC/OS-II source code to work with our custom instructions. We also adapted and used the Dhrystone code that came with the Nios II IDE to benchmark the system's performance improvement.

The main CPU is the Nios II processor, which was clocked at 50 MHz as the reference design. The design uses the Nios II/s processor, flash memory, and SDRAM controllers, which are connected to the Nios II processor by a tri-state Avalon® bridge, `ext_ram_bus`. We used additional modules, such as a timer, JTAG UART, etc., to run and debug the Nios II processor. Figure 5 shows the hardware modules in SOPC Builder.

Figure 5. Nios II Processor and Hardware Modules in SOPC Builder



We designed the following two custom instructions to make the system faster:

```
#define ALT_CI_TIMER (n, A, B) __builtin_custom_inii(0x20+(n&((1<<5)-1)), A, B)
#define ALT_CI_SCHD (A, B)  __builtin_custom_inii(0x0, A, B)
```

The custom instruction depends on a number of input parameters that cannot be passed to it directly. Additionally, the design requires functions to load and store values inside the instruction. Therefore, we set up the instruction to perform a number of different operations. The function is selected by changing the value of the prefix parameter. Because the custom instruction stores values, it requires two clock cycles to run. The time management custom instruction has the following operations:

- Set delay for task n.
- Clear delay for task n.
- Get delay for task n.
- System tick reduces all active counts by 1.
- Update suspended tasks and read status of tasks 0 to 31.
- Update suspended tasks and read status of tasks 32 to 63.
- Clear result registers.
- Update active bit for all tasks.

- Mark task n as suspended.
- Mark task n as unsuspended.

The scheduler custom instruction has the following operations:

- Put a task into ready list.
- Remove a task from ready list.
- Find the highest priority task from ready list.
- Return the ready group value.
- Clear all the task bits from ready list.
- Get the bit mask from 0 to 7.
- Get the bit information from OSUnMapTbl.
- Get one element value of OSRdyTbl[].

Design Features

Our design has the following main features:

- *Significantly reduces RTOS overhead*—While RTOS usage is considered essential in many modern CPU-based systems, the overhead can be significant. Our approach helps contain it.
- *More time for user tasks*—As demonstrated, properly using these instructions results in more time for user tasks, thereby allowing the same system to do more work at the same frequency. Alternatively, the system can be clocked at a lower frequency.
- *Improved determinism*—When using custom instructions, the execution time does not increase at a very steep rate as the number of system tasks increases. This situation provides better scalability and improved determinism because the custom instruction consumes almost the same amount of time, regardless of the number of tasks.
- *Potentially reduces interrupt latency*—Code that uses the custom instructions often runs in system areas that are either in ISRs or in areas that run with disabled interrupts. Reducing execution time of these modules reduces the system's worst-case interrupt latency.
- *Parameterized instructions*—The design can be easily parameterized, allowing the designer to tailor the custom instructions to the target application's specific requirements. This process is made even simpler by including SOPC Builder options such that the RTOS design is also generated when producing the designer's CPU software development kit (SDK).
- *Drop-in optimization*—Application-specific hardware/ software partitioning is a time-consuming task that requires many design and test iterations. In CPU-based software-intensive systems that require an RTOS, our approach offers an attractive method to improve the system performance by simply reducing the RTOS overhead, an area that system designers seldom touch. Additionally, because the custom instructions are independent and parameterized, the designer can utilize unused FPGA resources in a final system for RTOS acceleration (the hardware requirements can be managed by selecting the specific instruction, the number of tasks, and their timer solutions, etc.), which has a positive impact on the whole system. Because the RTOS and instruction

generation can be automated, this option can be provided as a customizable, pre-verified, drop-in optimization.

- *Wide applicability*—Because our work makes more CPU time available to user tasks, it is applicable to any system that is based on a customizable CPU on an FPGA, regardless of the target application. All software-based systems can benefit from our work.

Conclusion

This project builds on work that was done previously in our research group using the Nios II processor and an older version of µC/OS-II. Since then, the Quartus II software and SOPC Builder have evolved significantly, making a number of tasks easier. It is interesting to note that while the participants in our project group were mentored and supported by the people who did the earlier research, one of us used the Nios II processor for the first time in this project.

The project was a cumulative learning and knowledge enrichment experience regarding the Nios II processor and FPGAs. In addition to the fact that we achieved RTOS acceleration, most applications can reap immediate benefits when run on the modified RTOS.

In our opinion, the Nios II processor is very useful for embedded system engineers and is also an excellent tool for research engineers. The designer can easily change external peripherals and interfaces from within SOPC Builder, making a seamless interface between processor and hardware logic. Instead of passively adapting to the hardware processor, we can customize both hardware and software, particularly when using Nios II custom instructions that make the system much more flexible than using standard processors. The Avalon bus quickly helped us connect all the modules required in our system and made it just as easy to improve the system later when we realized that we lacked something.

While implementing the project, we learned the following things:

- SOPC Builder is a very flexible, powerful tool that allows even a software engineer to learn and design a hardware system quickly and efficiently on any Altera FPGA.

- We spent some time understanding the differences between the Nios and Nios II processors because we migrated some of the designs from a previous project.

- We found the Nios II IDE Project Wizard examples very useful because IDE generated the µC/OS-II and Dhrystone elements we used in this project.

- We found that the µC/OS-II source code is not copied to each project folder but instead uses the main copy in the Nios II IDE root folder, which made it difficult for us to switch projects between our optimized and regular RTOS. This setup is most likely due to the fact that most people treat the RTOS as a software entity that is not touched as part of the system design process.

- We were able to implement the accelerated RTOS using two custom instructions for the competition. It is unfortunate, however, that we did not have sufficient time to implement an event control block (ECB) custom instruction, which would have further improved the system performance (particularly for system synchronization and communication). In the future, we hope to integrate this custom instruction into the Nios II IDE, allowing developers to utilize the instruction with the Nios II Project Wizard. This modification will make optimization even more seamless. Additionally, we want to explore the C2H Compiler in the future to see if it can further ease the process of integrating modules for RTOS acceleration.

References

The references are for our group's relevant research publications.

[1] T F. Oliver, D L Maskell, "Accelerating an Embedded RTOS in a SOPC Platform," Proceedings of the annual technical conference of the IEEE Region 10 (TENCON), November 21 - 24, 2004.

[2] M Sindhwani, T Oliver, D L Maskell and T Srikanthan, "RTOS Acceleration Techniques - Review and Challenges," Proceedings of the Sixth Real-Time Linux Workshop, Singapore, pp. 123-128, November 2004.

[3] Z Jin, M Sindhwani and T Srikanthan, "RTOS Acceleration on Soft-core Processors Using Instruction Set Customization," 2004 IEEE International Conference on Field Programmable Technology (FPT 2004), Australia, pp. 371-374, December 2004.

[4] M Sindhwani and T Srikanthan, "Framework for Automated Application-Specific Optimization of Real-Time Operating Systems," Fifth International Conference on Information, Communications and Signal Processing (ICICS 2005), Thailand, pp. 1416-1420, December 2005.