THE RAVENSCAR-COMPLIANT HARDWARE RUN-TIME (RAVENHART) KERNEL

by

ANNA SILBOVITZ

Sc.B., Electrical Engineering
Brown University, 2001

SUBMITTED TO THE DEPARTMENT OF AERONAUTICS AND ASTRONAUTICS IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN AERONAUTICS AND ASTRONAUTICS
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

FEBRUARY 2004

Signature of Author: [illegible]
Department of Aeronautics and Astronautics
January 30, 2004

Certified by: [illegible]
I. Kristina Lundqvist
Charles S. Draper Assistant Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by: [illegible]
Edward M. Greitzer
H.N. Slater Professor of Aeronautics and Astronautics
Chair, Committee on Graduate Students

The Ravenscar-compliant Hardware Run-Time (RavenHaRT) Kernel

by

Anna Silbovitz

Submitted to the Department of Aeronautics and Astronautics
on January 30, 2004 in Partial Fulfillment of the
Requirements for the Degree of Master of Science in
Aeronautics and Astronautics

ABSTRACT

Real-time embedded systems are increasingly becoming the foundation of control systems in both the aerospace and automotive worlds. This class of systems has to meet three requirements: strict timing constraints on operational behavior, limited resource availability, and stringent certification standards. The heart of any embedded system is its run-time system (RTS), which provides resource management, task creation and deletion, and manages inter-task communication. The traditional Ada RTS does not provide deterministic behavior. In order to meet the requirement of a minimal, deterministic RTS, a formal model based on the Ravenscar profile of Ada95 was developed by Professor Kristina Lundqvist in 2000. This formal model forms the basis of the work carried out in this thesis.

This thesis aims to leverage the reliability and efficiency of programmable hardware to implement a run-time kernel called RavenHaRT. The kernel was designed to support Ravenscar compliant Ada95 code and provides task creation, task scheduling and inter-task communication capabilities. The timing properties embedded in the formal model are captured in terms of kernel performance within the hardware.

The kernel was implemented using a Xilinx Virtex-II Pro FPGA. The results from testing demonstrate that the hardware kernel has the expected behavior and can interface correctly with software code.

Thesis Supervisor: I. Kristina Lundqvist
Title: Charles S. Draper Assistant Professor of Aeronautics and Astronautics

Acknowledgements

I would first like to thank my advisor Professor Kristina Lundqvist, for all of the guidance and support you have given me. You encouraged me to consider ideas and problems I would not have otherwise, and I appreciate all of our discussions and what they have taught me.

Thanks also to MIT Lincoln Laboratory for making it possible for me to complete this degree through the Lincoln Scholars Program. I especially would like to thank my Lincoln advisor Herb Viggh, and everyone else in my group at work who allowed me to make my thesis my top priority.

I also received lots of support from the members of our team in Sweden. Thanks to Lars Asplund and his family for sharing their home with me when I visited. And a big thanks to Johan Furunäs: I probably couldn't have finished this thesis without your expertise. Thank you for answering my endless stream of questions.

Lastly, I would like to thank all of my family and friends, for putting up with me during this process. A special thanks to Katie, for all of the great conversations we've had, related to schoolwork or otherwise. And also to Justin – you have always been there willing to listen, giving support and advice. Thank you!

Table of Contents

Chapter 1
Introduction

An embedded system consists of computer hardware and software components that have a dedicated purpose within the context of a larger engineered system. Embedded systems are used in a wide variety of applications, including those that are safety-critical or mission-critical. Both safety- and mission-critical systems are systems that will result in a loss event if they fail, that may include equipment, money, or human life. Safety-critical systems typically have hard real-time constraints; a missed deadline may result in system failure. Therefore, these systems must be thoroughly analyzed, to ensure that all necessary deadlines can be met.

The heart of an embedded system is the run-time system, also called a real-time kernel (RTK). The RTK handles the scheduling of the software tasks that run on the processor. The timing properties of the RTK directly influence the timing properties of the embedded system. Therefore, it is essential that the RTK be thoroughly analyzed and designed so that it is deterministic and predictable.

Designing embedded systems for safety-critical applications is a difficult task. Such a system involves many aspects: deciding on an RTK that will suit the system, determining how to model and analyze the system, and planning how to do fault tolerance and fault recovery while the system is running. To aid in the design of safety-critical systems, the Gurkh environment is being developed. It includes a custom Ravenscar-compliant hardware RTK that is suitable for use in safety-critical applications, support tools for system analysis, and a component that aids in fault recovery.

The term *Ravenscar-compliant* refers to the Ravenscar profile of Ada95. Ada95 (hereafter referred to just as Ada) is a software language that can easily be restricted into subsets that are suited for different applications, such as safety-critical systems. The Ravenscar profile is a subset of the language that, when used, will ensure deterministic software behavior.

A Ravenscar-compliant hardware RTK, called the RavenHaRT Kernel (*Raven*scar-compliant *Ha*rdware *R*un-*T*ime *Kernel*), has been designed for use in safety-critical systems that use the Ravenscar profile. RavenHaRT is implemented in the hardware description language VHDL, so that it can be implemented on a Field Programmable Gate Array (FPGA). It is therefore very different from traditional RTKs, which are implemented in a software language and run directly on the processor, and are typically not fully predictable. The hardware implementation should provide faster response times, greater reliability and predictability, and better utilization of available power resources.

This thesis is organized as follows. The next three chapters provide additional background and motivation for the project. Chapter 2 describes the Gurkh project, including the different tools being developed to support the design and implementation of embedded systems using the Ravenscar profile, and how the RavenHaRT Kernel fits into the process. In chapter 3, the motivation for choosing Ada and Ravenscar is provided, along with a summary of the features of the Ravenscar profile that influence the RavenHaRT Kernel's design. Chapter 4 discusses hardware RTKs, the benefits of using them instead of software RTKs, and examples of hardware RTKs implemented as part of other projects.

Chapters 5 through 7 deal with the different stages of designing and implementing the RavenHaRT Kernel. Chapter 5 explains formal modeling, and why it is an important part of analysis. A formal model of the kernel is presented, which was used as the basis for the implementation in VHDL. However, there are many differences between how a model behaves and how VHDL code behaves, and several of these issues are also addressed. Chapter 6 provides a complete description of the VHDL RavenHaRT Kernel, including technical documentation of the code. Chapter 7 explains the tests that were performed on the kernel.

Chapter 8 contains a discussion of future work that could be carried out to design a more optimized version of the kernel. The Appendix holds all the RavenHaRT Kernel VHDL code.

Chapter 2
The Gurkh Project

The Gurkh project [AL03] is a joint effort between the Massachusetts Institute of Technology and Mälardalen University in Sweden. The aim of the project is to provide an integrated environment for implementing embedded systems for use in safety-critical applications. An overview of the environment is shown in Figure 1.

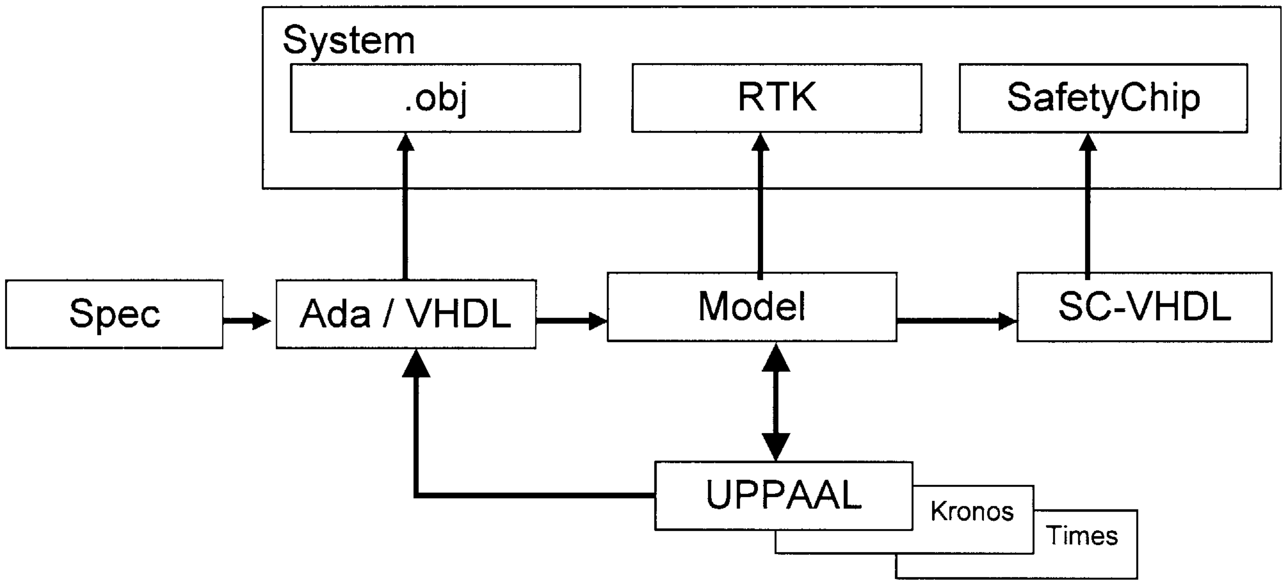

Figure 1. Overview of the Gurkh project

The project is composed of tools to support analysis of a design, and a system to support implementation. It has been created to be used with code written in either VHDL or the Ravenscar profile of Ada. The Ravenscar profile is chosen because of its usefulness in safety-critical systems; this decision will be discussed in more detail in chapter 3. At the current time, the Gurkh environment supports a single-processor system, but future work includes expanding it to support systems with multiple processors.

To take advantage of the Gurkh environment, users could start with specifications for a new project and develop Ada and/or VHDL code to satisfy those specifications. Alternatively, if they have a system that is already implemented, they could use Gurkh to analyze the design to confirm that it has correct behavior (or rewrite the code if it does not). Previously written code must be annotated with best and worst case execution times so that its timing behavior can be analyzed by the tools in Gurkh. Once the code has been developed, it must be thoroughly tested for correct functional behavior. To verify correct non-functional behavior, such as timing properties, the entire system (application tasks and the RTK) should be analyzed by using formal methods. The benefits of using formal methods and verification tools such as model checkers are described in chapter 5. A goal of the Gurkh project is to provide tools that automatically converts VHDL or Ada code into syntax for the input languages for different verification tools. At the current time, tools are being developed for the model checker UPPAAL [Upp], but future work will include conversion to additional verification tools such as Kronos [Yov97] and Times [AFM+02]. The conversion from VHDL or Ada code to a formal model usable in a verification tool is a two-step process. First, the code is translated into a readable intermediate format, so that users can easily visually analyze the behavior of their system. The second step is to convert the readable models into a syntax that can be read by verification tools. The decision was made to include different verification tools so that users will have flexibility, and also so that different verifications can be performed. Different tools verify different properties of the system. If multiple tools are used for analysis, and if they all agree on the validity of the system, users can have more confidence that the system is behaving as expected than if they had used just one tool. Note that in Figure 1, there is an arrow going from the block labeled 'Ada/VHDL' to the block labeled 'Model', but also one from the verification tools back to 'Ada/VHDL'. This is because writing code and modeling and verifying it is an iterative process. Errors could be found in the model, and then the code would need to be changed. Whenever changes are made, model verification (and testing) must be performed again.

Once the system model has been shown to be correct, the RTK can be extracted from it. For a hardware implementation of the RTK, VHDL code is written. This process is discussed in chapter 5. Note that different design projects do not necessarily need different, unique RTKs. If users have an application that will work with an RTK that has been previously designed, analyzed, and implemented in VHDL using the Gurkh process, it can be reused. The modeling step must be adjusted so that the new application is analyzed together with the existing kernel model, ensuring that the whole system still has the desired behavior. The RavenHaRT Kernel is being designed as part of the Gurkh project, so that users will already have an RTK that has been modeled and shown to be suitable for use in safety-critical systems.

The physical system is shown in the 'System' block in Figure 1. A specialized board will be developed to support the implemented design. The board will contain a Xilinx Virtex-II Pro FPGA [Xil02], which has both FPGA logic and an IBM PowerPC 405 (PPC) [PPC01] on-chip. The board will also contain whatever other peripherals are needed for the application, such as additional RAM. At Mälardalen University, a custom board has already been designed for use in the Gurkh project development. At MIT, the Memec Design Virtex-II Pro Development Kit [Mem03] is being used.

The application is compiled into machine code (or, if the processes are written in VHDL, they are synthesized and implemented on an FPGA). The machine code must run on the PPC. The RavenHaRT Kernel, which is the main focus of this thesis, is written in VHDL and implemented on the FPGA. When the application must make system calls, instead of making them to a traditional software operating system running on the PPC, it makes them to the RavenHaRT Kernel. The RavenHaRT Kernel manages the different software processes, and determines what tasks the PPC should run at any given time.

The final part of the system is the Safety Chip. The Safety Chip will monitor the other parts of the system (application and RavenHaRT Kernel) to ensure that all processes execute within their specified execution time limits. It raises a flag if unexpected system behavior occurs. The Safety Chip is developed from the formal models of the system. Using software tools, the necessary characteristics of the formal models can be extracted and VHDL for the Safety Chip can be generated. The Safety Chip should know what all valid system states are as well as system timing properties, so that it can identify any unexpected behavior, from any source. The design of the Safety Chip is still under development.

Chapter 3
Software Development in Ada

As discussed in [Shaw01], real-time software languages need features that other languages do not have. These include support for handling time and concurrency and exception handling. Also, the code produced should be predictable. The language must be able to perform actions such as setting and reading a timer, and delaying a task until an absolute or a relative time. It must support communication and scheduling among parallel processes. Predictability should exist with respect to both timing and functional behavior.

Ada was specifically developed with these features in mind. It has been proven effective over years of use, and is therefore a good choice for safety-critical embedded systems, particularly when a subset is used. This chapter discusses the features of Ada in general, and then the Ravenscar profile specifically.

3.1 Ada in Real-Time Safety-Critical Systems

Information concerning the history and use of the Ada language can be found at the Ada Information Clearinghouse [Ada]. In the mid-70s, the United States Department of Defense sponsored a competition to design a software language to be used in their applications. The result was Ada83, which became accepted as an ANSI standard in 1983. Ada95 was later released as an update to the language, with additions such as object-oriented properties and more tasking functionality. When Ada95 was approved by the International Organization of Standards (ISO), it became the first object-oriented language to be accepted as a standard.

Ada was designed to be a widely applicable programming language, with a large range of functionality. It was also designed to be highly reliable; errors are automatically caught in compiling, linking, and in the Ada Run Time Environment [Rei97]. The compilers undergo thorough testing using the Ada Compiler Validation Capability test suites so that they are verified to work correctly. Ada is structurally safe, and unlike other commonly used languages like C and FORTRAN it is type safe. It also has well defined semantics. These are all desirable features for a language used in safety-critical systems, but they still do not ensure a predictable system. To achieve this goal, Ada was designed to be easily reduced into subsets, or profiles. A profile restricts the features of the language that can be used. When chosen correctly, a profile can retain the needed functionality for a specific system and make the system predictable. More information on how to use Ada can be found in [Ada99].

When compared to other languages, Ada has been shown to be highly suitable in safety-critical applications [CGW91]. It satisfies many of the requirements for safety-critical systems, such as those listed above. It has also shown to be more cost-effective in terms of both time and money when compared with C [Zei95]. Ada has been used successfully in many projects, including the Boeing 777, the New York City Subway system, various banking systems, and space systems designed by NASA and the European Space Agency [Ada]. Since it has proven through use to improve reliability and reduce software and maintenance costs, it is an ideal choice for safety-critical systems and the Gurkh project.

3.2 The Ravenscar Profile

Despite the success of Ada, it has many features that are complex and are not predictable. Hard real-time systems must be proven to be predictable. Ada can be restricted into profiles, that when used produce predictable, deterministic code. Restricted profiles can provide the following benefits [BDR98]:

- increased efficiency from removing features with high overhead
- reduced non-determinacy
- simplified RTK
- only has features with a formal underpinning
- only has features that allow timing analysis

To allow for tasking and concurrency in safety-critical applications, a profile was defined at the Eighth International Real-Time Ada Workshop in 1997 for high-integrity, efficient, real-time systems [BDR98]. It is called the Ravenscar profile.

The Ravenscar profile allows for systems that use tasking to express concurrency. The complete definition of the Ravenscar profile is found in papers such as [BDR98] and [BDV03], but the characteristics important to the kernel design are summarized here. In a system using

Ravenscar, there are a fixed number of tasks. Tasks cannot be dynamically allocated, and they cannot terminate, so each is designed as an infinite loop. Each task has a single invocation event, but can be invoked an infinite number of times. Task invocations can either be time-triggered (periodic delays) or event-triggered (started by a signal from another task). Tasks can only interact through shared data. Either preemptive or non-preemptive fixed priority scheduling can be used, and the system can be analyzed for both functional and timing behavior. [BDR98]

The Ravenscar profile uses the Real-Time package instead of the Calendar package, so a high-fidelity real-time clock can be used. Only absolute delays are used for periodic time-triggered tasks: the *delay until* statement is valid, but the *delay* statement is not.

Protected Objects are used to share data between tasks, and to synchronize tasks. Protected Objects may have three different parts, the Protected Procedure, the Protected Function, and the Protected Entry. Protected Objects may have multiple Procedures and Functions, but in Ravenscar they may only have one Entry. The Entry is used for event-triggered task invocation. A task may become suspended when calling an Entry, depending on the value of a Boolean *Barrier* signal. When this occurs, the task is put on an *Entry queue.* In Ravenscar, only one task is allowed to queue on an Entry at a time, so a simple signal can be used instead of an actual queue. When a different task calls a Procedure routine, the *Barrier* signal of that Protected Object is checked before it exits. If a task is on the same Protected Object's Entry queue, and the *Barrier* value is correct, the remainder of the Entry code is executed in the context of the task calling the Procedure, and the task suspended on the Entry queue is released. The Procedure and the Function have the same properties in terms of timing behavior and execution handling, but the Function does not evaluate the *Barrier* or change the value of it and therefore cannot release a task on the Entry.

All tasks and Protected Objects have priorities, and scheduling of tasks in the RavenHaRT Kernel is done using priority based preemptive scheduling. For tasks with the same priority, *FIFO within priority* is used. Protected Objects have *Ceiling Locking.* This means that any task able to call a Protected Object must have a priority equal to or lesser than the priority of the Protected Object. When the Protected Object is called, the calling task's priority is raised to that of the Protected Object. This ensures that once one task has access to a Protected Object, no other task is capable of performing an interrupt and then accessing that same Protected Object.

This prevents deadlocks between tasks using the same Protected Object. This is also the only case where task priority is allowed to change.

Static timing analysis can be performed on code written using the Ravenscar profile, and so can schedulability analysis. Use of the profile allows for the design of deterministic hard real-time systems. Additionally, the Ravenscar profile can be mapped to a small RTK that can be used instead of Ada's full run-time system. This small RTK would only support the restricted behavior of the Ravenscar profile. The RavenHaRT Kernel is such a kernel, and will be described in more detail in the following chapters.

Chapter 4
Hardware Real-Time Kernels

Real-time embedded systems have strict timing requirements, and they must be both predictable and deterministic. This creates a problem when using a traditional software operating system (OS), particularly when multitasking must be done in a system with a single processor. The OS itself also runs on the processor. If multiple tasks exist, the OS interrupts the processor at regular intervals by performing a clock-tick interrupt. When interrupted in this manner, the processor must stop the task it was running so that the OS can check to see if another task should be running instead, and then resume. Even if the same task continues to run, the interrupt still must be made. This results in less processor time for actually running tasks [Kel03]. Additionally, the time taken for actions such as scheduling varies with the number of tasks, which makes the system less deterministic.

To save processor time and increase determinism, many of the behaviors of a software OS can be implemented in hardware. Good actions to place in hardware include: task handling (such as creation, deletion, and scheduling), synchronization (such as semaphores, flags, and resource sharing), and timing (such as delays, periodic starts, watchdogs, and interrupts). When all task management is performed in hardware, scheduling is done while the processor is running tasks, so scheduling does not occupy processor time. The only processor interrupts that need to be made occur when a task is actually changing (a 'task-switch' interrupt). This eliminates the need for clock-tick interrupts, and this change alone can give the processor up to 20 percent more time for running tasks [Kel03]. This potential speed increase, and the desire for a more deterministic system, resulted in the decision of implementing the RavenHaRT Kernel in hardware.

4.1 Summary of Related Work

Other research groups and organizations have implemented hardware RTKs. These RTKs include FASTCHART (A Fast Time Deterministic CPU and Hardware Based Real-Time Kernel), FASTHARD (A Fast Time Deterministic Hardware Based Real-Time Kernel), RTU (the Real-Time Unit), ATAC (Ada Tasking Coprocessor), and Silicon TRON.

FASTCHART [Lin91][LS91][LSF95], FASTHARD [Lin92][LSF95], and RTU [AFLS96][LSF+98][LSF95] are a series of deterministic hardware RTKs developed by research groups in Sweden under the direction of Lennart Lindh. All three have similar properties. FASTCHART includes a specialized deterministic processor with its own instruction set as well as an RTK, while FASTHARD is only an RTK that is designed to be used with almost any standard processor. The RTU is also a hardware RTK designed similarly to FASTHARD, except that it can support multiple processors. All of these RTKs are implemented on either an FPGA or an Application Specific Integrated Circuit (ASIC). Tasks running on these systems can only have four states: waiting to execute, executing, in delay, and terminated. Therefore the only task-related system calls are activate, terminate, and delay, and these calls are handled by the RTKs. Other behaviors of the RTKs include scheduling tasks and performing task switches.

ATAC [Roo91][ATAC95] was originally developed by Joakim Roos, and then further improved (ATAC 2.0) in conjunction with the European Space Agency. The original goal of the first version was to move the support for rendezvous from the Ada83 run-time system into a VLSI coprocessor. However, to reduce the passing of parameters between software and hardware parts and achieve the desired speed-up, all tasking functionality and support was incorporated into the hardware. A custom instruction set was developed for interfacing with a processor. ATAC 2.0 uses an ASIC to implement the complete Ada83 tasking semantic and also includes semaphores, *delay until*, priority inheritance, interrupts, and task scheduling among other functions.

The Silicon TRON [NUI+95] is a VLSI RTK chip. The goal of the design was to speed up system calls and scheduling. Hardware functions include task management, task-dependent synchronization, synchronization communication, and interrupt management. Functions such as timing and system management are still in software. The RTK can work with any processor.

The RavenHaRT Kernel has features similar to these previous designs. The decision was made to implement it on an FPGA because FPGAs can be easily reprogrammed, as opposed to using an ASIC or VLSI technology. ATAC provides a good example of how the Ada83 run-time system can be transferred to hardware; the RavenHaRT Kernel is similar except that it only supports the Ravenscar tasking profile of the updated Ada95 language. Even though FASTCHART, FASTHARD, and RTU are not Ada-specific, they show how a system with restrictions on task behavior works. The RavenHaRT Kernel is even more restricted; tasks cannot terminate, and can only be activated during elaboration. It currently is designed to support single processor systems, but future work may make it capable of supporting multiple processors. Investigating the functionality of the RTU would help with this design.

4.2 Evaluation of Hardware RTKs

When implementing an RTK in hardware, time can be saved in scheduling and with the removal of clock-tick interrupts, but the overhead from buses, hardware arbitration, and hardware interrupt latency is important [AFLS96]. For example, a scheduler in hardware saves processor time and is more deterministic, since only the time for the task-switch needs to be considered, not the time for the scheduling itself. However, the time for the task-switch is bus dependent.

Several of the RTKs mentioned in section 4.1 have been examined for timing, and compared with similar software systems. A hardware coprocessor (a scheduler only) based on the RTU underwent benchmarking tests [Fur00]. The coprocessor was located on a PCI bus, and the results showed that the PCI bus accesses were costly in terms of time. However, this problem could be solved by putting the coprocessor in a better location, such as integrating it with the processor, or putting it directly on the processor bus. Overall, the comparison showed that speed-ups were due to the faster hardware scheduler and lack of clock-tick interrupts, while system calls were still quicker in software because of the slow bus accesses.

Similar results have been found examining multiprocessor systems, such as the SARA system [SAN+03][LK99]. The SARA system uses an RTU similar to the one described in section 4.1. The hardware RTK was up to 2.6 times faster than the same implementation in software (using Assembler, C and C++). Comparisons were made for creating tasks, task-switch, communication bandwidth, and communication latency. Results showed that creating tasks was faster in hardware, and in software the time also increased with increasing number of tasks. Task-switch time was faster in hardware. Communication bandwidth was quicker in software between two tasks on the master node, but faster in hardware for communication across node boundaries. Communication latency was consistent in hardware, and in software was either slower or faster depending on the location of tasks.

The RavenHaRT Kernel handles scheduling, task creation and delay, sending interrupts to the processor for a task-switch, and it keeps track of resource use (Protected Objects) for monitoring and timing analysis. All of these actions are more deterministic in hardware, so the kernel has been implemented in VHDL. Bus access time is an issue, but as mentioned in chapter 2, the kernel will be implemented on a Xilinx Virtex-II Pro, which contains both FPGA logic and an IBM PPC on-chip. This setup allows the kernel to be placed directly on the processor local bus (PLB), which was one of the suggested improvements in the benchmark tests on other systems. In the future, the RavenHaRT Kernel may also be used in a multiprocessor system, and the benefits of using hardware will be advantageous in that situation as well, as shown in the analysis of the SARA system.

Chapter 5
Modeling the Kernel

Formal methods are mathematically based languages, techniques, and tools for specifying and verifying software and hardware systems [CW96]. They are used to evaluate system properties, such as functionality and timing. According to [CW96], there are two main techniques for applying formal methods, *model checking* and *theorem proving.*

In model checking, a finite model of the system is created. Then, the model is checked to see if certain properties hold. The check is done by performing a search of the entire state space. This can create state space explosion problems in very large systems, although many model checking tools can handle hundreds of state variables. Model checkers typically use one of two approaches. The first is temporal model checking, where specifications are given in temporal logic, and systems are modeled as finite state systems. In the second approach, both the specifications and the system are modeled as automata. They are compared to determine correct behavior. Many tools exist today that perform model checking automatically and quickly, and the process produces counter-examples for when a property fails.

In theorem proving, the system and its properties are expressed as logical formulas. Infinite state spaces are allowed, and automatic tools exist to aid in analysis. However, even the automatic tools require human interaction, so the process can be slow and error-prone.

A Ravenscar-compliant kernel was first modeled in previous work [LA03], and this model was then refined into the RavenHaRT Kernel in VHDL. The model is a collection of automata in the correct syntax for the model checker UPPAAL. Model checking was chosen to analyze the system since UPPAAL provides a fast, automatic way to check system properties. The first part of this chapter provides an overview of the UPPAAL tool, and the second part describes the kernel model in detail. The third section highlights several of the issues that were addressed when refining the model into VHDL code.

5.1 UPPAAL

The model checker UPPAAL [Upp] was developed jointly by Uppsala University and Aalborg University. UPPAAL is a tool for modeling, simulating, and verifying real-time systems. It can be used to analyze systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks [PL00][LPY97].

As described in [LPY97], UPPAAL actually consists of three parts: a description language, a simulator, and a model checker. The model checker checks for invariant and reachability properties of the system. Reachability testing determines whether certain combinations of control-nodes and constraints are reachable from an initial configuration. UPPAAL produces diagnostic traces to show how properties are (or are not) satisfied.

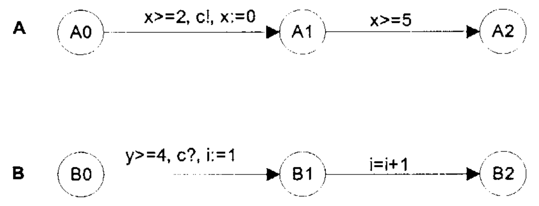

Figure 2. Example UPPAAL Automata

Models are created using timed automata, which are finite state machines with clocks. They communicate through channels and shared variables. All channels and variables are global in UPPAAL. An example of two UPPAAL automata, A and B, is shown in Figure 2. There are two clocks, x and y, one channel c, and one variable i. Clocks in UPPAAL are synchronous and count up from zero. A channel provides synchronization between two automata; the transition can only be made when both automata can transition. The channel c appears as *c!* on automaton A and *c?* on automaton B. A guard is a condition on either a clock or integer variable that must be satisfied for a transition to occur. In automaton A, the transition from A0 to A1 can only be made if x is greater than or equal to 2 and if synchronization can occur on channel c. Automata are able to transition when guards are satisfied or channels are valid. If a transition has a guard and a channel, both conditions must be met for the transition to occur. Assignments can also be made to either clocks or integer variables. A state can be declared as either committed or urgent; committed states are marked with a ‘c’ and urgent states with a ‘u’. When an automaton is in a committed state, time cannot progress if it is possible for control to leave that state, and any action that occurs must involve that component (the component is ‘committed’ to continue). An urgent state is similar to a committed in that if control can leave the state, it must before time progresses, but other actions that do not involve that state (and that happen in zero time) may occur first [LPY97].

A model is checked by creating assertions. Assertions are statements declaring properties that are or are not valid in the system. For example, an assertion may declare that a certain combination of states is never to be reached. Or it may specify valid combinations of states at certain points in time. UPPAAL automatically checks if all assertions made are true for the modeled system; it produces counter-examples for those that are not true. A drawback to this method is that it involves human interaction; the user must determine the list of assertions for UPPAAL to check. So UPPAAL's check for correct behavior is limited to what the user believes is correct behavior for the system being modeled.

5.2 The Ravenscar-compliant Kernel Model in UPPAAL

The model of the kernel [LA03] in UPPAAL (Figures 3-9) supports only Ravenscar-compliant operations. The automata created include the Ready Queue, the Delay Queue, one Protected Object consisting of a Protected Entry and one Protected Procedure, a System Clock, and an Idle Task. The Protected Function routine was not modeled since its behavior is almost identical to that of the Protected Procedure, as discussed in section 3.2. Three tasks are modeled as well, so a system model can be analyzed.

5.2.1 The Ready Queue

The Ready Queue automaton handles creation of tasks and priority based preemptive scheduling (described in section 3.2). When tasks are created, they are placed in an array called *RQ*. The array is indexed by the task's identification number. For suspended tasks, the array holds a zero; otherwise, it holds the task's priority.

Figure 3 shows the Ready Queue automaton. Tasks are created by synchronizing on the channel *Runable* during the elaboration phase, which is modeled to occur in zero time. During this elaboration, tasks synchronize on *Run* as they are created. The variable *Tcpu* holds the ID of the task currently running, and *Pcpu* holds the priority of that task. If a new task of higher priority is created, it will preempt (on the channel *Preempt*) the most recent task to synchronize on *Run*. At the end of elaboration, the highest priority task has synchronized on *Run* and would begin to execute. If a task becomes suspended, either by calling a Protected Entry under the right conditions (see section 5.3.3) or by making a *delay until* call, it will synchronize on the channel *Suspend*. Then the Ready Queue automaton searches through the array *RQ* to find the highest priority task to run. Note that this search is done through the use of urgent states (marked with a 'u' on the state node), so that the process of looping through the array can be performed without the passage of time.

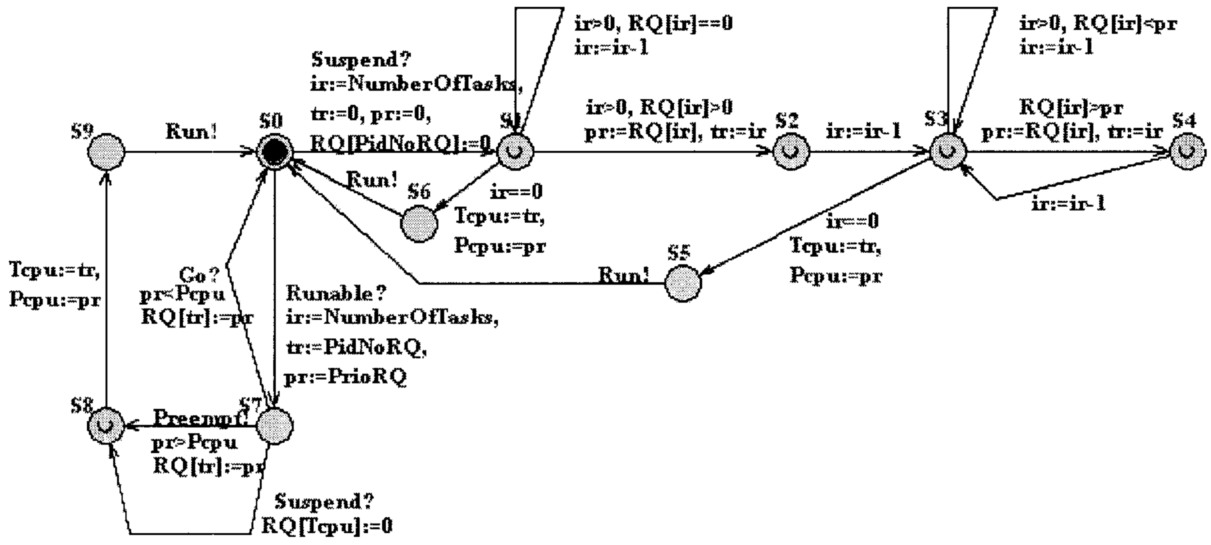

Figure 3. The Ready Queue in UPPAAL

5.2.2 The Delay Queue

The Delay Queue automaton, shown in Figure 4, handles *delay until* calls. Control remains in state S0 until a task makes a *delay until* call. The task making the call has its ID stored in *PidNoDQ* and the time to which it will delay stored in *DUntil*. The task synchronizes with the Delay Queue automaton on channel *Q*. The variable *len* keeps track of how many tasks are currently delayed. If none are already delayed, *PidDQ* is set to *PidNoDQ*, *t* is set to *DUntil*, and control returns to state S0. If at least one task is already delayed, a new *delay until* call brings control to the urgent state S2. If *DUntil* holds a later time than the value in *t*, *DUntil* is placed into the Delay Queue array *DQ*. Like the *RQ* array, the *DQ* array is indexed by the task's ID. If *DUntil* represents a time sooner than that in *t*, the value in *t* and its associated task ID are put into the *DQ* array; the new task and its delay time are saved in the variables *PidDQ* and *t*. This method makes sure that the task with the shortest delay time is always held in the variables *PidDQ* and *t*. The automaton can easily check these variables, to determine when the delay time is up, without having to access the array *DQ*.

As described in the previous section, a task making a *delay until* call will not only activate the Delay Queue automaton, but the task will also synchronize on the channel *Suspend* with the Ready Queue automaton (Ready Queue states S0 or S7) so that the Ready Queue knows that it should not run.

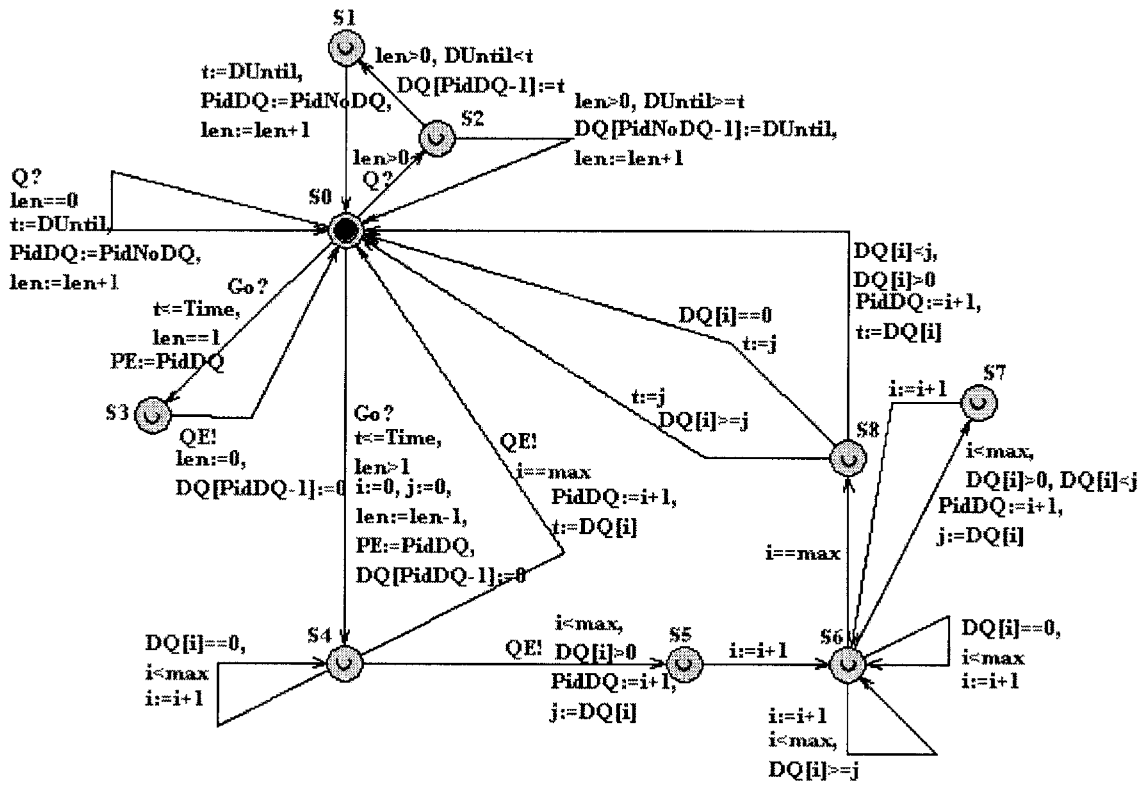

Figure 4. The Delay Queue in UPPAAL

When a task's delay time is up, the task synchronizes with *Runable* in the Ready Queue automaton (Ready Queue state S0) so that the Ready Queue knows that it can run again. Also, in the Delay Queue automaton, the values in *PidDQ* and *t* must be changed. If no other tasks are delayed (*len* == 1), the Delay Queue automaton transitions from state S0 to S3 and then back to

S0, and in the process sets *PidDQ* and *t* equal to zero. If other tasks are delayed (*len* > 1), the Delay Queue automaton takes the path begun by the transition to S4. It must search the *DQ* array for the task that will next wake up from a delay. When it finds this task, the task's ID and delay time are removed from the *DQ* array and placed in the variables *PidDQ* and *t*. The Delay Queue automaton returns to state S0 to wait for the new delay time to be reached, or for another task to delay.

5.2.3 Protected Objects

Calls to a Protected Object (PO) are handled by automata representing the different routines in a PO. A PO can consist of one Entry, any number of Procedures, and any number of Functions. The kernel model includes a PO with one Procedure (Figure 5) and one Entry (Figure 6). A system with multiple Protected Objects would need to be modeled with duplicates of these automata, with different variable names to distinguish between them (any variable in the pictured automata with a '1' in the name would need to be changed to have a '2' for the second PO, a '3' for the third, etc). There would be identical Entry automata for each Entry in the system, and identical Procedure automata for each Procedure in the system. If Functions are desired as well, either Function automata would need to be added, or the Procedure automata would need to be modified to account for the small differences between them.

The Protected Procedure call begins when the task and the Protected Procedure automaton synchronize on channel *Ps*. Section 3.2 contained a discussion of Protected Objects and how they affect task priorities. As mentioned in that section, the task calling a PO will inherit the ceiling priority of that PO, preventing any other tasks from interrupting it and accessing the same PO. Changing the calling task's priority is the first activity of the Procedure automaton. It stores the task's normal priority level in the variable *OldTaskPrioPO1*, and then changes the task's priority to the PO's ceiling priority, *Ppo1c*, by changing the priority value in the *RQ* array. After that, the Procedure waits to synchronize with the task again, either on *UPe* or *UPx*. *UPe* represents normal execution of the Procedure, and *UPx* (followed by *UPxe*) handles the case of an exception occurring. In either case, the Procedure must then check if a task is on the Protected Entry queue (*ECount* == 1) and if the Barrier variable is set to true (*Barrier* == True). If this is the case, the Entry code is executed within the context of the task that had been executing the Procedure code; this causes the task that had been suspended while calling the Entry to become active again. Finally, the task leaves the Procedure (on either *Pe* or *Px*), and the priority of the task is set back to its original value.

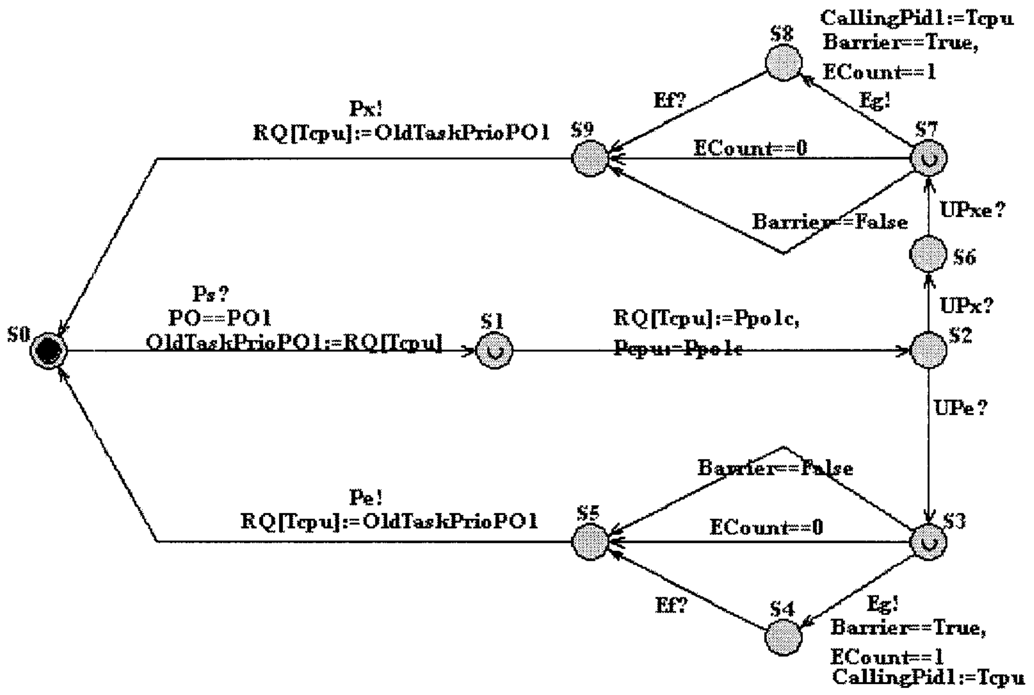

Figure 5. The Protected Procedure in UPPAAL

The Protected Entry call begins when the task and the Protected Entry automaton synchronize on channel *Es*. The Entry also changes the value of the task's priority in the *RQ* array to be the PO's ceiling priority, but it also immediately checks the *Barrier* value. If the *Barrier* has a value of true (*Barrier* == True), the Entry executes, with exceptions handled if they occur. If the *Barrier* has a value of false (*Barrier* == False), the task is placed on the Protected Entry queue (*ECount* := 1) and the task is suspended. The remainder of the Entry code can only be executed within the context of another task calling the Procedure as described above, with *Barrier* set to true. The task remains suspended at state S5 until the Entry completes in this manner.

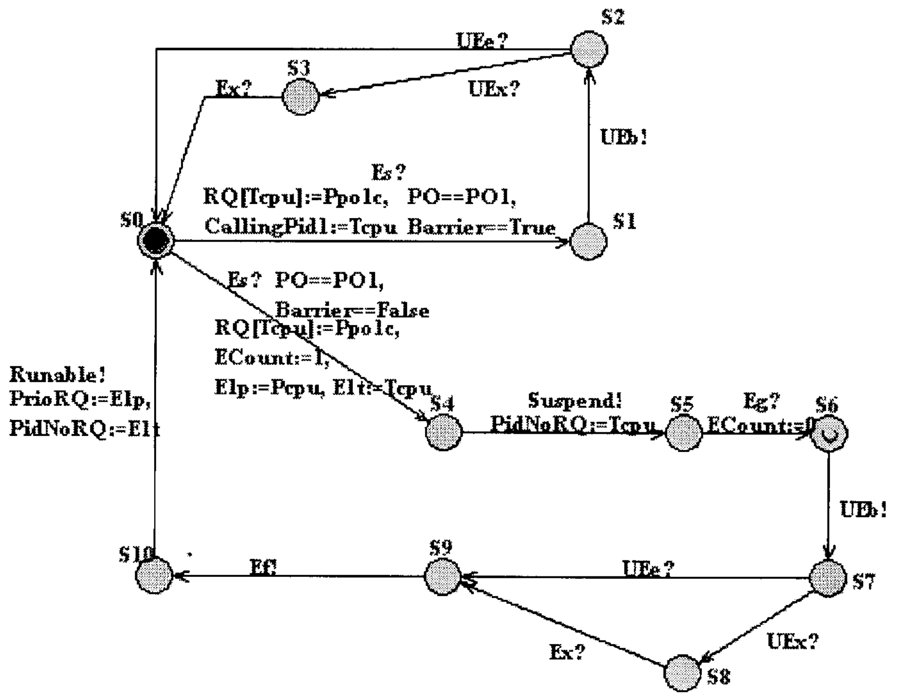

Figure 6. The Protected Entry in UPPAAL

5.2.4 *System Time*

Figure 7 shows the clock automaton. The variable *Time* increments by 1 every time unit. *Time* is used for *delay until* calls, keeping track of execution time of tasks, and for system verification. The channel *Go* is used to force transitions on other automata that would not otherwise be taken.

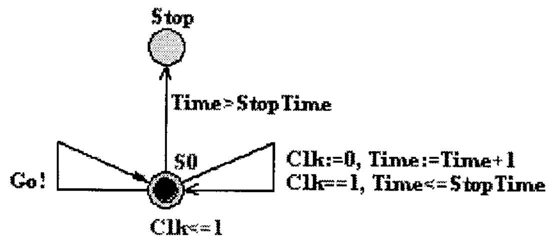

Figure 7. System Time in UPPAAL

5.2.5 Idle Task

An Idle Task, show in Figure 8, is also part of the model. It has the lowest priority of any task in the system. It runs at startup, or when there is no other task available to run. Having an idle task ensures that there is always a task running.

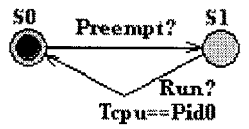

Figure 8. Idle Task in UPPAAL

5.2.6 Example Tasks and Verification

Three task automata are part of the model. Each of these automata represents an Ada task. An Ada program contains a section of sequential code (with no kernel calls), and this code is annotated with a Best Case Execution Time (BCET) and a Worst Case Execution Time (WCET). When modeling a task, all of the sequential code is condensed into a single preemtable state, and the execution time is included in the model. Other sections of an Ada program contain all calls to the RTK, such as *delay until* and PO calls, and these calls are included in the automaton model of a task. They become channels that synchronize with the different parts of the kernel model described above. An example of a task that repeatedly makes *delay until* and protected procedure calls is shown in Figure 9.

Verification of the model was performed using UPPAAL. Properties to be verified are specified using a formal syntax. There are four main properties of the model that must always be true for a single processor system:

1. Only one task may execute the body of a given PO at a given time.
2. A task on the Entry queue will always execute if *Barrier* == True when a Protected Procedure has finished executing.
3. When a task's delay is up, *Time* must be larger than or equal to what the tasks delay time was.
4. The highest priority task available to run will synchronize with the *Run* channel in the Ready Queue automaton.

These constraints were turned into queries for UPPAAL. UPPAAL then checked for safety and liveness of the system, and verified that all properties were satisfied. In addition, all possible system states were checked, and a total of 9000 verifications were done, with the model running until *Time* equaled 120.

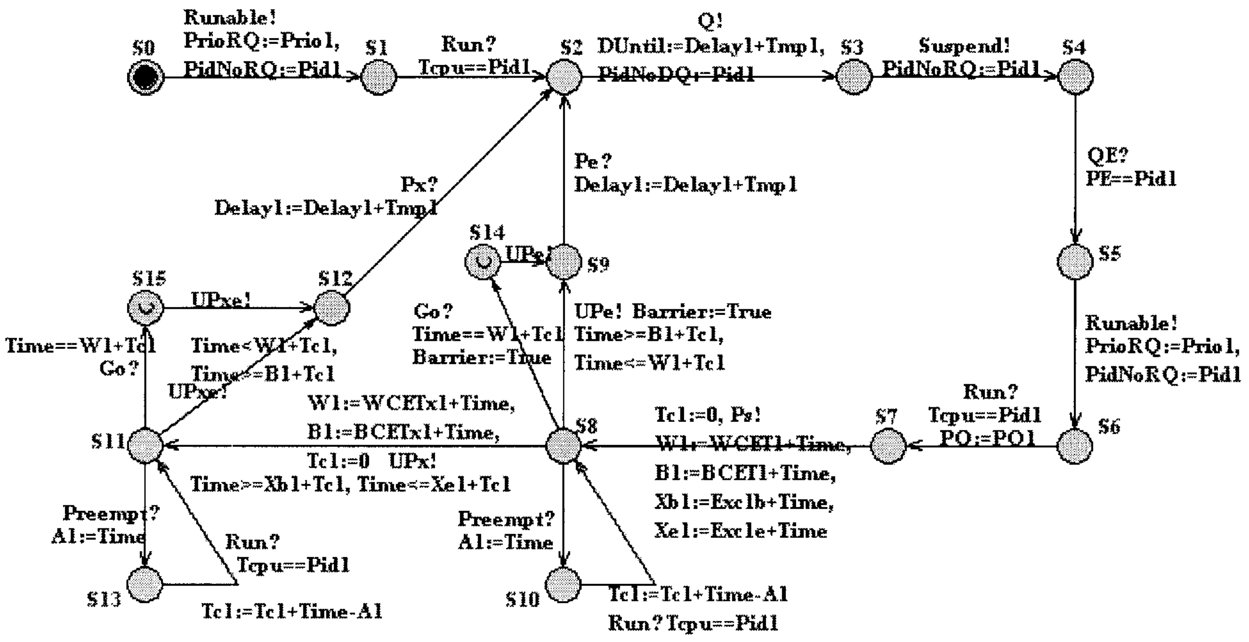

Figure 9. Task in UPPAAL

5.3 Issues in Refining the Model to VHDL

Analysis of the kernel model in UPPAAL provides a thorough description of what both the behavioral and timing properties of the RavenHaRT Kernel should be. To implement the RavenHaRT Kernel, the model had to be refined into VHDL code. The VHDL kernel should obey the same properties that have been formally verified in the model. However, the UPPAAL model is meant to be an abstraction of the real design. Certain features of UPPAAL were used in the model, but the way in which they were used create behavior that cannot be implemented in VHDL. These features include channels and urgent states, and also the method used for how tasks are allowed to interact with the kernel automata. This section discusses the different issues that arose when determining how best to refine the model into VHDL, and describes how the issues were handled.

5.3.1 Timing

A model in UPPAAL is an abstraction of the complexity of the actual implementation. One of the most critical mappings is the modeling of time. UPPAAL uses a unitless clock, which for the purposes of this discussion will be referred to as *UClk* (it was the variable *Time* in Figure 7). *UClk* counts up from zero. It is an abstraction of the Ada clock, which keeps track of real time and increments at a user specified resolution. The Ada clock and *UClk* are used for *delay until* calls. In Ada code, a task would *delay until* a particular point in time. In UPPAAL, a task would *delay until* a *UClk* number value, such as, *delay until UClk* equals 10. UPPAAL also uses *UClk* in analysis. As mentioned earlier in the chapter, assertions can be checked at certain points in time. Verification of the model included checking that the system state is valid at certain values of *UClk*.

In addition, automata transitions in UPPAAL occur in zero time. They do not need to transition with respect to any clock. An infinite number of transitions can occur between each increment of *UClk*, and *UClk* does not advance regularly with respect to real time. For example, in the model, a task may be created and add itself to the *RQ* array; a second task may add itself; the task with the highest priority will be found, begin to run, and possibly make other calls, all before *UClk* advances. Urgent states even force multiple transitions to occur before *UClk* changes. Only one transition might occur between changes in *UClk*, or 100 might; a transition does not take a finite amount of time with respect to *UClk*. This behavior of the model is consistent with respect to the critical timing behavior that the model captures, but it cannot be implemented.

The RavenHaRT Kernel is a collection of synchronous state machines, and since it is implemented in VHDL, it has a different type of clock. The main clock is a clock that is on the FPGA (referred to as *FClk*). *FClk* controls when the logic on the FPGA is evaluated; each state machine is evaluated simultaneously and may transition at most once each clock cycle. A clock can be created in VHDL that captures the behavior of *UClk*. It is essentially a counter, called

VCounter, that keeps track of time at the same resolution as specified in Ada, and it is used to check time for *delay until* calls.

Analysis of the RavenHaRT Kernel must consider both when *VCounter* changes and when *FClk* changes. Various logical equations are calculated in between each rising edge of *FClk*. It is necessary to determine that all the needed logic can physically be evaluated on the FPGA, and that it is evaluated accurately, during each cycle of *FClk*. Xilinx tools such as ISE Foundation are used to analyze the VHDL code. The tools determine what speed *FClk* can run at to successfully evaluate the logic.

Unlike *UClk, VCounter* does increment regularly with respect to real time, so that while it serves the same purpose as *UClk*, the system state at certain values of *VCounter* may not correspond to the model's system state at the same values of *UClk*. Each state machine transition takes a finite amount of time. Because of this, certain timing behaviors in the model cannot be followed in the implementation. One of these behaviors is the use of urgent states; the way in which they were used in the model cannot be implemented VHDL. The model of the Delay Queue was described in section 5.3.2 and is shown in Figure 4. When the automaton performs a search to find the task with the smallest delay time, it loops through many states. As it does this, variables are assigned, evaluated, and reassigned. In VHDL, a signal can be assigned a value during one *FClk* clock cycle, but then that signal cannot be used until the next clock cycle. All of the states (except S0) in the Delay Queue model are urgent, so in the model the search always completes before *UClk* advances. Since no time passes, tasks are able to wake up from a delay exactly when they are supposed to. However, in the VHDL implementation, *FClk* must advance at each state machine transition so that the signals are used and evaluated appropriately. *VCounter* is advancing as well, at a rate equal to or slower than *FClk*. Since the implementation of *VCounter* is dependent on *FClk*, it may (or may not) advance while the Delay Queue is searching for a task. The result is that a task may awaken after its specified time. Fortunately, in this case the difference in timing between the model and the actual implementation is not a problem, because the *delay until* command dictates that a task will delay *at least* until the specified time. Waking after that time is acceptable [BW98]. There may be other cases similar to this, where there are discrepancies between *UClk* and *VCounter*, and they must be evaluated thoroughly to ensure that the timing behavior of the implementation is still acceptable.

5.3.2 State Machine Synchronization

UPPAAL uses channels and shared variables to model communication between automata. These communication models need to be translated into actual VHDL constructs. Channels do not exist in VHDL, but can be created by setting a signal to only be true when the two state machines can transition, and then using that signal as a guard in each state machine. An example UPPAAL automaton was shown in section 5.1, in Figure 2. In that example, a channel *c* and the guards *x* and *y* control the transitions from A0 to A1 and B0 to B1. The corresponding VHDL code for this transition is shown in Figure 10.

```
c <= '1' when(stateA=A0 and stateB=B0 and x>=2 and y>=4) else '0';
......                       // channel c is now a guard
if(clk'event and clk=1) then
  case stateA;               // state machine A
    when A0 =>               // when in state A0
      if (c=1) then          // if guard c is true (c==1)
        stateA <= A1;        // can transition to A0
        x <= 0;
      else                   // if guard c is not true (c==0)
        stateA <= A0;        // must stay in A0
      end if;
  ......
  end case;
  case  stateB;              // state machine B
    when B0 =>               // when in state B0
      if (c=1) then          // if guard c is true (c==1)
        stateB <= B1;        // can transition to B0
        i <= 1;
      else                   // if guard c is not true (c==0)
        stateB <= B0;        // must stay in B0
      end if;
  ......
  end case;
end if;
```

Figure 10. VHDL Code for part of Figure 2

Channel *c* is now a guard that equals 1 only when control of state machine A is in state A0, control of state machine B is in state B0, *x* is greater than or equal to 2, and *y* is greater than or equal to 4. Otherwise, *c* equals 0. The guard *c* is set asynchronously and will therefore change whenever its conditions change, but the state machines can only transition on the rising edge of the clock. Both state machines, when in states A0 and B0, check to see that *c* is equal to 1 before they transition.

5.3.3 Interface With Tasks

As described in section 5.2, tasks are modeled so that a complete system model can be analyzed. The tasks are modeled as if they behave in the same way as the kernel. The tasks and kernel can synchronize on channels, and they can access shared variables. This system makes analysis of the model easier, but it cannot be implemented. Tasks will be Ravenscar-compliant Ada code running on the PPC, and the kernel will be located on an FPGA. Figure 11 shows how the various components are connected. After compilation, the code communicates with the RavenHaRT Kernel over a bus by sending instructions and parameters. Instructions are any system calls that the kernel handles, such as task creation, or *delay until.* The RavenHaRT Kernel has a specialized instruction set, so the Ada compiler needs to be modified in order to produce the right instructions in the right format. Parameters are defined as any variables that both the tasks and the RavenHaRT Kernel use, such as a task's priority and ID number; this data must be passed to the kernel over the bus. The bus protocol, instruction set, required parameters, and methods for creating the instructions will be discussed in the next chapter.

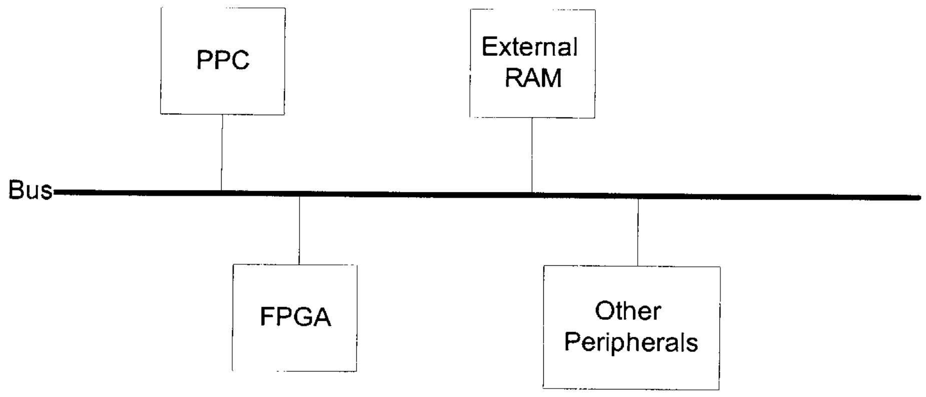

Figure 11. Components Connected by a Bus

Chapter 6
The RavenHaRT Kernel

This chapter presents a detailed description of the code for the kernel in VHDL, including the interface and instructions used for communicating with the PPC. Examples of pseudo-code are given to show what the corresponding software behavior should be for given situations.

The RavenHaRT Kernel is designed to support code written using the Ravenscar profile of Ada, with an appropriately modified Ada compiler, in a single processor system. As described in chapter 3, intertask communication is done via Protected Objects (POs) when using the Ravenscar profile. Tasks are allowed to make calls to POs and to use the *delay until* statement, and the kernel supports these actions. It determines what task should be running using priority based preemptive scheduling. To do scheduling correctly, the kernel keeps track of delays that have been made and all calls made to POs. PO calls can be calls to the Protected Entry, Protected Procedure, and Protected Function.

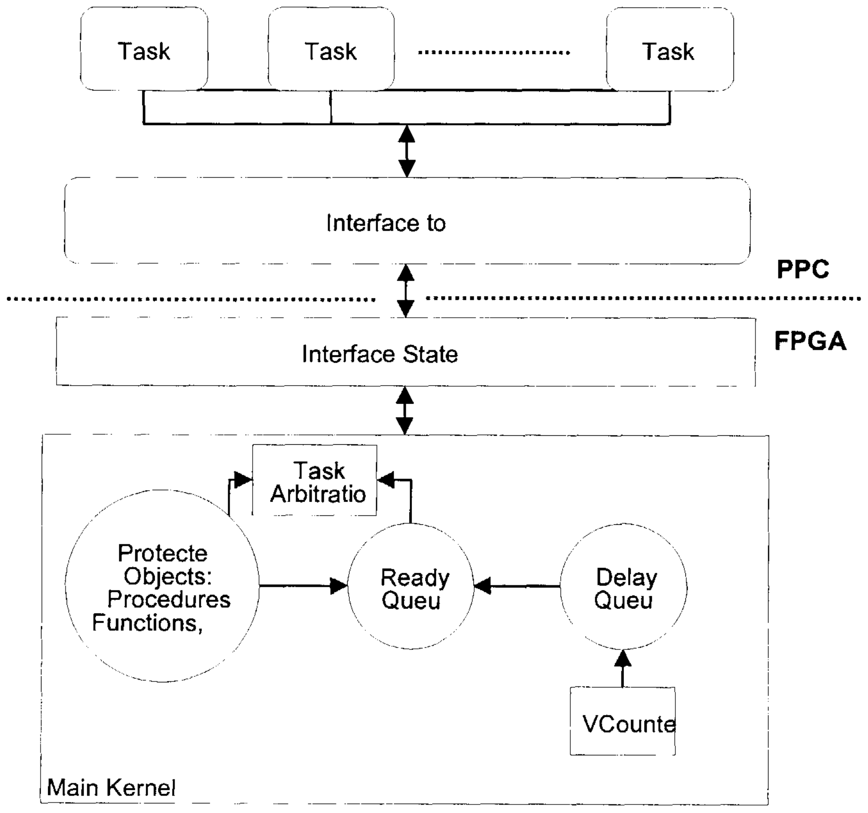

Figure 12. Architecture of Kernel and Application

Figure 12 shows the architecture of the kernel, and how it relates to the application. The application level involves everything above the dividing line between PPC and kernel. At the top are the software tasks, which are compiled code that has been loaded onto the PPC. Below that is the interface to the kernel, where instructions which are based on the compiled code are sent to the kernel, and where data from the kernel (such as what task to run) is read. This block may eventually consist of the modified Ada compiler, but is still under development.

The different components inside the FPGA that comprise the kernel are listed in Table 1. The first column shows the name of the kernel component that corresponds to the different FPGA blocks in Figure 12. The second column indicates the name of the piece of code the routines are located in, as well as where the code can be found in the Appendix. The third column briefly summarizes the behavior of that component.

There is one more important VHDL file that is used by all of the kernel components. It is a package file called *myvariables.vhd* (Appendix A.8), and in it all constants are defined, as well as the state types and state names for most of the state machines. Here, values such as the number of tasks or Protected Objects in the system can be changed. Also located in this file are all the bit sequences that are the instructions the kernel expects to receive for each command. Values in this file can be changed, if, for example, a kernel that can handle more tasks is needed. However, the FPGA would then need to be reprogrammed.

The following sections of this chapter describe each piece of the VHDL code, including how it was designed, how it functions and how it should be used. Most kernel tests were performed on a kernel with four tasks and four POs, so those values are used throughout this chapter. A diagram of each state machine in the kernel is shown. Note that in each diagram, both conditions and signal assignments are shown on many state transitions. Any assignments occur only after all transition conditions have been met.

As mentioned in section 5.3.3, a bus is used for communication between the kernel and the compiled tasks running on the PPC. The compiler will be altered in a way such that the compiled code creates the correct instructions for the kernel, and so that the PPC appropriately responds to data provided by the kernel. For simplicity, the 'compiled task code running on the PPC' will in this chapter be referred to just as 'tasks'.

Component	*Code*	*Behavior*
Interface State Machine	interface_ppc.vhd (Appendix A.1)	Handles PPC instructions, writes to data registers for PPC to read, generates PPC interrupts
Main Kernel Functions	kernel_internal.vhd (Appendix A.2)	Connects internal kernel components together
Ready Queue	rq_state_and_ram_and_arbit.vhd rq_ram.vhd rq_state.vhd arbitrate_rq_ram.vhd (Appendix A.3)	Ready Queue routines; creation of tasks, scheduling, maintenance of task priorities and their status
Delay Queue	dq_state_and_ram.vhd dq_ram.vhd dq_state.vhd (Appendix A.4)	Delay Queue routines; tracks task delays
VCounter	timer.vhd (Appendix A.5)	Manages signal *VCounter* (used by Delay Queue)
Protected Objects	multipo.vhd po_one.vhd channel_controller_po1.vhd entry.vhd procfunc.vhd (Appendix A.6)	All Protected Object routines
Task Arbitration	arbitrate_cpu.vhd (Appendix A.7)	Works with Ready Queue and POs to help determine what task should be running

Table 1. Kernel Components

Examples are given in each section below for what code would need to be running on the PPC to send and receive information to the kernel. These examples are intended to be pseudo-code. They include information on what commands and parameters to send in certain situations.

They also demonstrate the handshaking between the kernel and the PPC, since after commands are sent the task running must usually wait for a status signal from the kernel before it can proceed. Note that the examples assume that the correct, expected status will be sent, and do not include example code for handling a status value that shows otherwise (although this would be needed in the actual implementation). Also, the examples do not show how the task would recover if it receives an interrupt after sending a command and does not receive a status message indicating that command has been finished. Whenever the task resumes, it would need to resend the command, so additional loops would be needed in the compiled task code so that action occurs. This handshaking behavior will be explained in more detail in section 6.1.

6.1 Interface

The interface code really consists of two different parts. The first handles reading from and writing to registers for communication over the bus (called BISM, for Bus Interface State Machine). The second handles the high level acknowledgement and processing of instructions and data read from these registers, and the generation of any interrupts to the PPC or other data that will need to be placed in registers for the tasks to read (called KISM, for Kernel Interface State Machine).

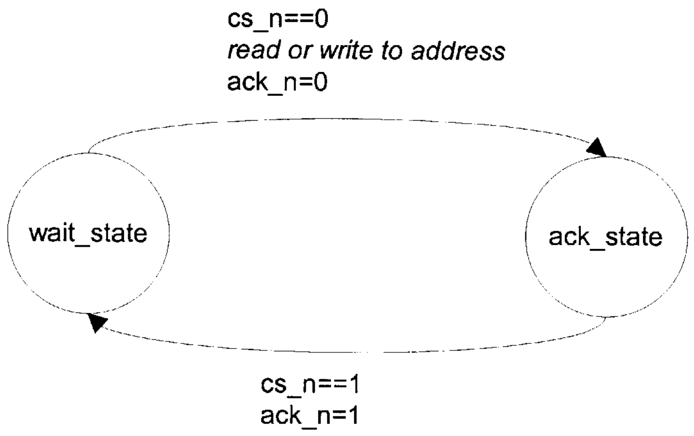

Figure 13. Bus Interface State Machine

The bus has a 32-bit input register (*din*) and a 32-bit output register (*dout*). The BISM, shown in Figure 13, has only two states, *wait_state* and *ack_state*. When in *wait_state*, the

BISM waits for the running task to provide an address, and the task either sets the *write_n* signal low if it wants to write to the bus or high if it wants to read from it. The address indicates which hardware register it will read from or write to. In order for this read or write to occur, the chip select signal *cs_n*, set by the task, must be low. Figure 13 does not fully show all of the actions taken for the different possible reads and writes. For a read, the appropriate data must be placed in the specified register for the PPC to read. For a write, the kernel must read the data the PPC has placed in the specified register, and follow through with any commands that data implies. After a read or write is performed, the output *ack_n* is set low by the BISM and the BISM transitions to *ack_state*. Setting *ack_n* low provides an acknowledge signal to the task indicating that the read or write has occurred. This acknowledge must be sent for both a write and read since the PPC is the bus master, and must know that either the data it wanted to write has actually be received by the kernel or that the data it wanted to read is available.

There are currently eight registers, and they are shown in Figures 14-21 with the corresponding addresses the VHDL code uses to uniquely identify the registers. The address values correspond to the actual location in memory the registers are located. All registers have only one possible configuration, except for the Lower Parameter/Barrier Register, which will contain different information in the different cases shown. In each figure, the numbers represent the number of register bits needed for representing that register's information. The figures assume a system with four tasks, four POs, and eight priority levels. The least significant bits of the register should be used when not all 32 bits are.

27	5
Don't Care	Command

Figure 14. Command Register (Input), Address 00000100

31	1
Don't Care	Ack

Figure 15. Interrupt Acknowledge Register (Input), Address 00010000

32
Higher Bits of Delay Until Time

Figure 16. Higher Parameter Register (Input), Address 00001100

Following a Create Command:

27	2	3
Don't Care	TaskID	Task Priority

Following a SetFreq Command:

24	8
Don't Care	VCounter Frequency

Following a PO Command:

27	3	2
Don't Care	CeilingPrio	PO ID

Following a DelayUntil Command or a Barrier Request:

32
Lower Bits of Delay Until Time / Barrier Values

Figure 17. Lower Parameter/Barrier Register (Input), Address 00001000

29	3
Don't Care	Status

Figure 18. Status Register (Output), Address 00010100

30	2
Don't Care	Task ID

Figure 19. New Task ID Register (Output), Address 00011000

32
Lower Bits of Kernel Time (VCounter)

Figure 20. Lower Time Register (Output), Address 00011100

32
Higher Bits of Kernel Time (VCounter)

Figure 21. Higher Time Register (Output), Address 00100000

The KISM acts as an interface between the bus registers and the main functions of the kernel itself. This state machine is shown in Figure 22. It has 11 states, *i0* through *i10*. It waits in state *i0* until a task sends a new command to the Command Register. All of the valid commands are listed in Table 2 with the expected bit sequence that defines that instruction. The KISM knows when a command is received because the BISM has set the signal *new_cmd* to 1. The KISM then knows that there is valid command data held in *cmd* and so it enters state *i3*. In state *i3*, the KISM waits for the task to send data to the Parameter Register if there is data associated with the received command. Most commands only have enough associated data to use the Lower Parameter Register; the exceptions are *GetTime* and *FindTask*, which have no associated data, and *DelayUntil*, which must use both the Lower and Higher Parameter Registers. The KISM is aware of these circumstances. If the values in *cmd* indicate either the command *GetTime* or *FindTask*, it proceeds directly to state *i4*. Otherwise, it waits in state *i3* until valid data is placed in the Lower Parameter Register. It knows this has occurred when the BISM sets the signal *new_param_low* high. If the values in *cmd* do not indicate the command *DelayUntil*, the KISM goes go state *i4*; otherwise, it goes to state *i5* to wait for data to arrive in the Higher Parameter Register, which it knows has occurred when BISM sets the signal *new_param_high* to 1. Then it finally proceeds to state *i4*.

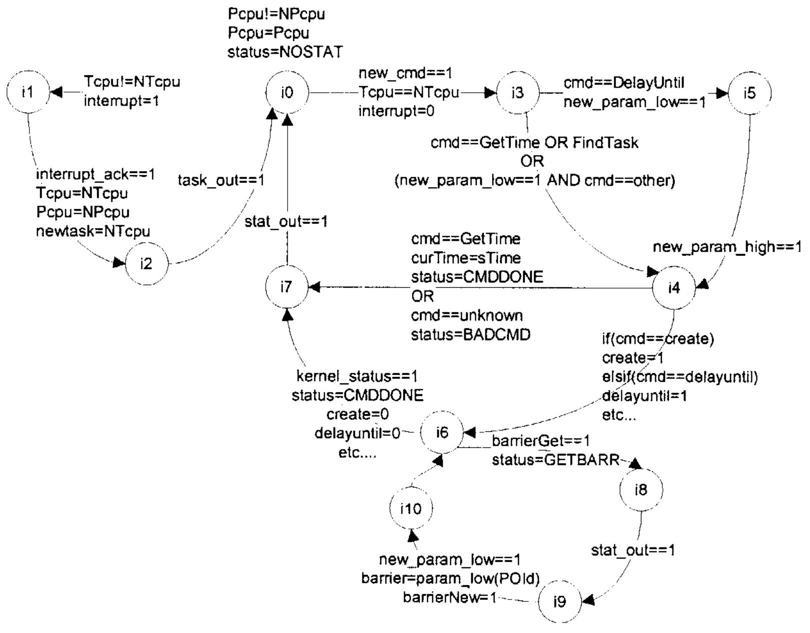

Figure 22. Kernel Interface State Machine

After a task sends a command (and possibly a parameter), it must wait for the kernel to respond by placing a message in the Status Register. Messages that can be put in the Status Register are listed in Table 3. This way, the task knows if the command was carried out, or if something else occurred. The KISM places the value in the Status Register, and then the BISM signals back to the KISM by setting *stat_out* to 1 when the task has completed the read of the Status Register. Figure 23 shows the handshaking that occurs between the PPC and the kernel for a typical command sequence. It shows how each PPC read or write must be acknowledged by the BISM, and how the PPC must receive a valid status after each command sequence before continuing.

Command	Instruction Bits
Create	10000
DelayUntil	00001
GetTime	01101
SetFreq	01110
FindTask	01111
FPs	00010
UPe	00011
UFe	10001
FPe	00100
UFPx	00101
UPxe	00110
UFxe	10010
FPx	00111
Es	01000
UEb	01001
UEe	01010
UEx	01011
Ex	01100

Table 2. Commands and Instruction Bits

Status	Register Bits
No Status	000
Bad Command	001
Command Done	010
Barrier Request	100

Table 3. Status Signals

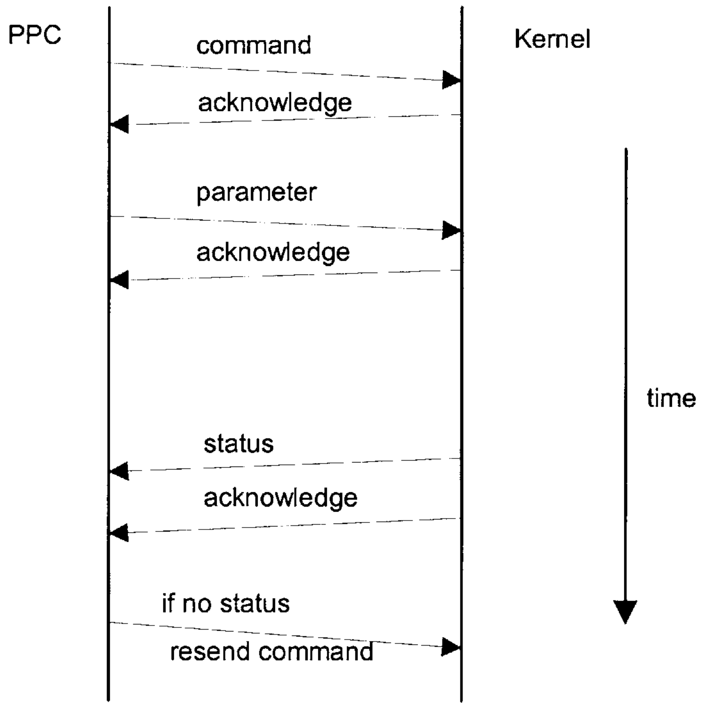

Figure 23. Handshaking Between PPC and Kernel

In state *i4* the kernel begins to carry out the received command. If the command is not valid, it sends the message 'Bad Command' to the Status Register. If the command is a *GetTime* command, the current time if the kernel counter *VCounter* is written to the Higher and Lower Time Registers, the status is set to 'Command Done', and control returns back to *i0*. For all other commands, the appropriate signals are sent to other parts of the kernel, and the KISM transitions to state *i6*. It waits here until one of two actions occurs. A PO may make a Barrier request, which will be described in more detail in the PO section below. When this happens, the status is set to 'Barrier Request', and the KISM waits for the task to write the Barrier values to the Lower Parameter Register. Since the Lower Parameter Register is being used, two important items must be noted: values must be written for all 32 bits, even those that do not correspond to a valid PO (zeros should be used in this case), and no more than 32 POs can be used in the system without changing this code. When the Barrier values have been received, the KISM returns to *i6*. The only other way to leave state *i6* is when the main section of the kernel sends a signal indicating the command has been completed. Then the KISM writes 'Command Done' to the Status Register, goes to state *i7* to wait until it knows the task has read the status, and then returns to *i0*.

The kernel is able to interrupt the PPC for a task-switch using a dedicated interrupt signal. The KISM generates this interrupt signal. It keeps track of what task is currently running, and what its priority is. It is alerted when the kernel changes the priority of a task, or finds a task to run of higher priority. These changes are only handled in state *i0*. This ensures that once a command is sent to the main kernel routines, it will not be interrupted. The task that sent the command will also always have the opportunity to read the status when a command has been completed before it receives an interrupt signal; this is done by checking that *stat_out* is set to 1 before transitioning to *i0* as described above. However, since new commands are also looked for in *i0*, it may happen that the task sends a command at the same time an interrupt occurs. The task must assume that the command it sent has not been done unless it receives the 'Command Done' status. So, if the task sends a command and the PPC receives an interrupt before the status is set to 'Command Done', the task must resend that command whenever it becomes active again.

The priority of a task may change because the priority was raised to the ceiling priority of a PO it calls. This does not create an interrupt because the task does not change. When the task itself changes, which may occur because a task wakes from a delay and has higher priority than the one running, an interrupt is sent to the PPC and the KISM transfers to state *i1*. The PPC must first send an interrupt acknowledge signal to the Interrupt Acknowledge Register. When this happens, the ID of the new task is put in the New Task ID Register and the KISM transfers to state *i2*. Once the New Task ID Register has been read, the KISM returns to *i0* to await the next command or interrupt.

6.2 The Main Kernel

The rest of the kernel code is considered to be the 'main' kernel. As mentioned above, the different parts of the main kernel are connected in the code *kernel_internal.vhd,* and then the main kernel is connected to the interface code. In addition, several signals are arbitrated here. The kernel *status* signal is set high if any of the lower level routines have set it. This signal indicates that any command made has been completed. Also, all of the signals from the multiple POs in the system are handled internally as an array of data. When a PO is accessed, the ID of the PO (*POId*) to be accessed enters the kernel as parameter data. The KISM passes *POId* down into this top-level main kernel code, so the correct PO signals are selected and accessed. Also, two channel signals are defined here. Channels are implemented in the kernel as described in section 5.3.2. The channels defined here, *QE* and *SuspendTask,* have dependencies on state machines in different pieces of lower level code, so they must be determined at this level. The uses of these channels will be described in later sections.

6.3 The Ready Queue

The Ready Queue keeps track of all tasks. It determines what tasks are capable of running, and makes sure the highest priority task is sent to the interface to request to be run. The Ready Queue state machine (RQSM) manages all of the tasks and acts as a scheduler. It has three main branches: one to create tasks and store their data, one to search for a task to run, and one to acknowledge and revalidate newly awakened tasks. The RQSM is shown in Figure 24.

When the system first starts up, an elaboration phase occurs. The elaboration phase occurs before any 'real' tasks start running; code is run on the PPC that tells the kernel what tasks exist in the system. The kernel must know each task's ID number and its priority. The first task should be task number one, the second task number two, etc. Task zero is reserved for the null task. Each task's information is sent individually to the kernel through the use of the *Create* command. First the instruction for *Create* is sent, and then the ID and priority information are put in the Lower Parameter Register.

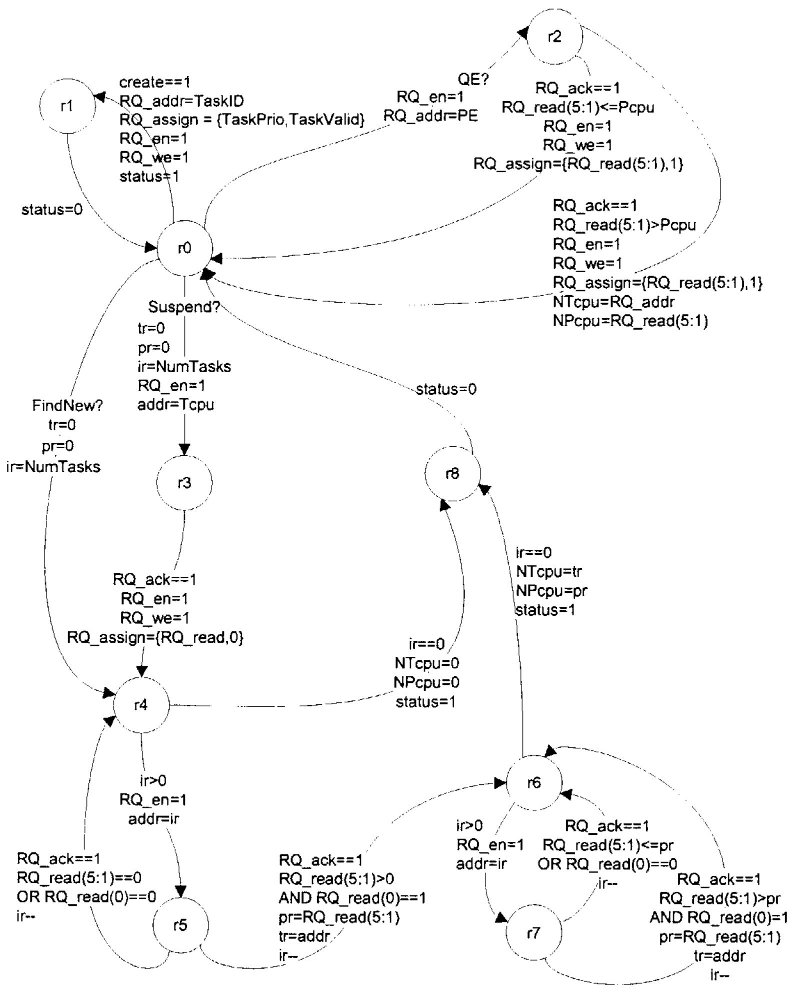

Figure 24. Ready Queue State Machine

For a system that allows four tasks (including the null task) and up to eight priority levels (including zero, which is also reserved for the null task), the Lower Parameter Register holds the task's priority in the lowest three bits and the task's ID number in the fourth and fifth bits (as shown in Figure 17). To create task of ID number two that has a priority of four, the following code sequence would be used:

```
Write Create, Command Register
Write 0x00000014, Lower Parameter Register
Read status, Status Register
While status == 'No Status'
        Read status, Status Register
```

During elaboration, the RQSM waits at state *r0* until it receives the *Create* command (*create* == 1). Then, it transfers to state *r1* and writes the task data that is in the Lower Parameter Register into the Ready Queue RAM (RQ_RAM). Simulating RAM in VHDL provides a good method of storing data while minimizing the amount of space needed on the FPGA. To access a task's data in the RQ_RAM, it is addressed with its ID number. The total number of tasks in the system, including the null task, should always be a power of two. Following this rule will ensure that the number of bits needed to define all ID numbers actually corresponds to valid tasks, and valid RQ_RAM locations. The RQ_RAM has one enable signal and one write enable signal, both of which are enabled when high. For each task, the RQ_RAM stores whether or not it is valid to run in the lowest bit; the remaining bits store the task's priority. The RQ_RAM is first populated during the elaboration phase, but it can also be altered during normal kernel operation. For example, when a task becomes suspended, its valid bit is changed from 1 to 0. Also, when a task has its priority raised to the ceiling priority of the PO it is calling, the priority for that task is changed in the RQ_RAM.

The clock counter *VCounter* is also set during elaboration, and that will be discussed in section 6.7. The last action of the elaboration phase is for the elaboration code to tell the kernel to find the first task to run using the *FindTask* command. Whenever the kernel finds a new task to run, an interrupt is made, even if no task is currently running. The *FindTask* command has no parameter, and the code for using it, including handling the interrupt, is as follows:

```
Write FindTask, Command Register
Read status, Status Register
While status == 'No Status'
        Read status, Status Register
--- Interrupt to PPC here ---
Write 0x00000001, Interrupt Acknowledge Register
Read NextTask, New Task ID Register
```

When the *FindTask* command is made, the channel *FindNew* in the RQSM is activated. This causes the RQSM to transition from state *r1* to state *r4,* and then it proceeds to search the RQ_RAM for the highest priority task. To do this, it reads each RQ_RAM entry one at a time, checking for the priority and ensuring that the task is valid. It holds the priority and task values in the temporary variables *tr* and *pr*, and when the search finds a task with a higher priority, the values in the variables are changed. When the entire RQ_RAM has been searched, *NTcpu* is set to *tr* and *NPcpu* to *pr*. *NTcpu* represents the next task ID to run on the PPC, and *NPcpu* represents the priority of that next task. This data is passed up to the interface code (by way of the task arbitration code, which will be discussed in section 6.6) so that an interrupt can be performed.

The section of the RQSM that finds new tasks can be entered at other times besides elaboration. Whenever a PO exits, it lowers the task's priority back to its original priority, and synchronizes with the RQSM on the channel *FindNew*. The RQSM does a search as described above, because a task may be valid to run that had lower priority than the PO's ceiling priority but higher than that of the current task.

This branch is also used whenever a task is suspended. A task may be suspended because it made a *delay until* call, or because it made a Protected Entry call with the *Barrier* equal to false (see 6.5 for more information). In either case, the RQSM synchronizes on the channel *Suspend* and first goes to state *r3* before continuing to *r4*, because it must set the valid bit in the RQ_RAM for the suspended task to 0. Then it searches for the task to run next.

The final branch synchronizes on the channel *QE* with the Delay Queue whenever a task wakes up from a *delay until* call. The RQSM transitions to *r2* and sets the valid bit for that task back to 1. If the newly valid task has a higher priority than the task currently running, *NTcpu* and *NPcpu* are set accordingly so an interrupt can be made. If this is not the case, then nothing else occurs.

The RQ_RAM can only be read from or written to by one state machine at a time. However, there are multiple state machines that are capable of accessing it. The RQSM both reads from and writes to it, and POs are capable of writing to it to change a task's priority. Problems may arise when two state machines attempt to access the RQ_RAM at the exact same time, so arbitration code is needed.

The only times that conflicts can possibly occur is when a task wakes up from a delay (causing the *QE* branch of the RQSM to be taken) at the same time a PO changes a task's priority. All other RQ_RAM accesses in the RQSM occur when no other processes are actively transitioning. Since only one PO can be active at a time, and also only one routine in that PO can be actively transitioning, the number of POs is not a concern; the arbitration routine only cares that any given PO is accessing the RQ_RAM. So no more than two accesses, one by a PO and one by the RQSM, are ever attempted at the same time. In this dual access case, all that needs to be determined is which access should occur first. The one that occurs second will never conflict with an additional access because of the nature of the interacting state machines, so the dual access case is the only one that needs to be accounted for.

Two interacting state machines are used for the arbitration, one to handle the different dual access cases and one to handle reads to the RQ_RAM. Whenever a read is done, an acknowledge signal is sent back to the process requesting the read, to notify it when the read is complete and valid data is available. Whenever a PO tries to write at the same time the RQSM tries to do any access, the PO always writes first. Both requests are stored in the arbitration code, and both will be performed before any interrupts can occur. This method ensures that the PO accurately completes its current command (like changing the running task's priority) before a newly awakened task can be allowed to even check to see if it can interrupt. This method provides a simple and safe way to control access to the RQ_RAM.

6.4 The Delay Queue

The Delay Queue keeps track of what tasks have made a *delay until* call, and checks to see when they should become active again. A task that calls *delay until* in Ada must make two commands to the kernel. First it must send *GetTime*, to get the current time of the counter *VCounter*. After reading the status to know that the *GetTime* command has been performed, the task must read from the two time registers, Lower Time and Higher Time, which now hold the value of *VCounter*. Then once the task calculates what the delay time is relative to *VCounter*, it sends the *DelayUntil* command. Both Parameter Registers are used to send the *delay until* time, to ensure that the full delay time is accurately given to the kernel. The lower delay time bits must always be sent first, in the Lower Parameter Register. The code would be as follows. Since a *DelayUntil* always results in an interrupt being performed so a new task can be run, that is included.

```
Write GetTime, Command Register
Read status, Status Register
While status == 'No Status'
        Read status, Status Register
Read VCounter[31:0], Lower Time Register
Read VCounter[63:32], Higher Time Register
```

```
Write DelayUntil, Command Register
Write DelayTime[31:0], Lower Parameter Register
Write DelayTime[63:32], Higher Parameter Register
Read status, Status Register
While status == 'No Status'
        Read status, Status Register
--- Interrupt to PPC here ---
Write 0x00000001, Interrupt Acknowledge Register
Read NextTask, New Task ID Register
```

Delay data is stored in RAM simulated in VHDL (DQ_RAM), and the design is similar to the RQ_RAM. The DQ_RAM has an entry for each task, and the data is accessed by task ID number in the same way as the RQ_RAM. The DQ_RAM holds zeros for each task unless that task has made a *delay until* call, in which case it holds the time (relative to *VCounter*) at which the delay is up. Only the Delay Queue routines can access the DQ_RAM, so no arbitration code is needed like it was with the RQ_RAM. The DQ_RAM still does send an acknowledge signal whenever a read is performed, to let the Delay Queue know when valid data has been returned.

There are actually two state machines that interact to handle delays. Two are used so that certain actions can be performed in parallel. The first state machine, the Delay Queue Acknowledge state machine (DQASM) responds directly to commands from the interface, triggers the RQSM to find a new task, and triggers the second state machine, the Delay Queue Organize state machine (DQOSM). The DQOSM keeps track of what tasks have been delayed, and checks for when a task's delay time is up.

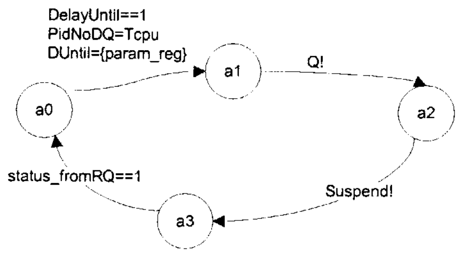

Figure 25. Delay Queue Acknowledge State Machine

The DQASM, seen in Figure 25, only has four states. When it receives the *DelayUntil* command from the kernel interface, it saves the ID of the current task and gets that task's delay time from the Parameter Registers. Then it synchronizes with the DQOSM on channel *Q*. This causes the DQOSM to record the delay information. Next, the DQASM synchronizes with the RQSM on the channel *Suspend*, so that the task can be made invalid and the next task to run can be found. When the RQSM has found the next task, the DQASM returns back to state *a0* to await the next *DelayUntil* command.

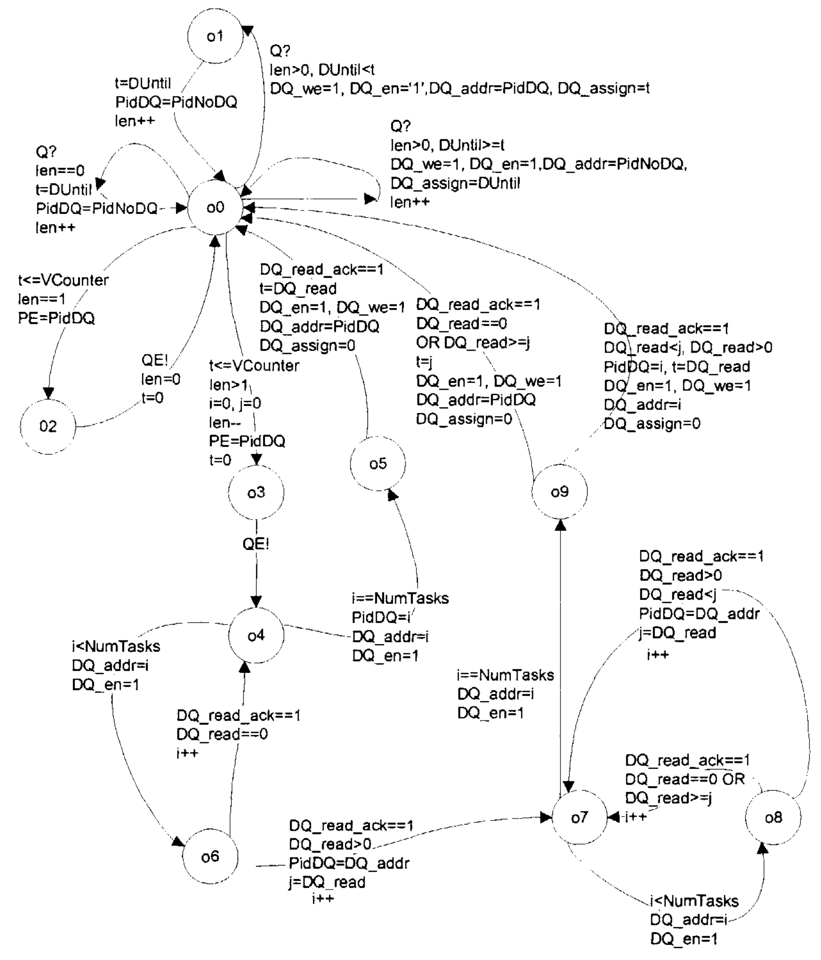

Figure 26. Delay Queue Organize State Machine

The DQOSM, in Figure 26, both keeps track of delay information with help from the DQ_RAM, and it checks to see when a task's delay is up. It keeps track of how many tasks are currently delayed with the variable *len*. When it synchronizes with the DQASM on *Q*, there are three cases that cause different behavior by the state machine.

1. No tasks currently delayed (*len* == 0): DQOSM saves task's ID in the variable *PidDQ*, its delay time in the variable *t*.

2. At least one task is currently delayed (*len*>0), new task being delayed has a delay time greater than (or equal to) *t*: new task being delayed and its time are entered into the DQ_RAM.
3. At least one task is currently delayed (*len*>0), new task being delayed has a delay time less than *t*: the data held in *PidDQ* and *t* is put into the DQ_RAM, the new task's ID is put into *PidDQ*, and the new task's delay time is put into *t*.

This method ensures that the task with the smallest delay time (i.e., the one that will be made active again the soonest) is being held in *PidDQ* and *t*. This makes it easy to check for when the delay is up, without having to constantly access the DQ_RAM.

The DQOSM also checks for when *VCounter* has reached (or passed) the delay time *t*. If only one task is currently delayed, *t* is set back to zero and the DQOSM synchronizes with the RQSM on *QE* so that the task can be made valid again. If more than one task is currently delayed, the synchronization with *QE* is still performed, but then the DQ_RAM needs to be searched for the task with the next smallest delay time. The search is made in a similar way to the search made in the RQSM when a new task to run must be found. When the next task is found, it is removed from the DQ_RAM and put in the variables *PidDQ* and *t*.

Making a *delay until* call in Ada ensures that the task will not run again until at least the time specified. However, it may end up running later than that time. The DQOSM can only check to see if the time *t* has been reached when it is in state *o0*, so it is possible that while the DQOSM is in the process of searching the DQ_RAM for the next task due to end its delay, the time may be reached. But as soon as the DQOSM returns to state *o0*, this will be acknowledged, and if needed a task-switch interrupt will be made.

6.5 Protected Objects

Even though the number of POs that can be declared is variable, each PO routine only needs to be written in VHDL code once. Additionally, since the Protected Procedure and Protected Function routines have such similar behavior, and since only one Procedure or one Function in a given PO can be accessed at one time, one state machine can account for both types of calls. The routines are combined to create one Protected Object. Some signal arbitration must be done, since the different PO routines are able to set the same signals, such as those used to access the RQ_RAM. Since only one PO routine is active at a time, if any routine that is part of the PO sets any of these signals, it can be said that the PO in general is setting them. From the single PO, the total number of POs needed is generated. The number of POs should always be a power of two. This is because the signals from all the POs, as described earlier, are combined into arrays, and values in the arrays that correspond to the current PO being used is accessed by an index, *POId*. *POId* should only be able to access valid POs; if the total number of POs is not a power of two, an error somewhere (such as a bit flip) could result in an access of invalid data. Also, PO(0) is reserved as a null PO. This is because *POId* uses a default value of zero. It is unsafe for this default value to access a valid PO, because then the PO could be accessed unintentionally. So if four POs are created, three can be validly accessed by tasks, and they are numbered PO(1), PO(2) and PO(3).

The channel controller code determines when channels related to a single PO are valid. Two channels used are *FindNew* and *Suspend*, which are also used elsewhere, but if they were resolved at a higher level in the code then the state bits for every PO would have to passed up to that higher level code, which would create a more complicated design. By evaluating these here, a *FindNew* signal and a *Suspend* signal for each PO must still be passed to higher level code, but the number of signals passed is still much smaller. The channel *Ef* is also determined here, which is a signal internal to the PO. It is the only exception to the rule that no more than one PO routine cannot transition at once. It is used in the case where a task calling the Procedure routine releases the Entry routine, and will be discussed more below. The two routines transitioning together on *Ef* do not cause a conflict on any of the signals they are both capable of driving; the channel is used here to save one clock cycle.

The first PO state machine handles a Protected Entry call, and it is shown in Figure 27. To start an Entry call, the task sends the *Es* command followed by data in the Lower Parameter Register that tells the kernel the PO's ID and the PO's ceiling priority. For a system with four POs and eight priority levels, the PO's ID would be in the lower two bits of the Lower Parameter Register and the ceiling priority of that PO would be in the third, fourth and fifth bits, as shown in Figure 17.

When the Entry state machine (ESM) receives the *Es* command, it transitions from *E0* to *E1* and sets *BarrierReq* equal to 1 to request that the calling task supply the value of the *Barrier* for that PO. The interface code coordinates the Barrier request as described in section 6.1. If the

Barrier is true (*Barrier* == 1), the ESM raises the calling task's priority to the ceiling priority by writing to the RQ_RAM. Then it continues to process the rest of the Entry commands, including the case of an exception occurring (states *E1* through *E6*). When the Entry call is completed, the RQ_RAM is written to again, to lower the priority of the task back to what it was. Because of this priority change, the ESM synchronizes with the RQSM on *FindNew*, so that the RQSM can check if a task of higher priority is valid to run. This would occur if a task that had been delayed woke up while the Entry was executing, and that task has a higher priority than the running task's normal priority but a lower priority than the Entry's ceiling priority. When the running task's priority is lowered back to normal, it would be interrupted.

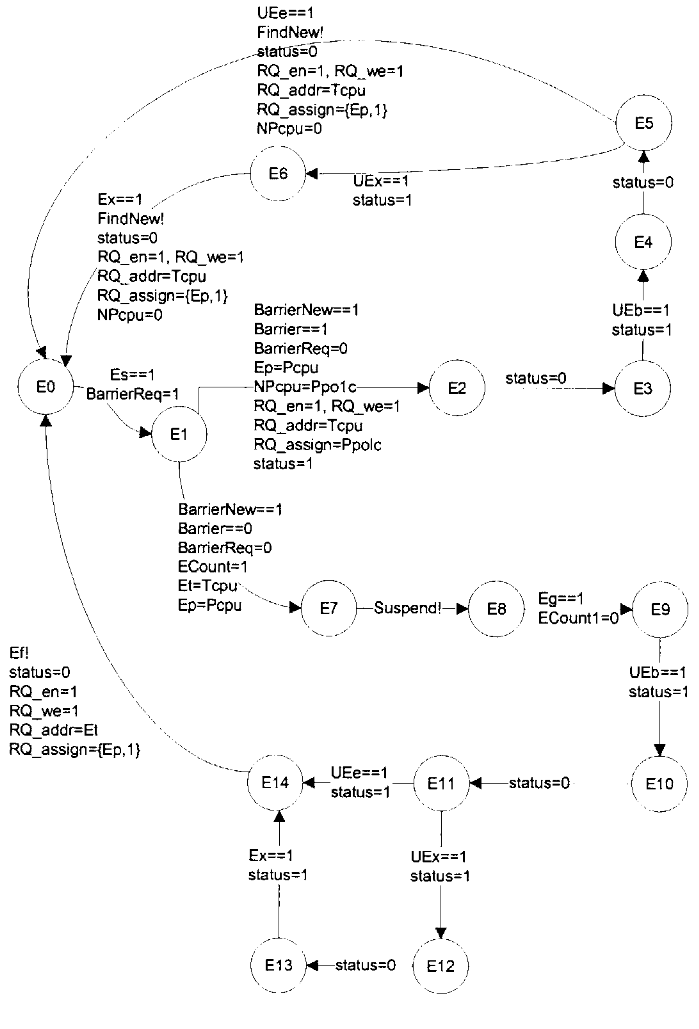

Figure 27. Protected Entry State Machine

The task receives a status back after each Entry command that it sends. All commands after *Es* have a parameter that just has the PO's ID in it (the ceiling priority is no longer needed). After the command to conclude the Entry call, the status indicating the command is complete is not sent until after the *FindNew* branch of the RQSM has been completed. This keeps the task from continuing with another call until it has been determined whether or not an interrupt should occur.

If the result of the Barrier request shows that the *Barrier* value is false (*Barrier* == 0), then the task calling the Entry suspends itself and adds itself to the entry queue (*ECount* := 1). When this happens, the ESM synchronizes with the RQSM on *Suspend*, the current task is made invalid, a new one to run is found and an interrupt is sent to the PPC. The only way for the Entry to be completed is for the Procedure routine of the same PO to be called with the *Barrier* set to true. When this happens, the task calling the Procedure code finishes calling the Entry code. Upon completion of the Entry in this manner, the original task that called the Entry is made valid again.

Code that would call the Entry of PO(2) with ceiling priority four is below. The exception code is included (jump to exception_E if an exception occurs), as well as the behavior for different values of the *Barrier* signal. Note that after making the *Es* command, the task should expect the returning status to indicate a Barrier request. After the Barrier request is complete, it looks for a status indicating that the command has been completed.

```
Write Es, Command Register
Write 0x00000012, Lower Parameter Register
Read status, Status Register
While status != 'Barrier Request'
        Read status, Status Register
Write Barrier[31:0], Lower Parameter Register
Read status, Status Register
While status == 'No Status'
        Read status, Status Register
If(Barrier[POId] == True)
        Write UEb, Command Register
        Write 0x00000002, Lower Parameter Register
        Read status, Status Register
        While status == 'No Status'
                Read status, Status Register
        Set long jump (exception_E)
        Jump Entry
        Write UEe, Command Register
        Write 0x00000002, Lower Parameter Register
        Read status, Status Register
        While status == 'No Status'
                Read status, Status Register

        exception_E:
        Write UEx, Command Register
        Write 0x00000002, Lower Parameter Register
        Read status, Status Register
        While status == 'No Status'
                Read status, Status Register
        Write Ex, Command Register
        Write 0x00000002, Lower Parameter Register
        Read status, Status Register
        While status == 'No Status'
                Read status, Status Register

If(Barrier[POId] == False)
        --- Interrupt to PPC here ---
        Write 0x00000001, Interrupt Acknowledge Register
        Read NextTask, New Task ID Register
```

The second PO state machine handles both Protected Procedure and Protected Function calls, and it is shown in Figure 28.

To start either a Procedure or Function call, the task sends the command *FPs* and a parameter containing the PO's ID and the ceiling priority in the same manner as when an Entry call is made. The Function/Procedure state machine (FPSM) raises the priority of the calling task to the PO ceiling priority by accessing the RQ_RAM and then moves to state *P2*. The FPSM routine handles both normal and exception cases; for normal, it next transitions to state *P3*, and for exception it transitions to state *P7*. After each command, the status is sent indicating the command has been completed. When the command is made to exit the Procedure or Function, the FPSM lowers the task's priority back to what it originally was and synchronizes with the RQSM on *FindNew* to see if a higher priority task exists. The status is set to indicate the exit command has been completed only after the *FindNew* sequence is complete. At states *P3* and *P9* (depending on regular execution or exception), different commands must be sent depending on the type of call, Function (*UFe* and *UFxe*) or Procedure (*UPe* and *UPxe*). For a Procedure call, the FPSM checks to see if the *ECount* variable is set to 1. If it is, then that means there is a task on the entry queue as described earlier. A Barrier request must be made. If the *Barrier* is true, then the calling task first completes the Entry call before exiting the Procedure. If the *ECount* variable is set to 0, or it is 1 but the *Barrier* is false, then the Procedure continues without calling Entry code. For a Function call, the *ECount* variable is not checked and a Barrier request is not made. The FPSM just continues to state *P6* or *P12* to wait for the command to exit the Function.

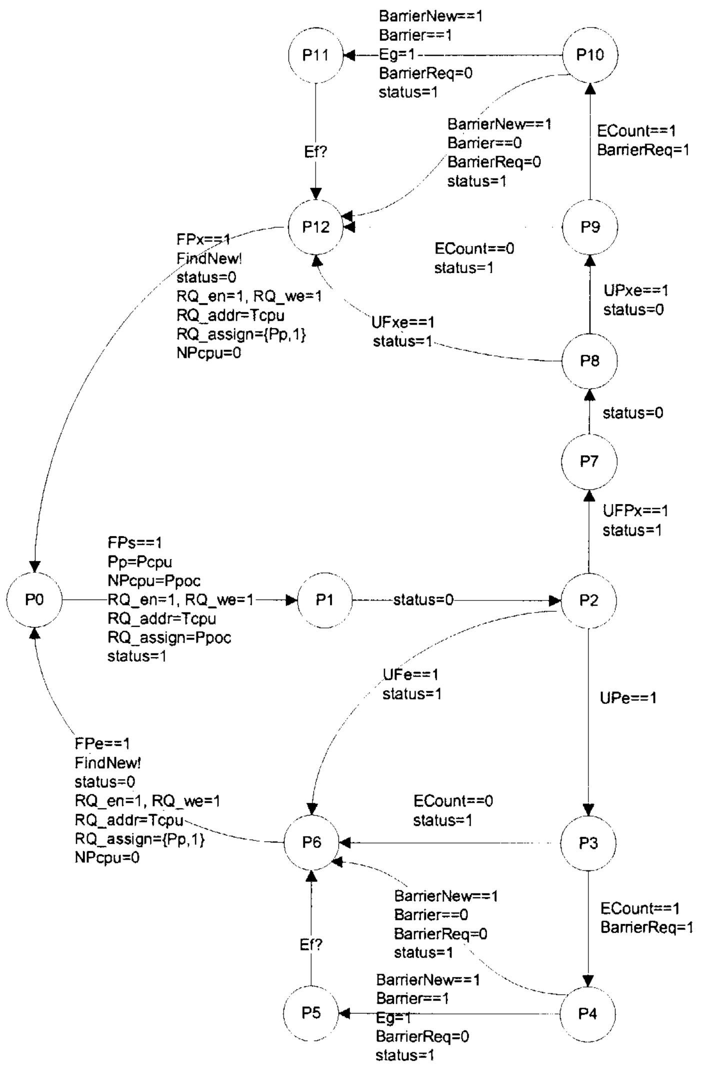

Figure 28. Protected Function/Procedure State Machine

The code used to call the Procedure routine of PO(2) with ceiling priority four is as follows. Again both exception code and code for different *Barrier* values is shown (so it can be seen how the task finishes calling the Entry under certain conditions before completing the

Procedure). The code for calling the Function routine would be similar but much simpler; the status would never return a request for the *Barrier* values.

```
Write FPs, Command Register
Write 0x00000012, Lower Parameter Register
Read status, Status Register
While status == 'No Status'
        Read status, Status Register
Set long jump (exception_P)
Jump Procedure
Write UPe, Command Register
Write 0x00000002, Lower Parameter Register
Read status, Status Register
While status == 'No Status'
        Read status, Status Register
If(status == 'Barrier Request')
        Write Barrier[31:0], Lower Parameter Register
        Read status, Status Register
        While status == 'No Status'
                Read status, Status Register
        If(Barrier[POId] == True)
                Write UEb, Command Register
                Write 0x00000002, Lower Parameter Register
                Read status, Status Register
                While status == 'No Status'
                        Read status, Status Register
                Set long jump (exception_E)
                Jump Entry
                Write UEe, Command Register
                Write 0x00000002, Lower Parameter Register
                Read status, Status Register
                While status == 'No Status'
                        Read status, Status Register

                exception_E:
                Write UEx, Command Register
                Write 0x00000002, Lower Parameter Register
                Read status, Status Register
                While status == 'No Status'
                        Read status, Status Register
                Write Ex, Command Register
                Write 0x00000002, Lower Parameter Register
                Read status, Status Register
                While status == 'No Status'
                        Read status, Status Register
```

```
Write *FPe*, Command Register
Write 0x00000002, Lower Parameter Register
Read *status*, Status Register
While *status* == 'No Status'
        Read *status*, Status Register

exception_P:
Write *UFPx*, Command Register
Write 0x00000002, Lower Parameter Register
Read *status*, Status Register
While *status* == 'No Status'
        Read *status*, Status Register
Write *UPxe*, Command Register
Write 0x00000002, Lower Parameter Register
Read *status*, Status Register
While *status* == 'No Status'
        Read *status*, Status Register
If(*status* == 'Barrier Request')
        ---REPEAT CODE FROM SAME 'IF' STATEMENT ABOVE---
Write *FPx*, Command Register
Write 0x00000002, Lower Parameter Register
Read *status*, Status Register
While *status* == 'No Status'
        Read *status*, Status Register
```

6.6 Task Arbitration

Different state machines in the kernel may request that different tasks run at the same time. For example, a high-priority task may wake up from a delay, so the Ready Queue requests that it be run; at the same time, the running task may be calling a PO that changes its priority. Even though a PO will make the priority change by actually changing the value in the RQ_RAM as described above, the Ready Queue may not get this change immediately, if it is busy doing something else. So, additional code is needed to arbitrate the different requests for what should be run on the PPC. This code checks to see what the priorities are of the different tasks that different processes are requesting to run. Whichever task has the highest priority is declared to be the 'next' task to run, *NTcpu*, and its priority is the 'next' priority, *NPcpu*. (Note that these variable names are used elsewhere, such as in the Ready Queue. Variable names are local to the routines they are in, so they can be reused without conflict.) This task and priority information is passed up to the KISM. If *NPcpu* is greater than the priority of the currently running task (*Pcpu*), and *NTcpu* and the current task (*Tcpu*) are different, than an interrupt occurs. Often *NTcpu* will equal *Tcpu* even if the priorities are not equal; this occurs in the case mentioned of a PO changing the priority of the running task. In this routine, *NTcpu* and *NPcpu* are determined from data from both the PO state machines and the Ready Queue.

6.7 Ada Time

When a *delay until* call is made in Ada, the time is specified at a resolution of the users' choice. This resolution can be as fast as the FPGA clock, or some fraction of it. It is kept track of in the code as a counter *VCounter*, as described in section 6.1. The kernel must be aware of the resolution specified in Ada in order to accurately determine when a *delay until* time is up. This is set during the elaboration phase using the *SetFreq* command, after all tasks are sent to the kernel (using the *Create* command). The frequency is set by sending data in the lower eight bits of the Lower Parameter Register (the higher 24 bits must be zeros). A parameter of one (00000001) sets the *VCounter* resolution equal to that of the kernel clock; a parameter of two (00000010) sets it to half the kernel clock, a parameter of 3 (00000011) set it to one third of the kernel clock, etc. When this value is set, the kernel begins to count up from zero. An example of the compiled code used to set the frequency to one quarter of the kernel clock speed is:

```
Write SetFreq, Command Register
Write 0x00000004, Lower Parameter Register
Read status, Status Register
While status == 'No Status'
        Read status, Status Register
```

The counter cannot be restarted or stopped once it is set in the elaboration phase. With variables set as they are now, *VCounter* can currently count up to 64 bits. At one microsecond resolution, the counter will be useful for over 500,000 years.

Chapter 7
Testing and Implementation

The RavenHaRT Kernel has been tested for a very small design, one with only four tasks and four POs. The results of these tests, and of attempts made to implement the design so that it can be run on the development board, demonstrate its functionality but were hindered by problems with the Xilinx tools. A summary of the test files is included in Appendix B.

7.1 Kernel Testing

The VHDL code for the kernel was written using the Xilinx tool ISE Foundation 5.2i. ISE can be linked to Model Technology's tool ModelSim so that both behavioral and timing testing can be performed. ModelSim 5.8 PE with the SWIFT interface [Mod] was used. Testing done in ISE examines the kernel as a stand-alone unit. It is simpler to first test it this way before integrating it into a larger system (with other components such as the PPC). Two types of tests were performed:

- Behavioral tests, which demonstrate that the code has the correct logical functionality.
- Timing tests, which are performed after the tool attempts to place-and-route the design on the target FPGA. They help evaluate whether the logic is still valid at certain clock speeds given the current design layout.

Note though that the on-chip layout will change when the kernel is integrated into a system. Variances in the clock speed and design implementation for when the full system is discussed in the next section.

For behavioral testing, each VHDL state machine was first tested individually. As different parts of the kernel were combined, incremental tests were done. After the incremental tests demonstrated correct behavior, a number of top-level tests were performed. The tests were all written in VHDL. A sample result waveform for a behavioral test in ModelSim is shown in Figure 29.

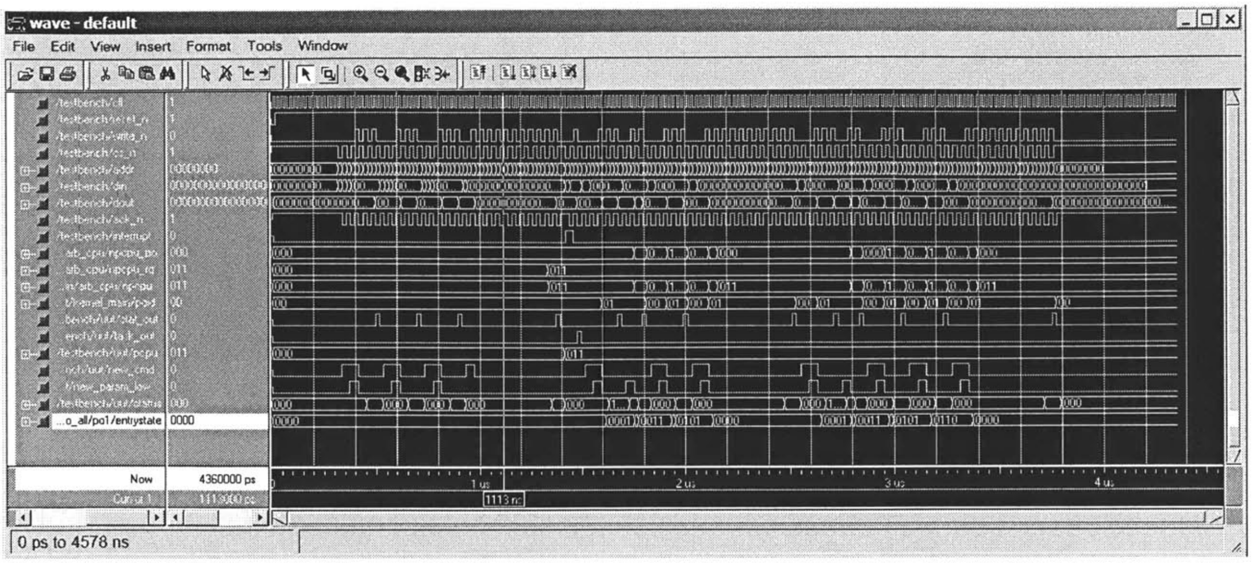

Figure 29. Example Waveform From Test of Entry Routine in ISE

Timing analysis was performed after ISE implemented the design. For implementation, a number of settings in ISE can be adjusted. Any setting that required the tool to optimize the design for speed was set, as well as any that had the tool use extra effort in determining the chip layout. Additionally, hierarchy was kept so that the general structure of the VHDL code was retained and so that signals could be easily identified in testing. The remainder of the settings were kept at the tool's default values. Additionally, the file *myconstraints.ucf* (in Appendix A.9) had to be used for implementation. In this file, constraints such as the clock speed and input and output offsets can be specified. After some experimentation, it was discovered that the tool could not make the clock run faster than 80 MHz. That is listed as the clock speed currently in the file, and some offsets are constrained as well. However, pin constraints were not declared since the kernel was considered in this phase of testing to be a stand-alone unit, so the offset constraints do not have much meaning. More information on the Xilinx tools and timing can be found in chapter 8.

For simplicity, all timing tests were run with a clock speed of 50 MHz. Only top-level timing tests were performed, and the set of tests is identical to those used for behavioral testing. A summary of all top-level tests is as follows:

- Elaboration: creates tasks, sets *VCounter*, and finds the highest priority task to run

- Entry: elaboration occurs, then a task calls the Entry with *Barrier* == 1, for normal case and exception
- Procedure: elaboration occurs, then a task calls the Procedure with *ECount* == 0, for normal case and exception
- Function: elaboration occurs, then a task calls the Function for normal case and exception
- Delay Until: elaboration occurs, then each task periodically delays
- Procedure-calling-Entry: elaboration occurs, then a task calls the Entry with *Barrier* == 0 so that it suspends. A second task calls the Procedure and from there the Entry is completed and the first task is released
- Interrupt Entry (multiple tests): elaboration occurs, then a task is delayed for varying lengths of time. A second task calls the Entry with *Barrier* == 1, and it is interrupted by the first task
- Interrupt Procedure (multiple tests): elaboration occurs, then a task is delayed for varying lengths of time. A second task calls the Procedure with *ECount* == 0, and it is interrupted by the first task
- Interrupt Procedure-calling-Entry (multiple tests): elaboration occurs, then a task is delayed for various lengths of time. A second task calls the Entry with *Barrier* == 0 so that it also suspends. A third task calls the Procedure and then completes the Entry, but this is interrupted by the first task

The purpose of these tests was to see if the kernel behaved correctly when tasks would perform various calls, such as making a delay or calling a PO. Various cases of one task interrupting another were tested to make sure that the system recovered correctly from an interrupt. The tests all demonstrate that the kernel works accurately, but they of course do not account for every possible case. Exhaustive testing in general is not possible, and even longer tests (that represent a system running for hours or days or longer) are not feasible because of the time they would take, and because it would be nearly impossible to verify the correctness of the waveforms they produce through manual examination. Tests with incorrect inputs could be performed; however, it has been assumed in this design that the Ada code will be written correctly and the Ada compiler modified correctly so that incorrect commands and parameters are not sent, and so that certain incorrect and unexpected cases cannot occur.

7.2 System Testing

After kernel tests in ISE were complete, the kernel was tested and implemented as part of a larger system. This was done using Xilinx's Embedded Development Kit 3.2 (EDK). In EDK, a project can be set up that contains the VHDL kernel, the PPC, buses, and any other peripherals desired in the system, such as external RAM. Code can then be written for the PPC, and the system can be simulated in ModelSim to make sure that all peripheral devices, including the kernel, are behaving correctly.

The PPC provides two options for communication with peripheral devices. One is the On-Chip Peripheral Bus (OPB), the other is the Processor Local Bus (PLB). The PLB is faster, since the peripheral would be directly on the processor bus. The OPB is slower, since the processor is connected to it by a PLB-to-OPB bridge. The OPB is more commonly used, and was therefore easier to set up in EDK since support code for the OPB bus was provided as part of the tool. The kernel should eventually run on the PLB, but for testing purposes the OPB was determined to be satisfactory.

Tests were written in C code similar to the tests listed in the previous section in VHDL. C code was chosen because EDK has a built in C compiler; a cross-compiler will be needed to compile Ada code for the PPC, and this has not yet been developed. C code can be written that mimics the behavior of the compiled Ada code. It sends kernel commands to various registers, and handles interrupts to the PPC. As in ISE, there is an option to run either behavioral or timing tests. All behavioral tests worked successfully and demonstrated correct functionality. An example result waveform is shown in Figure 30.

Performing timing tests was problematic because of the difficulties in using EDK. Although many of the same implementation settings available in ISE appeared to be available in EDK, they were more difficult to find and change. When possible, the settings were adjusted to optimize for speed and to have to tool use highest effort. Also, settings were changed to keep the code hierarchy, but this did not appear to work correctly. Unlike in ISE, original signal names were not retained in ModelSim when the system was analyzed for timing. Instead, the signal names seemed to indicate that the design had been flattened, making it nearly impossible to determine which signals to examine to ascertain correct behavior.

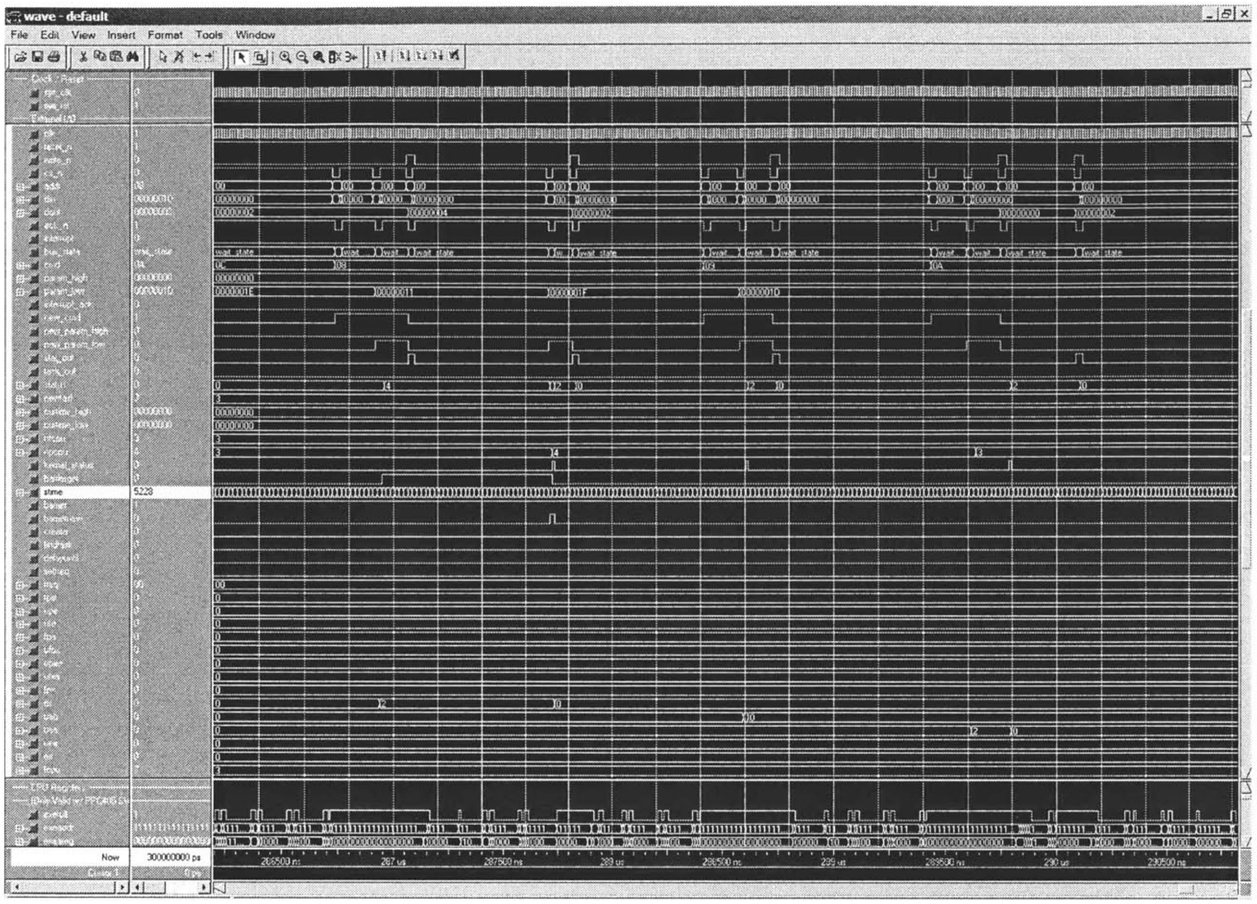

Figure 30. Example Waveform From Test of Entry Routine in EDK

An alternative to running timing simulations was to run actual tests on the development board. EDK provides means to program the Virtex-II Pro Chip, so that the C code written as test code can be run as if it were code in an actual system. Edits are made to the code so that data is printed to the computer's screen giving notification that it has reached certain points.

To run the system correctly on the development board, the kernel was implemented with the setting described. A constraints file is also needed, although a different one is used than in ISE. This constraints file, *system.ucf* (Appendix A.9), has both a clock constraint, as well as all the pin constraints. The clock constraint indicates that the clock should run at 100 MHz. When the EDK implementation tool was run, the results showed that this constraint was not met. However, where in ISE the tool could not implement the design at a faster clock than 80 MHz, in EDK the design works at approximately 95 MHz. This result is unexpected since a larger design is implemented in EDK, because additional VHDL code (most supplied by EDK) is needed to integrate the FPGA with the other system components.

The development board tests were run with the clock left at 100 MHz, since that is the speed of the PPC clock. The tests worked at this speed, most likely because the tests were simple and the tool determined the actual clock to be close to that speed. More complicated tests were not possible, since the only way to determine correct execution was to examine any data printed back to the computer screen. Running real Ada programs would give more indication of long-term, correct kernel behavior.

Chapter 8
Future Work

As described in chapter 7, tests of the basic kernel functionality show that it works correctly. However, there are many improvements that can be made. These include optimizing the implementation through use of the Xilinx tools and changes to the VHDL code itself. Also, more extensive testing could be performed to further prove its correct functional and timing behavior.

8.1 Use of Tools

As mentioned in chapter 7, attempts to perform timing tests resulted in difficulties specifying and determining the clock speed of the FPGA in both ISE and EDK. Implementation in each tool provided different and unexpected results. EDK results are more important to analyze, since these are the ones that determine what is actually loaded onto the FPGA. However, the two tools are closely related and use many of the same internal tools in design implementation, so the differences should be investigated.

A design implemented in either ISE or EDK can be examined using the Timing Analyzer, which is an associated Xilinx tool. This tool will identify worst-case timing paths, which can then be constrained appropriately. Repeated analysis in this way may allow the FPGA clock to run faster.

Additionally, there may be more implementation settings that could be changed, particularly in EDK. Changing settings in EDK is not straightforward, and requires editing files that are not typically accessed by users. All available settings that can be changed can be found in [Xil03].

In general, the Xilinx tools could be further used to optimize the kernel design, but this would require a very thorough understanding of how best to use them. Also, minor bugs were discovered in using the tools, which may or may not have affected results. Newer versions may solve these problems, and perhaps prove easier to use.

8.2 Code Changes

Various code changes may or may not affect the implementation and speed of the design. The first of these possible changes concerns how task-switch interrupts are handled. Currently, an interrupt is always performed when a new task is found to run. However, the commands *FindTask* and *DelayUntil* always result in a new task being found. The design could be simplified by having the hardware not perform an interrupt after these tasks, and instead just requiring that the task look at the new task register after the status shows the *FindTask* or *DelayUntil* command has been completed. Also, the *interrupt_ack* signal from the task could probably be eliminated. The hardware already waits and makes sure the PPC has read the new task, so the interrupt acknowledge signal does not add any new information. These two changes will reduce the number of clock cycles needed to perform the interrupt operations, and may possibly create simpler, faster code.

Another possible code change concerns how data is stored and searched for in the RAMs. There are many different software (and VHDL) methods for sorting and storing large amounts of data, and for searching through arrays for information. The kernel uses these types of algorithms in handling task data in the Ready Queue and the Delay Queue. When determining how to translate the kernel model into VHDL, the decision was made to use the algorithms that were used in the model for consistency, and so that the VHDL kernel would be as similar to the model as possible.

The RAM associated with the Ready Queue is filled during elaboration, as was described in chapter 7. The RQ_RAM is indexed by the task's ID values, and the tasks and their data are entered into the RQ_RAM in the order in which they are sent to the kernel by the *Create* instruction. This makes filling it during elaboration very simple. However, when a search needs to be made during normal operation to find the task with the highest priority that is able to run, a brute-force search is done. The Ready Queue always must search through every element in the RQ_RAM to make sure it has found the highest priority task. While this search is deterministic, it is potentially slow, and could take many clock cycles.

A possible alternative to this method would be to sort the tasks better during elaboration. Instead of being left in the RQ_RAM in the order in which they were created, the tasks could be sorted by priority. Then, when the Ready Queue needs to find a task to run, it just needs to start at the beginning of the RQ_RAM and find the first task that is valid to run. Searches would take varying amounts of time, with the longest search being equal to that of the current method. Of course, there are other situations that need to be accounted for if this method is used. For instance, calling a PO changes the priority of the calling task, and this would need to be handled somehow, either by resorting the RQ_RAM, or by altering the search method.

The Delay Queue faces a similar problem. The DQ_RAM is also set up in the order that tasks are created. It could possibly be sorted according to delay time, although then it would constantly need to be resorted every time a task makes a *delay until* call, and every time a task's delay time is up.

Changing the sorting and searching strategies may or may not be beneficial. The methods mentioned, as well as others, could be implemented and analyzed to determine what algorithms are the best to use. Besides looking at just the average and worst-case number of clock cycles to do something such as complete a search or resort data, the way that each algorithm affects the timing of the whole system must be considered. Resorting the data may make search times shorter, but if the resort causes a delay in executing a task, it may not be beneficial to do.

Also, the Delay Queue currently does not take task priority into consideration, and this could possibly cause a problem in the case that two tasks delay until the exact same time. For example, take a system with two tasks, Task A with a priority of 5 and Task B with a priority of 7. Task B is suspended for some reason, so Task A runs and delays until time T. Task B becomes active to run, and also delays until time T. Since their relative priorities are not considered by the Delay Queue, at time T Task A will first wake up and possibly begin to run on the PPC. Then, Task B will wake up. There is a chance in this case that Task A will begin to run before Task B interrupts. If the Delay Queue checked task priorities, Task B would correctly run first. This is a very specific case but it should be accounted for in the design.

Lastly, the Ready Queue could be altered to contain more information about the tasks. Ada stores task information in Task Control Blocks (TCB). This information includes the location in memory where the task is actually stored. The memory location of each task could be given to the kernel and stored in the RQ_RAM. Then, when a task-switch occurs, instead of the PPC receiving the new task's ID from the kernel, it could receive the memory location of the task and could immediately start running that task.

8.3 Testing and Analysis

The tests performed to this point demonstrate basic behavior, but most are behavioral and all only mimic the actions of real code. More extensive real-time tests will be easier to develop once the Ada compiler is modified to work with the RavenHaRT Kernel. Then, real Ada programs that have real functionality (as opposed to just imitating kernel calls as the C code does) can be written, and tested to see if they execute on the kernel correctly.

Also, the differences between the timing properties of the implemented kernel and the timing properties of the UPPAAL model have not been examined. It is expected that the implemented kernel will have very different timing properties than the original model of the kernel, because of reasons described earlier in this paper (such as the use of urgent states in UPPAAL). However, many of these differences might be acceptable. They can also be easily measured. Since all state transitions in the implementation occur on the rising edge of the FPGA's clock, the amount of time taken for different actions can be determined if the number of state transitions needed to perform them is known (assuming the state machine will not wait at any state for external data). The worst-case times for searches for new tasks can be found, as well as for a task waking up from delay. Data such as this, combined with statistics for communication over the bus, will tell how long, for example, the delay is from when a task should be running (the time when it has become valid and has become the highest priority task) to when it actually begins to run. From this information, it can be seen if timing behavior is acceptable, or if changes need to be made to the VHDL code.

Alternatively, to check that the implementation has acceptable timing behavior, the assertions used in the model could be analyzed, and timing tests of the VHDL kernel could be developed based on the assertions. While doing this would be very beneficial, it is very tedious and prone to human error. Another solution would be to use one of the tools developed as part of the Gurkh project. This tool [NL03] automatically converts VHDL code into formal models that can be understood by UPPAAL. The implemented RavenHaRT Kernel should be converted back to a model using this tool. It can then be compared with the original model, and checked using UPPAAL to make sure that it has the expected properties.

References

[Ada] www.adaic.org

[Ada99] "Programming Languages – Guide for the Use of Ada Programming Language in High Integrity Systems", 1999, ISO/IEC JTC1 /SC 22/WG 9 N 359r, ISO/IEC DTR 15942.

[AL03] L. Asplund, K. Lundqvist, "The Gurkh Project: A Framework for Verification and Execution of Mission Critical Applications", 22nd Digital Avionics Systems Conference, 2003.

[AFLS96] J. Adomat, J. Furunas, L. Lindh, J. Starner, "Real-Time Kernel in Hardware RTU: A step toward deterministic and high performance real-time systems", 8th Euromicro Workshop on Real-Time Systems, 1996, pp 164-168.

[AFM+02] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, W. Yi, "TIMES – A Tool for Modeling and Implementation of Embedded Systems", 8th International Conference, TACAS 2002, part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, LNCS 2280, pp 460-464.

[ATAC95] http://www.estec.esa.nl/wsmwww/components/atac/atac.html

[BDR98] A. Burns, B. Dobbing, G. Romanski, "The Ravensccar Tasking Profile for High Integrity Real-Time Programs", Ada-Europe 98, LNCS 1411, 1998, pp 263-275.

[BDV03] A. Burns, B. Dobbing, T. Vardanega, "Guide for the Use of the Ada Ravenscar Profile in High Integrity Systems", University of York Technical Report YCS-2003-348, 2003.

[BW98] A. Burns, A. Wellings, "Concurrency in Ada", Second Edition, Cambridge University Press, New York, 1998.

[CGW91] W.J. Cullyer, S.J. Goodenough, B.A. Wichmann, "The Choice of Computer Languages for Use in Safety-Critical Systems", Software Engineering Journal, 19991.

[CW96] E. Clarke, J. Wing, "Formal Methods: State of the Art and Future Directions", ACM Computing Surveys, 1996.

[Fur00] J. Furunas, "Benchmarking of a Real-Time System that utilizes a booster", International Conference on Parallel and Distributed Processing Techniques and Applications, 2000.

[Kel03] T. Klevin, "Multitasking Operations Require More Hardware Based RTOSes", EE Times, 2003, www.eetimes.com/story/OEG20030221S0027

[LA03] K. Lundqvist, L. Asplund, "A Ravenscar Compliant Run-Time Kernel for Safety-Critical Systems", Real-Time Systems Volume 24, 2003, pp 29-54.

[Lin91] L. Lindh, "FASTCHART – A Fast Time Deterministic CPU and Hardware Based Real-Time-Kernel", IEEE Press, Euromicro Workshop on Real Time Systems, 1991, pp 36-40.

[Lin92] L. Lindh, "FASTHARD – A Fast Time Deterministic HARDware Based Real-time Kernel", 4th Euromicro Workshop on Real-Time Systems, 1992, pp 21-25.

[LK99] L. Lindh, T. Kelvin, "Scalable Architecture for Real-time Applications and use of bus-monitoring", Proceedings of Sixth International Conference on Real-Time Computing Systems ad Applications, 1999, pp 208-211.

[LPY97] K. Larsen, P. Pettersson, W. Yi, "UPPAAL in a Nutshell", Springer International Journal of Software Tools for Technology Transfer, 1997.

[LS91] L. Lindh, F. Stanischewski, "FASTCHART – Idea and Implementation", IEEE Press, International Conference on Computer Design (ICCD), 1991, pp 401-404.

[LSF95] L. Lindh, J. Starner, J. Furunas, "From Single to Multiprocessor Real-Time Kernels in Hardware", IEEE Press, Real-Time Technology and Applications Symposium, 1995, pp 42-43.

[LSF+98] L. Lindh, J. Starner, J. Furunas, J. Adomat, M. El Shobaki, "Hardware Accelerator for Single and Mulitprocessor Real-Time Operating Systems", 7th Swedish Workshop on computer Systems Architecture, 1998.

[Mem03] "Virtex-II Pro (P4/P7) Development Board User's Guide", version 3.0, Memec Design, 2003, http://www.memec.co.jp/html/xilinx/eboard/docs/dskit2v/v2pro_board_users_guide_1_0.pdf

[Mod] www.model.com

[NL03] C. Nehme, K. Lundqvist, "A Tool for Translating VHDL to Finite State Machines", 22nd Digital Avionics System Conference, 2003.

[NUI+95] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, M. Imai, "Hardware Implementation of a Real-Time Operating System", IEEE Press, 12th TRON Project International Symposium, 1995.

[PL00] P. Pettersson, K. Larsen, "Uppaal2K", Bulletin of the European Association for Theoretical Computer Science Volume 70, 2000, pp 40-44.

[PPC01] "PowerPC 405 Embedded Processor Core User's Manual", Fifth Edition, 2001, http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/D060DB54BD4DC4F2872569D2004A30D6/$file/405_um.pdf

[Rei97] R. Reihle, "Can Software Be Safe? – An Ada Viewpoint", Embedded Systems Programming, 1997, www.embedded.com/97/feat9612.htm

[Roo91] J. Roos, "Designing a Real-Time Coprocessor for Ada Tasking", IEEE Press, Design and Test of Computers, Volume 8, Issue 1, 1991, pp 67-79.

[SAN+03] T. Samuelsson, M. Akerholm, P. Nygren, J. Starner, L. Lindh, "A Comparison of Multiprocessor Real-Time Operating Systems Implemented in Hardware and Software", International Workshop on Advanced Real-Time Operating System Services, 2003.

[Shaw01] A. Shaw, "Real-Time Systems and Software", John Wiley and Sons, Inc, New York, 2001.

[Upp] www.uppaal.com

[Xil02] "Virtex-II Pro Platform FPGA Handbook", v1.0, 2002, www.xilinx.com

[Xil03] "Development System Reference Guide", 2003, http://toolbox.xilinx.com/docsan/xilinx6/books/manuals.htm

[Yov97] S. Yovine, "Kronos: A Verification Tool for Real-Time Systems", Springer International Journal on Software Tools for Technology Transfer, 1997, pp 134-152.

[Zei95] S. Zeigler, "Comparing Development Costs of C and Ada", Rational Software Corporation, 1995, www.adaic.com/whyada/ada-vs-c/cada_art.html

Appendix A
VHDL Kernel Code

A.1 Interface

```
-----------------------------------------------------------------------------------------------
-- NAME: interface_ppc.vhd
--
-- INPUTS:    clk          FPGA clock
--            reset_n      FPGA reset, active low
--            write_n      PPC reading (high) or writing (low)
--            cs_n         PPC chip select, active low
--            addr         address of register PPC is reading from or writing too
--            din          input data
--
-- OUPUTS:    dout         output data
--            ack_n        acknowledge that read or write occurred
--            interrupt    task switch interrupt to PPC
--
-- DESCRIPTION: top leveL kernel code.  Contains the Bus Interface State Machine (BISM)
--        and the Kernel Interface State Machine (KISM), and code to connect to rest of kernel

-- NOTES:
--        this scheme works for up to 32 POs, and 32 bits must be sent over bus (or set to zero)
--        when barrier is sent, regardless of NUMPO (total number of POs, defined in
--        myvariables.vhd)

--        SW assumes cmd given is NOT done until it receives status = CMDDONE, so if it gets
--        an interrupt right after sending a cmd, and hasn't yet gotten the CMDDONE, then it will
--        need to resend command when that tasks resumes later

--        all params should be addressed to param_low
--        except for DelayUntil: first low bits must be sent on param_low, then high on
--        param_high

--        on create: param_low(BITPRIO-1 downto 0) holds TaskPrio (BITPRIO is number of bits
--        needed to represent the priority, defined in myvariables.vhd)
--        param_low(BITTASKS+BITPRIO-1 downto BITPRIO) holds TaskID (BITTASKS is
--        number of bits needed to represent the task, defined in myvariables.vhd)

--        on fps and es: param_low(BITPO-1 downto 0) holds POId (BITPO is number of bits
--        needed to represent the PO ID, defined in myvariables.vhd)
--        param_low(BITPO+BITPRIO-1 downto BITPO) holds Ceil Prio
--        on other PO cmds: param_low(BITPO-1 downto 0) holds POId

--        on setfreq: param_low(FREQREG-1 downto 0) holds fractional frequency (FREQREG is
--        the register size allowed for setting the frequency, defined in myvariables.vhd)
--      on gettime: sw should first check for status=CMDDONE, then read from the curtime regs
----------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.variables.ALL;

entity interface_ppc is
   Port ( clk           : in std_logic;
         reset_n        : in std_logic;
         write_n        : in std_logic;
         cs_n           : in std_logic;
         addr           : in std_logic_vector(NREGS-1 downto 0);
         din            : in std_logic_vector(REG_LENGTH-1 downto 0);
         dout           : out std_logic_vector(REG_LENGTH-1 downto 0);
         ack_n          : out std_logic;
         interrupt      : out std_logic);
end interface_ppc;

architecture Behavioral of interface_ppc is

         ------- state variables for bus interface ---------------
         type bus_state_type is (wait_state, ack_state);
         signal bus_state               : bus_state_type;

         ------- regs holding input from bus ---------------------
         signal cmd                     : std_logic_vector(CMDREG-1 downto 0);
         signal param_high              : std_logic_vector(PARAMREG-1 downto 0);
         signal param_low               : std_logic_vector(PARAMREG-1 downto 0);
         signal interrupt_ack           : std_logic;

         ------- signals set when above regs are filled ----------
         signal new_cmd                 : std_logic;
         signal new_param_high          : std_logic;
         signal new_param_low           : std_logic;
         signal stat_out                : std_logic;
         signal task_out                : std_logic;

         ------- regs holding output to bus ----------------------
         signal status                  : std_logic_vector(STATREG-1 downto 0);
         signal newtask                 : std_logic_vector(BITTASKS-1 downto 0);
         signal curTime_high            : std_logic_vector(REG_LENGTH-1 downto 0);
         signal curTime_low             : std_logic_vector(REG_LENGTH-1 downto 0);
```

```
------- signals interface gets from kernel ---------------
signal NTcpu              : std_logic_vector(BITTASKS-1 downto 0);
signal NPcpu              : std_logic_vector(BITPRIO-1 downto 0);
signal kernel_status      : std_logic;
signal BarrierGet         : std_logic;
signal sTime              : std_logic_vector(MAXTIME-1 downto 0);

------- signals interface sends to kernel ----------------
signal barrier            : std_logic;
signal barrierNew         : std_logic;
signal create             : std_logic;
signal findtask           : std_logic;
signal delayuntil         : std_logic;
signal setfreq            : std_logic;
signal freq               : std_logic_vector(FREQREG-1 downto 0);
signal fps                : std_logic_vector(NUMPO-1 downto 0);
signal upe                : std_logic_vector(NUMPO-1 downto 0);
signal ufe                : std_logic_vector(NUMPO-1 downto 0);
signal fpe                : std_logic_vector(NUMPO-1 downto 0);
signal ufpx               : std_logic_vector(NUMPO-1 downto 0);
signal upxe               : std_logic_vector(NUMPO-1 downto 0);
signal ufxe               : std_logic_vector(NUMPO-1 downto 0);
signal fpx                : std_logic_vector(NUMPO-1 downto 0);
signal es                 : std_logic_vector(NUMPO-1 downto 0);
signal ueb                : std_logic_vector(NUMPO-1 downto 0);
signal uee                : std_logic_vector(NUMPO-1 downto 0);
signal uex                : std_logic_vector(NUMPO-1 downto 0);
signal ex                 : std_logic_vector(NUMPO-1 downto 0);
signal Tcpu               : std_logic_vector(BITTASKS-1 downto 0);
signal Pcpu               : std_logic_vector(BITPRIO-1 downto 0);
signal DUntil             : std_logic_vector(MAXDUNTIL-1 downto 0);
signal TaskID             : std_logic_vector(BITTASKS-1 downto 0);
signal TaskPrio           : std_logic_vector(BITPRIO-1 downto 0);
signal CeilPrio           : std_logic_vector(BITPRIO-1 downto 0);
signal POId               : std_logic_vector(BITPO-1 downto 0);

------- state variables for interface --------------------
type state_type  is (i0, i1, i2, i3, i4, i5, i6, i7, i8, i9, i10);
signal state              : state_type;

-- connection to rest of kernel
component kernel_internal
        port (  clk          : in std_logic;
                reset_n      : in std_logic;
                DelayUntil   : in std_logic;
                SetFreq      : in std_logic;
```

```
                    Freq            : in std_logic_vector(FREQREG-1 downto 0);
                    DUntil          : in std_logic_vector(MAXDUNTIL-1 downto 0);
                    Tcpu            : in std_logic_vector(BITTASKS-1 downto 0);
                    Pcpu            : in std_logic_vector(BITPRIO-1 downto 0);
                    CeilPrio        : in std_logic_vector(BITPRIO-1 downto 0);
                    FPs             : in std_logic_vector(NUMPO-1 downto 0);
                    Upe             : in std_logic_vector(NUMPO-1 downto 0);
                    Ufe             : in std_logic_vector(NUMPO-1 downto 0);
                    Ufpx            : in std_logic_vector(NUMPO-1 downto 0);
                    UPxe            : in std_logic_vector(NUMPO-1 downto 0);
                    UFxe            : in std_logic_vector(NUMPO-1 downto 0);
                    Barrier         : in std_logic;
                    BarrierNew      : in std_logic;
                    FPe             : in std_logic_vector(NUMPO-1 downto 0);
                    FPx             : in std_logic_vector(NUMPO-1 downto 0);
                    Es              : in std_logic_vector(NUMPO-1 downto 0);
                    UEb             : in std_logic_vector(NUMPO-1 downto 0);
                    UEe             : in std_logic_vector(NUMPO-1 downto 0);
                    UEx             : in std_logic_vector(NUMPO-1 downto 0);
                    Ex              : in std_logic_vector(NUMPO-1 downto 0);
                    Create          : in std_logic;
                    FindTask        : in std_logic;
                    POId            : in std_logic_vector(BITPO-1 downto 0);
                    TaskId          : in std_logic_vector(BITTASKS-1 downto 0);
                    TaskPrio        : in std_logic_vector(BITPRIO-1 downto 0);
                    BarrierGet      : out std_logic;
                    NPcpu           : out std_logic_vector(BITPRIO-1 downto 0);
                    NTcpu           : out std_logic_vector(BITTASKS-1 downto 0);
                    Status          : out std_logic;
                    sTime           : out std_logic_vector(MAXTIME-1 downto 0));
        end component;

begin
        kernel_main : kernel_internal
                port map (clk=>clk, reset_n=>reset_n, DelayUntil=>delayuntil, SetFreq=>setfreq,
                        Freq=>freq, DUntil=>Duntil, Tcpu=>Tcpu, Pcpu=>Pcpu,
                        CeilPrio=>CeilPrio, FPs=>fps, Upe=>upe, Ufe=>ufe, Ufpx=>ufpx,
                        UPxe=>upxe, UFxe=>ufxe, Barrier=>barrier, BarrierNew=>barrierNew,
                        FPe=>fpe, FPx=>fpx, Es=>es, UEb=>ueb, UEe=>uee, UEx=>uex,
                        Ex=>ex, Create=>create, FindTask=>findtask, POId=>POId,
                        TaskId=>TaskId, TaskPrio=>TaskPrio,BarrierGet=>BarrierGet,
                        NPcpu=>NPcpu, NTcpu=>NTcpu, Status=>kernel_status,
                        sTime=>sTime);

        -- Bus Interface State Machine
        BISM : process(clk, reset_n)
```

```
begin
	if(reset_n='0') then
		bus_state <= wait_state;
		dout <= (others => '0');
		ack_n <= '1';
		cmd <= (others => '0');
		param_high <= (others => '0');
		param_low <= (others => '0');
		interrupt_ack <= '0';
		new_cmd <= '0';
		new_param_high <= '0';
		new_param_low <= '0';
		stat_out <= '0';
		task_out <= '0';
	elsif(clk'event and clk='1') then
		case bus_state is
			when wait_state =>
				if(cs_n='0') then
					-- sw writes to cmd reg
					if(addr = addr_reg1 and write_n = '0') then
						cmd <= din(CMDREG-1 downto 0);
						new_cmd <= '1';
						ack_n <= '0';
						bus_state <= ack_state;
					-- sw writes to lower param reg: param or barrier
					elsif(addr = addr_reg2 and write_n = '0') then
						param_low <= din;
						new_param_low <= '1';
						ack_n <= '0';
						bus_state <= ack_state;
					-- sw writes to higher param reg (for delay until)
					elsif(addr = addr_reg3 and write_n = '0') then
						param_high <= din;
						new_param_high <= '1';
						ack_n <= '0';
						bus_state <= ack_state;
					-- sw writes to interrupt acknowledge signal
					elsif(addr = addr_reg4 and write_n = '0') then
						interrupt_ack <= din(0);
						ack_n <= '0';
						bus_state <= ack_state;
					-- hw writes status reg out; all sigs are set low
					elsif(addr = addr_reg5 and write_n = '1') then
						dout(STATREG-1 downto 0) <= status;
						dout(REG_LENGTH-1 downto STATREG)
							<= (others => '0');
```

```
                    new_cmd <= '0';
                    new_param_low <= '0';
                    new_param_high <= '0';
                    interrupt_ack <= '0';
                    if(status /= NOSTAT) then
                            stat_out <= '1';
                    end if;
                    ack_n <= '0';
                    bus_state <= ack_state;
            -- hw writes newtask reg out
            -- set new_cmd low, in case it was set high right
            -- before interrupt -> no cmd done, no status sent
            elsif(addr = addr_reg6 and write_n = '1') then
                    dout(BITTASKS-1 downto 0) <= newtask;
                    dout(REG_LENGTH-1 downto
                            BITTASKS) <= (others => '0');
                    task_out <= '1';
                    new_cmd <= '0';
                    ack_n <= '0';
                    bus_state <= ack_state;
            -- hw writes curTime low out
            elsif(addr = addr_reg7 and write_n = '1') then
                    dout <= curTime_low;
                    ack_n <= '0';
                    bus_state <= ack_state;
            -- hw writes curTime high out
            elsif(addr = addr_reg8 and write_n = '1') then
                    dout <= curTime_high;
                    ack_n <= '0';
                    bus_state <= ack_state;
            -- error state
            else
                    ack_n <= '0';
                    bus_state <= ack_state;
            end if;
    else
            ack_n <= '1';
            bus_state <= wait_state;
    end if;

when ack_state =>
    if(cs_n = '1') then
            ack_n <= '1';
            stat_out <= '0';
            task_out <= '0';
            interrupt_ack <= '0';
```

```
                        bus_state <= wait_state;
                else
                        ack_n <= '0';
                        bus_state <= ack_state;
                end if;

            when others =>
                -- error state
                bus_state <= wait_state;
        end case;
    end if;
end process BISM;

-- Kernel Interface State Machine
KISM : process(clk, reset_n)
begin
    if(reset_n='0') then
        state <= i0;
        interrupt <= '0';
        status <= NOSTAT;
        newtask <= (others => '0');
        curTime_high <= (others => '0');
        curTime_low <= (others => '0');
        barrier <= '0';
        barrierNew <= '0';
        create <= '0';
        findtask <= '0';
        delayuntil <= '0';
        setfreq <= '0';
        freq <= (others => '0');
        fps <= (others => '0');
        upe <= (others => '0');
        ufe <= (others => '0');
        fpe <= (others => '0');
        ufpx <= (others => '0');
        upxe <= (others => '0');
        ufxe <= (others => '0');
        fpx <= (others => '0');
        es <= (others => '0');
        ueb <= (others => '0');
        uee <= (others => '0');
        uex <= (others => '0');
        ex <= (others => '0');
        Tcpu <= (others => '0');
        Pcpu <= (others => '0');
        DUntil <= (others => '0');
        TaskID <= (others => '0');
        TaskPrio <= (others => '0');
        CeilPrio <= (others => '0');
        POId <= (others => '0');
elsif(clk'event and clk='1') then
        case state is
                when i0 =>
                        -- new cmd, no interrupt
                        status <= NOSTAT;
                        if(new_cmd='1' and Tcpu=NTcpu) then
                                interrupt <= '0';
                                state <= i3;
                        -- interrupt occurs - don't care if new_cmd or not
                        -- sw just needs to know if it sent one, and got
                        -- interrupt before CMDDONE
                        elsif(Tcpu/=NTcpu) then
                                POID <= (others => '0');
                                interrupt <= '1';
                                state <= i1;
                        else
                                interrupt <= '0';
                                state <= i0;
                        end if;
                        -- if Prio changed due to Ceil Prio change, but same task
                        if(Pcpu/=NPcpu) then
                                Pcpu <= NPcpu;
                        end if;
                when i1 =>
                        if(interrupt_ack='1') then
                                interrupt <= '0';
                                -- sw check for newtask after it acks
                                newtask <= NTcpu;
                                Tcpu <= NTcpu;
                                Pcpu <= NPcpu;
                                state <= i2;
                        else
                                interrupt <= '1';
                                state <= i1;
                        end if;
                when i2 =>
                        if(task_out='1') then
                                state <= i0;
                        else
                                state <= i2;
                        end if;
                when i3 =>
        if(new_param_low='1') then
                if(cmd=DELAYUNTILi) then
                        state <= i5;
                else
                        state <= i4;
                end if;
        -- these don't have params
        elsif(cmd=FINDTASKi or cmd=GETTIMEi) then
                state <= i4;
        else
                state <= i3;
        end if;
when i4 =>
        if(cmd=CREATEi) then
                create <= '1';
                TaskPrio <= param_low(BITPRIO-1 downto 0);
                TaskID <= param_low(BITTASKS+BITPRIO-1
                        downto BITPRIO);
                state <= i6;
        elsif(cmd=FINDTASKi) then
                findtask <= '1';
                state <= i6;
        elsif(cmd=DELAYUNTILi) then
                delayuntil <= '1';
                -- **this assumes MAXTIME = 2*paramreg
                DUntil(PARAMREG-1 downto 0) <= param_low;
                DUntil(MAXTIME-1 downto PARAMREG) <=
                        param_high;
                state <= i6;
        -- for all PO cmds, set bit corresponding to POId high
        elsif(cmd=FPSi) then
                CeilPrio <= param_low(BITPO+BITPRIO-1
                        downto BITPO);
                POId <= param_low(BITPO-1 downto 0);
                fps(conv_integer(param_low(BITPO-1 downto 0)))
                        <= '1';
                state <= i6;
        elsif(cmd=UPEi) then
                POId <= param_low(BITPO-1 downto 0);
                upe(conv_integer(param_low(BITPO-1 downto 0)))
                        <= '1';
                state <= i6;
        elsif(cmd=UFEi) then
                POId <= param_low(BITPO-1 downto 0);
                ufe(conv_integer(param_low(BITPO-1 downto 0)))
                        <= '1';
        state <= i6;
elsif(cmd=FPEi) then
        POId <= param_low(BITPO-1 downto 0);
        fpe(conv_integer(param_low(BITPO-1 downto 0)))
                <= '1';
        state <= i6;
elsif(cmd=UFPXi) then
        POId <= param_low(BITPO-1 downto 0);
        ufpx(conv_integer(param_low(BITPO-1 downto
                0))) <= '1';
        state <= i6;
elsif(cmd=UPXEi) then
        POId <= param_low(BITPO-1 downto 0);
        upxe(conv_integer(param_low(BITPO-1 downto
                0))) <= '1';
        state <= i6;
elsif(cmd=UFXEi) then
        POId <= param_low(BITPO-1 downto 0);
        ufxe(conv_integer(param_low(BITPO-1 downto
                0))) <= '1';
        state <= i6;
elsif(cmd=FPXi) then
        POId <= param_low(BITPO-1 downto 0);
        fpx(conv_integer(param_low(BITPO-1 downto 0)))
                <= '1';
        state <= i6;
elsif(cmd=ESi) then
        CeilPrio <= param_low(BITPO+BITPRIO-1
                downto BITPO);
        POId <= param_low(BITPO-1 downto 0);
        es(conv_integer(param_low(BITPO-1 downto 0)))
                <= '1';
        state <= i6;
elsif(cmd=UEBi) then
        POId <= param_low(BITPO-1 downto 0);
        ueb(conv_integer(param_low(BITPO-1 downto 0)))
                <= '1';
        state <= i6;
elsif(cmd=UEEi) then
        POId <= param_low(BITPO-1 downto 0);
        uee(conv_integer(param_low(BITPO-1 downto 0)))
                <= '1';
        state <= i6;
elsif(cmd=UEXi) then
        POId <= param_low(BITPO-1 downto 0);
        uex(conv_integer(param_low(BITPO-1 downto 0)))
```

```
                <= '1';
            state <= i6;
        elsif(cmd=EXi) then
            POId <= param_low(BITPO-1 downto 0);
            ex(conv_integer(param_low(BITPO-1 downto 0)))
                <= '1';
            state <= i6;
        elsif(cmd=SETFREQi) then
            setfreq <= '1';
            freq <= param_low(FREQREG-1 downto 0);
            state <= i6;
        elsif(cmd=GETTIMEi) then
            -- again, this assumes sizes work right...
            curTime_low <= sTime(PARAMREG-1 downto 0);
            curTime_high <= sTime(MAXTIME-1 downto
                PARAMREG);
            status <= CMDDONE;
            state <= i7;
        else
            state <= i7;
            status <= BADCMD;
        end if;
    when i5 =>
        if(new_param_high='1') then
            state <= i4;
        else
            state <= i5;
        end if;
    when i6 =>
        if(barrierGet='1') then
            status <= GETBARR;
            state <= i8;
        elsif(kernel_status='1') then
            barrierNew <= '0';
            create <= '0';
            findtask <= '0';
            delayuntil <= '0';
            setfreq <= '0';
            freq <= (others => '0');
            fps(conv_integer(POId)) <= '0';
            upe(conv_integer(POId)) <= '0';
            ufe(conv_integer(POId)) <= '0';
            fpe(conv_integer(POId)) <= '0';
            ufpx(conv_integer(POId)) <= '0';
            upxe(conv_integer(POId)) <= '0';
            ufxe(conv_integer(POId)) <= '0';
```

```
                                        fpx(conv_integer(POId)) <= '0';
                                        es(conv_integer(POId)) <= '0';
                                        ueb(conv_integer(POId)) <= '0';
                                        uee(conv_integer(POId)) <= '0';
                                        uex(conv_integer(POId)) <= '0';
                                        ex(conv_integer(POId)) <= '0';
                                        TaskID <= (others => '0');
                                        TaskPrio <= (others => '0');
                                        DUntil <= (others => '0');
                                        CeilPrio <= (others => '0');
                                        status <= CMDDONE;
                                        state <= i7;
                                else
                                        state <= i6;
                                end if;
                        when i7 =>
                                if(stat_out='1') then
                                        state <= i0;
                                else
                                        state <= i7;
                                end if;
                        when i8 =>        -- wait for barrier stat to be read
                                if(stat_out='1') then
                                        state <= i9;
                                else
                                        state <= i8;
                                end if;
                        when i9 =>
                                -- got barrier
                                if(new_param_low='1') then
                                        barrier <= param_low(conv_integer(POId));
                                        barrierNew <= '1';
                                        status <= NOSTAT;
                                        state <= i10;
                                else
                                        state <= i9;
                                end if;
                        when i10 =>
                                -- let BarrierGet be reset so barr req isn't made again
                                state <= i6;
                        when others =>
                                state <= i0;
                end case;
        end if;
    end process KISM;
end Behavioral;
```

A.2 Main Kernel

```
--------------------------------------------------------------------------------------------------------------------------
-- NAME: kernel_internal.vhd
--
-- INPUTS:    clk          FPGA clock
--            reset_n      FPGA reset, active low
--            DelayUntil   signal specifyng delay until command has been made
--            SetFreq      signal specifying a set frequency command has been made
--            Freq         the frequency to set the clock at
--                         (only has data when SetFreq==1)
--            DUntil       the time to which a task will delay until
--                         (only has data when DelayUntil==1)
--            Tcpu         the ID of the task currently running
--            Pcpu         the priority of the task currently running
--            CeilPrio     the ceiling priority of the PO
--                         (only has data when FPs(POId)==1 or Es(POId)==1)
--            FPs          array of signals specifying FPs commands for POs
--            Upe          array of signals specifying Upe commands for POs
--            Ufe          array of signals specifying Ufe commands for POs
--            Ufpx         array of signals specifying Ufpx commands for POs
--            UPxe         array of signals specifying UPxe commands for POs
--            UFxe         array of signals specifying UFxe commands for POs
--            Barrier      barrier value of current PO
--                         (only valid if BarrierNew==1)
--            BarrierNew   signal indicating if Barrier holds a current value
--            FPe          array of signals specifying FPe commands for POs
--            FPx          array of signals specifying FPx commands for POs
--            Es           array of signals specifying Es commands for POs
--            UEb          array of signals specifying UEb commands for POs
--            UEe          array of signals specifying UEe commands for POs
--            UEx          array of signals specifying UEx commands for POs
--            Ex           array of signals specifying Ex commands for POs
--            Create       signal specifying create command has been made
--            FindTask     signal specifying find task command has been made
--            POId         ID of PO being called
--                         (only valid when a task is calling a PO)
--            TaskId       ID of task being created
--                         (only has data when Create==1)
--            TaskPrio     priority of task being created
--                         (only has data when Create==1)
--
-- OUTPUTS: BarrierGet     signal to request the PO's barrier value from the PPC
--            NPcpu        priority of the task that should now run on the PPC
--            NTcpu        ID of the task that should now run on the PPC
--            Status       signal indicating command has been completed
--            sTime        value of VCounter
--
```

```
-- DESCRIPTION: connects all the kernel components except interface routines.  Evaluates
--        channels that multiple components need, and determines kernel status.  Passes up to
--        interface new task to run, or a barrier request, and the current VCounter time
----------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.variables.ALL;

entity kernel_internal is
   Port ( clk          : in std_logic;
          reset_n      : in std_logic;
          DelayUntil   : in std_logic;
          SetFreq      : in std_logic;
          Freq         : in std_logic_vector(FREQREG-1 downto 0);
          DUntil       : in std_logic_vector (MAXDUNTIL-1 downto 0);
          Tcpu         : in std_logic_vector(BITTASKS-1 downto 0);
          Pcpu         : in std_logic_vector(BITPRIO-1 downto 0);
          CeilPrio     : in std_logic_vector(BITPRIO-1 downto 0);
          FPs          : in std_logic_vector(NUMPO-1 downto 0);
          Upe          : in std_logic_vector(NUMPO-1 downto 0);
          Ufe          : in std_logic_vector(NUMPO-1 downto 0);
          Ufpx         : in std_logic_vector(NUMPO-1 downto 0);
          UPxe         : in std_logic_vector(NUMPO-1 downto 0);
          UFxe         : in std_logic_vector(NUMPO-1 downto 0);
          Barrier      : in std_logic;
          BarrierNew   : in std_logic;
          FPe          : in std_logic_vector(NUMPO-1 downto 0);
          FPx          : in std_logic_vector(NUMPO-1 downto 0);
          Es           : in std_logic_vector(NUMPO-1 downto 0);
          UEb          : in std_logic_vector(NUMPO-1 downto 0);
          UEe          : in std_logic_vector(NUMPO-1 downto 0);
          UEx          : in std_logic_vector(NUMPO-1 downto 0);
          Ex           : in std_logic_vector(NUMPO-1 downto 0);
          Create       : in std_logic;
          FindTask     : in std_logic;
          POId         : in std_logic_vector(BITPO-1 downto 0);
          TaskId       : in std_logic_vector(BITTASKS-1 downto 0);
          TaskPrio     : in std_logic_vector(BITPRIO-1 downto 0);
          BarrierGet   : out std_logic;
          NPcpu        : out std_logic_vector(BITPRIO-1 downto 0);
          NTcpu        : out std_logic_vector(BITTASKS-1 downto 0);
          Status       : out std_logic;
          sTime        : out std_logic_vector(MAXTIME-1 downto 0));
```

```
end kernel_internal;

architecture Behavioral of kernel_internal is
        signal NPcpu_fromPO         : std_logic_vector(BITPRIO-1 downto 0);
        signal NPcpu_fromPO_all     : all_NPcpu;
        signal NPcpu_fromRQ         : std_logic_vector(BITPRIO-1 downto 0);
        signal NTcpu_fromRQ         : std_logic_vector(BITTASKS-1 downto 0);
        signal QE                   : std_logic; -- channel QE
        signal system_time          : std_logic_vector(MAXTIME-1 downto 0);
        signal SuspendTask          : std_logic; -- channel Suspend
        signal Suspend_fromPO       : std_logic_vector(NUMPO-1 downto 0);
        signal DQA_State            : dqa;
        signal DQO_State            : dqo;
        signal AwakeTask            : std_logic_vector(BITTASKS-1 downto 0); -- PE
        signal RQState              : rqs;
        signal FindNew              : std_logic; -- sig to readyqueue
        signal FindNew_all          : std_logic_vector(NUMPO-1 downto 0);
        signal PO_en                : std_logic;
        signal PO_en_all            : std_logic_vector(NUMPO-1 downto 0);
        signal PO_we                : std_logic;
        signal PO_we_all            : std_logic_vector(NUMPO-1 downto 0);
        signal PO_addr              : std_logic_vector(BITTASKS-1 downto 0);
        signal PO_addr_all          : all_RQaddr;
        signal PO_assign            : std_logic_vector(RQDATASIZE-1 downto 0);
        signal PO_assign_all        : all_RQassign;
        signal PO_Status            : std_logic_vector(NUMPO-1 downto 0);
        signal RQ_Status            : std_logic;
        signal timer_Status         : std_logic;
        signal BarrierReq           : std_logic_vector(NUMPO-1 downto 0);
        signal DQ_status            : std_logic;

        -- all of the kernel components
        component arbitrate_cpu
                port (  NPcpu_po    : in std_logic_vector(BITPRIO-1 downto 0);
                        Tcpu        : in std_logic_vector(BITTASKS-1 downto 0);
                        NPcpu_rq    : in std_logic_vector(BITPRIO-1 downto 0);
                        NTcpu_rq    : in std_logic_vector(BITTASKS-1 downto 0);
                        NPcpu       : out std_logic_vector(BITPRIO-1 downto 0);
                        NTcpu       : out std_logic_vector(BITTASKS-1 downto 0));
        end component;

        component dq_state_and_ram
                port (  clk         : in std_logic;
                        reset_n     : in std_logic;
                        delayuntil  : in std_logic;
                        param_reg   : in std_logic_vector (MAXDUNTIL-1 downto 0);
                qe              : in std_logic;
                foundnext       : in std_logic;
                stime           : in std_logic_vector(MAXTIME-1 downto 0);
                suspend         : in std_logic;
                tcpu            : in std_logic_vector(BITTASKS-1 downto 0);
                dqackstate      : out dqa;
                dqorgstate      : out dqo;
                pe              : out std_logic_vector(BITTASKS-1 downto 0);
                status          : out std_logic);
end component;

component multiPO
        port (  clk             : in std_logic;
                reset_n         : in std_logic;
                Pcpu            : in std_logic_vector(BITPRIO-1 downto 0);
                Tcpu            : in std_logic_vector(BITTASKS-1 downto 0);
                CeilPrio        : in std_logic_vector(BITPRIO-1 downto 0);
                FPs             : in std_logic_vector(NUMPO-1 downto 0);
                Upe             : in std_logic_vector(NUMPO-1 downto 0);
                Ufe             : in std_logic_vector(NUMPO-1 downto 0);
                Ufpx            : in std_logic_vector(NUMPO-1 downto 0);
                UPxe            : in std_logic_vector(NUMPO-1 downto 0);
                UFxe            : in std_logic_vector(NUMPO-1 downto 0);
                Barrier         : in std_logic;
                BarrierNew      : in std_logic;
                FPe             : in std_logic_vector(NUMPO-1 downto 0);
                FPx             : in std_logic_vector(NUMPO-1 downto 0);
                Es              : in std_logic_vector(NUMPO-1 downto 0);
                UEb             : in std_logic_vector(NUMPO-1 downto 0);
                UEe             : in std_logic_vector(NUMPO-1 downto 0);
                UEx             : in std_logic_vector(NUMPO-1 downto 0);
                Ex              : in std_logic_vector(NUMPO-1 downto 0);
                RQState         : in rqs;
                BarrierReq      : out std_logic_vector(NUMPO-1 downto 0);
                Suspend         : out std_logic_vector(NUMPO-1 downto 0);
                FindNew         : out std_logic_vector(NUMPO-1 downto 0);
                NPcpu           : out all_NPcpu;
                RQ_en           : out std_logic_vector(NUMPO-1 downto 0);
                RQ_we           : out std_logic_vector(NUMPO-1 downto 0);
                RQ_addr         : out all_RQaddr;
                RQ_assign       : out all_RQassign;
                PO_Status       : out std_logic_vector(NUMPO-1 downto 0));
end component;

component rq_state_and_ram_and_arbit
        port (  clk             : in std_logic;
                    reset_n         : in std_logic;
                    create          : in std_logic;
                    PO_addr         : in std_logic_vector(BITTASKS-1 downto 0);
                    PO_assign       : in std_logic_vector(RQDATASIZE-1 downto 0);
                    PO_en           : in std_logic;
                    PO_we           : in std_logic;
                    find_new        : in std_logic;
                    pcpu            : in std_logic_vector(BITPRIO-1 downto 0);
                    pe              : in std_logic_vector(BITTASKS-1 downto 0);
                    qe              : in std_logic;
                    suspend         : in std_logic;
                    task_id         : in std_logic_vector(BITTASKS-1 downto 0);
                    task_prio       : in std_logic_vector(BITPRIO-1 downto 0);
                    tcpu            : in std_logic_vector(BITTASKS-1 downto 0);
                    npcpu           : out std_logic_vector(BITPRIO-1 downto 0);
                    ntcpu           : out std_logic_vector(BITTASKS-1 downto 0);
                    rqstate         : out rqs;
                    status          : out std_logic);
        end component;

        component timer
                port (  clk             : in std_logic;
                        reset_n         : in std_logic;
                        setfreq         : in std_logic;
                        freq            : in std_logic_vector(FREQREG-1 downto 0);
                        status          : out std_logic;
                        VCounter        : out std_logic_vector(MAXTIME-1 downto 0));
                end component;

begin
        -- channels
        QE <= '1' when (RQState=r0 and (DQO_State=o2 or DQO_State=o3)) else '0';
        -- the or'd things CAN'T happen at same time
        SuspendTask <= '1' when (Suspend_fromPO(conv_integer(POId))='1'
                                        or (DQA_State=a2 and RQState=r0)) else '0';

        -- status
        Status <= PO_Status(conv_integer(POId)) or RQ_Status or timer_Status or DQ_status;

        -- barrier request
        BarrierGet <= BarrierReq(conv_integer(POId));

        -- assigning sigs from multi POs
        NPcpu_fromPO <= NPcpu_fromPO_all(conv_integer(POId));
        FindNew <= FindNew_all(conv_integer(POId)) OR FindTask;
        PO_en <= PO_en_all(conv_integer(POId));
        PO_we <= PO_we_all(conv_integer(POId));
        PO_addr <= PO_addr_all(conv_integer(POId));
        PO_assign <= PO_assign_all(conv_integer(POId));

        -- pass time array up
        sTime <= system_time;

        arb_cpu: arbitrate_cpu
                port map (NPcpu_po=>NPcpu_fromPO, Tcpu=>Tcpu,
                        NPcpu_rq=>NPcpu_fromRQ, NTcpu_rq=>NTcpu_fromRQ,
                        NPcpu=>NPcpu, NTcpu=>NTcpu);

        dq_all: dq_state_and_ram
                port map (clk=>clk, reset_n=>reset_n, delayuntil=>DelayUntil,
                        param_reg=>DUntil, qe=>QE, foundnext=>RQ_Status,
                        stime=>system_time, suspend=>SuspendTask, tcpu=>Tcpu,
                        dqackstate=>DQA_State, dqorgstate=>DQO_State, pe=>AwakeTask,
                        status=>DQ_status);

        PO_all: multiPO
                port map (clk=>clk, reset_n=>reset_n, Pcpu=>Pcpu, Tcpu=>Tcpu,
                        CeilPrio=>CeilPrio, FPs=>FPs, Upe=>Upe, Ufe=>Ufe,
                        Ufpx=>Ufpx, UPxe=>UPxe, UFxe=>UFxe, Barrier=>Barrier,
                        BarrierNew=>BarrierNew, FPe=>FPe, FPx=>FPx, Es=>Es,
                        UEb=>UEb, UEe=>UEe, UEx=>UEx, Ex=>Ex,
                        RQState=>RQState, BarrierReq=>BarrierReq,
                        Suspend=>Suspend_fromPO, FindNew=>FindNew_all,
                        NPcpu=>NPcpu_fromPO_all, RQ_en=>PO_en_all, RQ_we=>PO_we_all,
                        RQ_addr=>PO_addr_all, RQ_assign=>PO_assign_all,
                        PO_Status=>PO_Status);


        RQ_all: rq_state_and_ram_and_arbit
                port map (clk=>clk, reset_n=>reset_n, create=>Create, PO_addr=>PO_addr,
                        PO_assign=>PO_assign, PO_en=>PO_en, PO_we=>PO_we,
                        find_new=>FindNew, pcpu=>Pcpu, pe=>AwakeTask, qe=>QE,
                        suspend=>SuspendTask, task_id=>TaskId, task_prio=>TaskPrio,
                        tcpu=>Tcpu, npcpu=>Npcpu_fromRQ,
                        ntcpu=>NTcpu_fromRQ, rqstate=>RQState, status=>RQ_Status);

        timer_routine: timer
                port map(clk=>clk, reset_n=>reset_n, setfreq=>SetFreq, freq=>Freq,
                        status=>timer_Status, VCounter=>system_time);

end Behavioral;
```

A.3 Ready Queue

```
------------------------------------------------------------------------------------------------
-- NAME: rq_state_and_ram_and_arbit.vhd
--
-- INPUTS:     clk            FPGA clock
--             reset_n        FPGA reset, active low
--             create         signal specifying create command has been made
--             PO_addr        address location in the RQ_RAM a PO wants to access
--             PO_assign      data a PO wants to put in the RQ_RAM
--             PO_en          enable signal for the RQ_RAM, activated by a PO
--             PO_we          write enable signal for the RQ_RAM, activated by a PO
--             find_new       signal specifying find new command has been made
--             pcpu           priority of task running on PPC
--             pe             from delay queue, ID of a task that has awoken from delay
--                            (only has valid data when qe==1)
--             qe             channel indicating a task has awoken from delay
--             suspend        signal indicating the current task has suspended
--             task_id        ID of task being created
--                            (only has data when create==1)
--             task_prio      priority of task being created
--                            (only has data when create==1)
--             tcpu           ID of currently running task
--
-- OUTPUTS: npcpu          priority of task that should now run on PPC
--             ntcpu          ID of task that should now run on PPC
--             rqstate        state of the RQSM
--             status         status of this component
--
-- DESCRIPTION:  this combines the rq state machine, the rq ram, and the ram arbitrator
------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.variables.ALL;

entity rq_state_and_ram_and_arbit is
  Port ( clk             : in std_logic;
         reset_n         : in std_logic;
         create          : in std_logic;
         PO_addr         : in std_logic_vector(BITTASKS-1 downto 0);
         PO_assign       : in std_logic_vector(RQDATASIZE-1 downto 0);
         PO_en           : in std_logic;
         PO_we           : in std_logic;
         find_new        : in std_logic;
```

```
        pcpu            : in std_logic_vector(BITPRIO-1 downto 0);
        pe              : in std_logic_vector(BITTASKS-1 downto 0);
        qe              : in std_logic;
        suspend         : in std_logic;
        task_id         : in std_logic_vector(BITTASKS-1 downto 0);
        task_prio       : in std_logic_vector(BITPRIO-1 downto 0);
        tcpu            : in std_logic_vector(BITTASKS-1 downto 0);
        npcpu           : out std_logic_vector(BITPRIO-1 downto 0);
        ntcpu           : out std_logic_vector(BITTASKS-1 downto 0);
        rqstate         : out rqs;
        status          : out std_logic);
end rq_state_and_ram_and_arbit;

architecture Behavioral of rq_state_and_ram_and_arbit is
        signal addr             : std_logic_vector (BITTASKS-1 downto 0);
        signal en               : std_logic;
        signal ram_assign       : std_logic_vector (RQDATASIZE-1 downto 0);
        signal ram_out          : std_logic_vector (RQDATASIZE-1 downto 0);
        signal read_ack         : std_logic;
        signal read_data        : std_logic_vector (RQDATASIZE-1 downto 0);
        signal rq_addr          : std_logic_vector (BITTASKS-1 downto 0);
        signal rq_assign        : std_logic_vector (RQDATASIZE-1 downto 0);
        signal rq_en            : std_logic;
        signal rq_we            : std_logic;
        signal we               : std_logic;

        component rq_state
                port (  clk             : in std_logic;
                        reset_n         : in std_logic;
                        create          : in STD_LOGIC;
                        qe              : in std_logic;
                        suspend         : in std_logic;
                        findnew         : in std_logic;
                        rq_ack          : in std_logic;
                        taskid          : in std_logic_vector(BITTASKS-1 downto 0);
                        taskprio        : in std_logic_vector(BITPRIO-1 downto 0);
                        pe              : in std_logic_vector(BITTASKS-1 downto 0);
                        tcpu            : in std_logic_vector(BITTASKS-1 downto 0);
                        pcpu            : in std_logic_vector(BITPRIO-1 downto 0);
                        rq_read         : in std_logic_vector(RQDATASIZE-1 downto 0);
                        rq_en           : out std_logic;
                        rq_we           : out std_logic;
                        status          : out std_logic;
                        rq_addr         : out std_logic_vector(BITTASKS-1 downto 0);
                        rq_assign       : out std_logic_vector(RQDATASIZE-1 downto 0);
                        ntcpu           : out std_logic_vector(BITTASKS-1 downto 0);
                npcpu           : out std_logic_vector(BITPRIO-1 downto 0);
                rqstate         : out rqs);
    end component;

    component rq_ram
        port (  clk     : in std_logic;
                en      : in std_logic;
                we      : in std_logic;
                addr    : in std_logic_vector(BITTASKS-1 downto 0);
                din     : in std_logic_vector(RQDATASIZE-1 downto 0);
                dout    : out std_logic_vector(RQDATASIZE-1 downto 0));
    end component;

    component arbitrate_rq_ram
        port (  clk             : in std_logic;
                reset_n         : in std_logic;
                rq_en           : in std_logic;
                rq_we           : in std_logic;
                PO_en           : in std_logic;
                PO_we           : in std_logic;
                rq_addr         : in std_logic_vector(BITTASKS-1 downto 0);
                rq_assign       : in std_logic_vector(RQDATASIZE-1 downto 0);
                PO_addr         : in std_logic_vector(BITTASKS-1 downto 0);
                PO_assign       : in std_logic_vector(RQDATASIZE-1 downto 0);
                ram_read        : in std_logic_vector(RQDATASIZE-1 downto 0);
                ram_en          : out std_logic;
                ram_we          : out std_logic;
                read_ack        : out std_logic;
                ram_addr        : out std_logic_vector(BITTASKS-1 downto 0);
                ram_assign      : out std_logic_vector(RQDATASIZE-1 downto 0);
                read_data       : out std_logic_vector(RQDATASIZE-1 downto 0));
    end component;

begin
    rq_state_routine : rq_state
        port map (clk=>clk, reset_n=>reset_n, create=>create, qe=>qe,
                suspend=>suspend, findnew=>find_new, rq_ack=>read_ack,
                taskid=>task_id, taskprio=>task_prio, pe=>pe, tcpu=>tcpu, pcpu=>pcpu,
                rq_read=>read_data, rq_en=>rq_en, rq_we=>rq_we, status=>status,
                rq_addr=>rq_addr, rq_assign=>rq_assign, ntcpu=>ntcpu, npcpu=>npcpu,
                rqstate=>rqstate);

    rq_ram_routine : rq_ram
        port map (clk=>clk, en=>en, we=>we, addr=>addr, din=>ram_assign,
                dout=>ram_out);
```

```
          rq_arbitrator : arbitrate_rq_ram
                    port map (clk=>clk, reset_n=>reset_n, rq_en=>rq_en, rq_we=>rq_we,
                              po_en=>po_en, po_we=>po_we, rq_addr=>rq_addr,
                              rq_assign=>rq_assign, po_addr=>po_addr, po_assign=>po_assign,
                              ram_read=>ram_out, ram_en=>en, ram_we=>we, read_ack=>read_ack,
                              ram_addr=>addr, ram_assign=>ram_assign, read_data=>read_data);

end Behavioral;

---------------------------------------------------------------------------------------------------------
-- NAME: rq_ram.vhd
--
-- INPUTS:    clk     FPGA clock
--            en      RAM enable
--            we      RAM write enable – write high, read low
--            addr    location in RAM being accessed
--            din     data to put in RAM addr (for write)
--
-- OUTPUT:    dout    data in RAM addr (for read)
--
-- DESCRIPTION:  this simulates RAM to hold the ready queue data.  It size is defined by the
--        variables NUMTASKS and RQDATASIZE, which are both defined in myvariables.vhd
---------------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.variables.ALL;

entity RQ_RAM is
   Port (clk     : in std_logic;
         en      : in std_logic;
         we      : in std_logic;
         addr    : in std_logic_vector(BITTASKS-1 downto 0);
         din     : in std_logic_vector(RQDATASIZE-1 downto 0);
         dout    : out std_logic_vector(RQDATASIZE-1 downto 0));
end RQ_RAM;

architecture Behavioral of RQ_RAM is
         type ram_type is array (NUMTASKS-1 downto 0) of std_logic_vector(RQDATASIZE-1
                  downto 0);
         signal ram_array : ram_type;
```

```
begin
        process(clk)
        begin
                if(clk'event and clk='1') then
                        if(en='1') then
                                if(we='1') then
                                        ram_array(conv_integer(addr))<=din;
                                        dout <= din;
                                else
                                        dout <= ram_array(conv_integer(addr));
                                end if;
                        end if;
                end if;
        end process;
end Behavioral;

--------------------------------------------------------------------------------------------------------
-- NAME: rq_state
--
-- INPUTS:    clk          FPGA clock
--            reset_n      FPGA reset, active low
--            create       signal specifying create command has been made
--            TaskID       ID of task being created
--                         (only has valid data when create==1)
--            TaskPrio     priority of task being created
--                         (only has valid data when create==1)
--            PE           from delay queue, ID of a task that has awoken from delay
--                         (only has valid data when QE==1)
--            Tcpu         ID of currently running task
--            Pcpu         priority of currently running task
--            QE           channel indicating that a task has awoken
--            Suspend      signal indicating current task has suspended
--            FindNew      signal indicating a new task must be found to run
--            RQ_read      data from the RQ_RAM
--            RQ_ack       signal from RQ_RAM arbitrator indicating RAM access occurred
--
-- OUTPUTS: RQ_en          enable signal for accessing RQ_RAM
--            RQ_we        write enable signal for accessing RQ_RAM
--            RQ_addr      location in RQ_RAM to access
--            RQ_assign    data to write to RQ_addr location in RQ_RAM
--            Status       status of this component
--            NTcpu        ID of task that should now run on PPC
--            NPcpu        priority of task that should now run on PPC
--            RQstate      state of the RQSM
--
-- DESCRIPTION:  this is the state machine for the ready queue.  It handles creation of tasks,
--      and finds the task that should run after elaboration, after a task has suspended, or after a
--      task wakes from a delay
----------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.variables.ALL;

entity rq_state is
   Port ( clk           : in std_logic;
          reset_n       : in std_logic;
          create        : in std_logic;
          TaskID        : in std_logic_vector(BITTASKS-1 downto 0);
          TaskPrio      : in std_logic_vector(BITPRIO-1 downto 0);
          PE            : in std_logic_vector(BITTASKS-1 downto 0);
          Tcpu          : in std_logic_vector(BITTASKS-1 downto 0);
          Pcpu          : in std_logic_vector(BITPRIO-1 downto 0);
          QE            : in std_logic;
          Suspend       : in std_logic;
          FindNew       : in std_logic;
          RQ_read       : in std_logic_vector(RQDATASIZE-1 downto 0);
          RQ_ack        : in std_logic;
          RQ_en         : out std_logic;
          RQ_we         : out std_logic;
          RQ_addr       : out std_logic_vector(BITTASKS-1 downto 0);
          RQ_assign     : out std_logic_vector(RQDATASIZE-1 downto 0);
          Status        : out std_logic;
          NTcpu         : out std_logic_vector(BITTASKS-1 downto 0);
          NPcpu         : out std_logic_vector(BITPRIO-1 downto 0);
          RQstate       : out rqs);
end rq_state;

architecture Behavioral of rq_state is
          signal RQ                  : rqs;
          signal addr                : std_logic_vector(BITTASKS-1 downto 0);
          NPcpu_internal             : std_logic_vector(BITPRIO-1 downto 0);

begin
          NPcpu <= NPcpu_internal;

          process(clk, reset_n)
          variable tr       : std_logic_vector(BITTASKS-1 downto 0);
          variable ir       : std_logic_vector(BITTASKS-1 downto 0);
          variable pr       : std_logic_vector(BITPRIO-1 downto 0);
```

```
begin
        if(reset_n='0') then
                status <= '0';
                NTcpu <= (others => '0');
                NPcpu_internal <= (others => '0');
                RQ_en <= '1';
                RQ_we <= '1';
                addr <= (others => '0');
                RQ_assign <= (others => '0');
                tr := (others => '0');
                pr := (others => '0');
                ir := (others => '0');
                RQ <= reset_ram;
        elsif(clk'event and clk='1') then
                case RQ is
                        when reset_ram =>
                                if(addr < (NUMTASKS-1))    then
                                        -- RQ_assign has already been set to 0
                                        -- enable and write are set
                                        addr <= addr + 1;
                                        RQ <= reset_ram;
                                else
                                        RQ <= r0;
                                end if;
                        when r0 =>
                                if(create='1') then
                                        Status <= '1';
                                        RQ_en <= '1';
                                        RQ_we <= '1';
                                        addr <= TaskID;
                                        RQ_assign(RQDATASIZE-1 downto 1) <=
                                                TaskPrio;
                                        RQ_assign(0) <= '1';
                                        RQ <= r1;
                                elsif(QE='1') then
                                        -- A TASK HAS AWOKEN FROM DELAY
                                        -- MAKE IT VALID AGAIN
                                        Status <= '0';
                                        RQ_en <= '1';
                                        RQ_we <= '0';
                                        addr <= PE;
                                        RQ <= r2;
                                elsif(Suspend='1') then
                                        -- CURRENT TASK HAS BEEN SUSPENDED
                                        -- MAKE INVALID, SEARCH NEXT TO RUN
                                        Status <= '0';
```

```
                    RQ_en <= '1';
                    RQ_we <= '0';
                    addr <= Tcpu;
                    tr := (others => '0');
                    pr := (others => '0');
                    ir := conv_std_logic_vector(NUMTASKS-1,
                              BITTASKS);
                    RQ <= r3;
          elsif(FindNew='1') then
                    -- A TASK HAS COMPLETED
                    -- NEED TO FIND A NEW ONE
                    -- SAME ROUTINE AS SUSPEND, BUT NO
                    -- NEED TO CHANGE VALID BITS
                    Status <= '0';
                    RQ_en <= '0';
                    RQ_we <= '0';
                    tr := (others => '0');
                    pr := (others => '0');
                    ir := conv_std_logic_vector(NUMTASKS-1,
                              BITTASKS);
                    RQ <= r4;
          else
                    Status <= '0';
                    RQ_en <= '0';
                    RQ_we <= '0';
                    RQ <= r0;
          end if;
when r1 =>
          RQ_en <= '0';
          RQ_we <= '0';
          Status <= '0';
          RQ <= r0;
-- PATH FOR MAKING TASK VALID AGAIN
when r2 =>
          if(RQ_ack='1') then
                    RQ_en <= '1';
                    RQ_we <= '1';
                    -- ADDR IS STILL PE
                    RQ_assign(RQDATASIZE-1 downto 1) <=
                              RQ_read(RQDATASIZE-1 downto 1);
                    -- SETS VALID BIT BACK TO 1
                    RQ_assign(0) <= '1';
                    RQ <= r0;
                    if(RQ_read(RQDATASIZE-1 downto 1)>Pcpu
                       or RQ_read(RQDATASIZE-1 downto 1 >
                       NPcpu_internal) then
                    NPcpu_internal <=
                      RQ_read(RQDATASIZE-1 downto 1);
                    NTcpu <= addr;
                end if;
        else
                RQ_en <= '0';
                RQ_we <= '0';
                RQ <= r2;
        end if;
------------------------------------------------
-- PATH FOR MAKING A TASK INVALID (r4 and r5), AND
-- FINDING ONE WITH NEXT HIGHEST PRIO (r6 on)
when r3 =>
        if(RQ_ack='1') then
                RQ_en <= '1';
                RQ_we <= '1';
                -- ADDR IS STILL TCPU
                RQ_assign(RQDATASIZE-1 downto 1) <=
                        RQ_read(RQDATASIZE-1 downto 1);
                RQ_assign(0) <= '0'; -- SETS VALID BIT TO 0
                RQ <= r4;
        else
                RQ_en <= '0';
                RQ_we <= '0';
                RQ <= r3;
        end if;
when r4 =>
        if(ir>0) then
                RQ_en <= '1';
                RQ_we <= '0';
                addr <= ir;
                Status <= '0';
                RQ <= r5;
        else
                -- if no other tasks can run, run null
                RQ_en <= '0';
                RQ_we <= '0';
                NTcpu <= (others => '0');
                NPcpu_internal <= (others => '0');
                Status <= '1';
                RQ <= r8;
        end if;
when r5 =>
        if(RQ_ack='1') then
                RQ_en <= '0';
                RQ_we <= '0';
```

```
                    ir := ir-1;
                    if((RQ_read(RQDATASIZE-1 downto 1)>0) AND
                                (RQ_read(0)='1')) then
                        pr := RQ_read(RQDATASIZE-1 downto 1);
                        tr := addr;
                        RQ <= r6;
                    else
                        RQ <= r4;
                    end if;
                else
                    RQ_en <= '0';
                    RQ_we <= '0';
                    RQ <= r5;
                end if;
            when r6 =>
                RQ_we <= '0';
                if(ir>0) then
                    RQ_en <= '1';
                    addr <= ir;
                    Status <= '0';
                    RQ <= r7;
                else
                    RQ_en <= '0';
                    NTcpu <= tr;
                    NPcpu_internal <= pr;
                    Status <= '1';
                    RQ <= r8;
                end if;
            when r7 =>
                if(RQ_ack='1') then
                    RQ_en <= '0';
                    RQ_we <= '0';
                    ir := ir-1;
                    RQ <= r6;
                    if((RQ_read(RQDATASIZE-1 downto 1) > pr)
                                AND (RQ_read(0)='1')) then
                        pr := RQ_read(RQDATASIZE-1 downto 1);
                        -- ADDR IS STILL IR
                        tr := addr;
                    end if;
                else
                    RQ_en <= '0';
                    RQ_we <= '0';
                    RQ <= r7;
                end if;
            when r8 =>
```

```
                                        RQ_en <= '0';
                                        RQ_we <= '0';
                                        Status <= '0';
                                        RQ <= r0;
                                when others =>
                                        RQ <= r0;
                        end case;
                end if;
        end process;

        -- ASSIGN STATE BITS, RAM ADDRESS TO OUTPUT
        RQState <= RQ;
        RQ_addr <= addr;

end Behavioral;

-----------------------------------------------------------------------------------------------------------
-- NAME: arbitrate_rq_ram.vhd
--
-- INPUTS:      clk             FPGA clock
--              reset_n         FPGA reset, active low
--              RQ_en           RAM enable from RQSM
--              RQ_we           RAM write enable from RQSM
--              RQ_addr         RAM address RQSM wants to access
--              RQ_assign       data RQSM wants to assign to RAM
--              PO_en           RAM enable from a PO
--              PO_we           RAM write enable from a PO
--              PO_addr         RAM address a PO wants to access
--              PO_assign       data a PO wants to assign to RAM
--              ram_read        data output from the RQ_RAM
--
-- OUTPUTS: ram_en              enable sent to RAM
--              ram_we          write enable sent to RAM
--              ram_addr        address sent to RAM
--              ram_assign      data to assign sent to RAM
--              read_data       data read from RAM to redirect to RQSM or PO
--              read_ack        acknowledge that RAM read has occurred
--
-- DESCRIPTION:  this arbitrates simultaneous access to the RQ_RAM, by the RQSM and a PO
--
-- NOTES:
--      conflicts will only occur between the QE branch of the readyqueue, and any one entry or
--      function/procedure process
--      assumes that create branch of readyqueue is ONLY called at initialization
--      cases:
--      1) readyqueue tries to read at r0, entry tries to write at E0
--              entry should write first, readyqueue will stall while waiting for read results; no
--              issues with access the immediate cycles after
--      2) readyqueue tries to read at r0, entry tries to write at E15
--              entry should write first, readyqueue will stall while waiting for read results; no
--              issues with access the immediate cycles after, because even though after entry
--              writes, control goes to proc, and immedaitely wants to write again, it must synch
--              on FindNew, which means it must wait for readyqueue to return to r0
--      3) readyqueue tries to read at r0, procedure tries to write at P0
--              procedure should write first, readyqueue will stall while waiting for read results;
--              no issues with access the immediate cycles after
--      4) readyqueue tries to write at r2, entry tries to write at E0
--              let entry write first, but doesn't matter; no issues with access the immediate cycles
--              after
--      5) readyqueue tries to write at r2, entry tries to write at E15
--              let entry write first, but doesn't matter; no issues
--      6) readyqueue tries to write at r2, procedure tries to write at P0
--              let proc write first, but doesn't matter; same issues as case 4 though

--      SUMMARY: only have to deal with two things happening at once, but nothing
--      happening the immediate cycle after - so we won't get a back up, we can
--      use a state machine with a branch for each double command
--      for ease, always have the entry/proc write first

--      have a seperate machine to handle reads - since we need to wait an extra
--      cycle for valid read, and send acknowledge, but a write could occur while
--      waiting (ie readyqueue sends read, waiting at r2 for ack, entry now writes)
--------------------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.variables.ALL;

entity arbitrate_rq_ram is
   Port ( clk           : in std_logic;
          reset_n       : in std_logic;
          RQ_en         : in std_logic;
          RQ_we         : in std_logic;
          RQ_addr       : in std_logic_vector(BITTASKS-1 downto 0);
          RQ_assign     : in std_logic_vector(RQDATASIZE-1 downto 0);
          PO_en         : in std_logic;
          PO_we         : in std_logic;
          PO_addr       : in std_logic_vector(BITTASKS-1 downto 0);
          PO_assign     : in std_logic_vector(RQDATASIZE-1 downto 0);
          ram_read      : in std_logic_vector(RQDATASIZE-1 downto 0);
```

```
		ram_en		: out std_logic;
		ram_we		: out std_logic;
		ram_addr	: out std_logic_vector(BITTASKS-1 downto 0);
		ram_assign	: out std_logic_vector(RQDATASIZE-1 downto 0);
		read_data	: out std_logic_vector(RQDATASIZE-1 downto 0);
		read_ack	: out std_logic);
end arbitrate_rq_ram;

architecture Behavioral of arbitrate_rq_ram is
	type state_type is (s0, s1, s2);
	signal state			: state_type;
	type read_states is (r0, r1);
	signal r			: read_states;
	signal read			: std_logic;
	signal temp_addr		: std_logic_vector(BITTASKS-1 downto 0);
	signal temp_assign		: std_logic_vector(RQDATASIZE-1 downto 0);

begin
	-- synch machines, on read
	reading : process(clk, reset_n)
	begin
		if(reset_n='0') then
			read_data <= (others => '0');
			read_ack <= '0';
			r <= r0;
		elsif(clk'event and clk='1') then
			case r is
				when r0 =>
					read_data <= (others => '0');
					read_ack <= '0';
					if(read='1') then
						r <= r1;
					else
						r <= r0;
					end if;
				when r1 =>
					read_data <= ram_read;
					read_ack <= '1';
					r <= r0;
				when others =>
					r <= r0;
			end case;
		end if;
	end process reading;

	arbitrate : process(clk, reset_n)
```

```
begin
        if(reset_n='0') then
                ram_en <= '0';
                ram_we <= '0';
                ram_addr <= (others => '0');
                ram_assign <= (others => '0');
                temp_addr <= (others => '0');
                temp_assign <= (others => '0');
                read <= '0';
                state <= s0;
        elsif(clk'event and clk='1') then
                case state is
                        when s0 =>
                                -- case: only 1 process (readyqueue) is trying to write
                                if(RQ_en='1' and RQ_we='1' and PO_en='0' and
                                                PO_we='0') then
                                        ram_en <= '1';
                                        ram_we <= '1';
                                        ram_addr <= RQ_addr;
                                        ram_assign <= RQ_assign;
                                        read <= '0';
                                        state <= s0;
                                -- case: only 1 process (entry/procedure) is trying to write
                                elsif(RQ_en='0' and RQ_we='0' and PO_en='1' and
                                                PO_we='1') then
                                        ram_en <= '1';
                                        ram_we <= '1';
                                        ram_addr <= PO_addr;
                                        ram_assign <= PO_assign;
                                        read <= '0';
                                        state <= s0;
                                -- case: readyqueue is trying to read - calls reading
                                elsif(RQ_en='1' and RQ_we='0' and PO_en='0' and
                                                PO_we='0') then
                                        ram_en <= '1';
                                        ram_we <= '0';
                                        ram_addr <= RQ_addr;
                                        read <= '1';
                                        state <= s0;
                                -- case: readyqueue tries read, entry/procedure tries to write
                                elsif(RQ_en='1' and RQ_we='0' and PO_en='1' and
                                                PO_we='1') then
                                        temp_addr <= RQ_addr;
                                        ram_en <='1';
                                        ram_we <= '1';
                                        ram_addr <= PO_addr;
                                        ram_assign <= PO_assign;
                                        read <= '0';
                                        state <= s1;
                                -- case: readyqueue tries write, entry/procedure tries write
                                elsif(RQ_en='1' and RQ_we='1' and PO_en='1' and
                                                PO_we='1') then
                                        temp_addr <= RQ_addr;
                                        temp_assign <= RQ_assign;
                                        ram_en <= '1';
                                        ram_we <= '1';
                                        ram_addr <= PO_addr;
                                        ram_assign <= PO_assign;
                                        read <= '0';
                                        state <= s2;
                                else
                                        ram_en <= '0';
                                        ram_we <= '0';
                                        ram_addr <= (others => '0');
                                        ram_assign <= (others => '0');
                                        read <= '0';
                                        state <= s0;
                                end if;
                        when s1 =>
                                ram_en <= '1';
                                ram_we <= '0';
                                ram_addr <= temp_addr;
                                read <= '1';
                                state <= s0;
                        when s2 =>
                                ram_en <= '1';
                                ram_we <= '1';
                                ram_addr <= temp_addr;
                                ram_assign <= temp_assign;
                                read <= '0';
                                state <= s0;
                        when others =>
                                state <= s0;
                end case;
        end if;
    end process arbitrate;

end Behavioral;
```

A.4 Delay Queue

```
------------------------------------------------------------------------------------------------
-- NAME: dq_state_and_ram.vhd
--
-- INPUTS:   clk          FPGA clock
--           reset_n      FPGA reset, active low
--           delayuntil   signal specifying a delay until command has been made
--           param_reg    the time to which the task will delay until
--                        (only valid when delayuntil==1)
--           qe           channel indicating a task has awoken from delay
--           foundnext    status signal from RQ, indicating next task to run has been found
--           stime        value of VCounter
--           suspend      signal indicating a task has been suspended
--           tcpu         ID of task running on PPC
--           dqackstate   state of DQASM
--           dqorgstate   state of DQOSM
--
-- OUTPUTS: pe            ID of task that has awoken from a delay
--           status       status of this component
--
-- DESCRIPTION:  This combines the delay queue state machine, and the associated RAM
------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.variables.ALL;

entity DQ_state_and_ram is
   Port ( clk           : in std_logic;
          reset_n       : in std_logic;
          delayuntil    : in std_logic;
          param_reg     : in std_logic_vector (MAXDUNTIL-1 downto 0);
          qe            : in std_logic;
          foundnext     : in std_logic;
          stime         : in std_logic_vector(MAXTIME-1 downto 0);
          suspend       : in std_logic;
          tcpu          : in std_logic_vector(BITTASKS-1 downto 0);
          dqackstate    : out dqa;
          dqorgstate    : out dqo;
          pe            : out std_logic_vector(BITTASKS-1 downto 0);
          status        : out std_logic);
end dq_state_and_ram;

architecture Behavioral of dq_state_and_ram is
```

```
    signal dq_addr          : std_logic_vector(BITTASKS-1 downto 0);
    signal dq_assign        : std_logic_vector(MAXDUNTIL-1 downto 0);
    signal dq_en            : std_logic;
    signal dq_read          : std_logic_vector(MAXDUNTIL-1 downto 0);
    signal dq_we            : std_logic;
    signal read_ack         : std_logic;

    component dq_state
            port (  clk             : in std_logic;
                    reset_n         : in std_logic;
                    delayuntil      : in std_logic;
                    suspend         : in std_logic;
                    qe              : in std_logic;
                    foundnext       : in std_logic;
                    dq_read_ack     : in std_logic;
                    stime           : in std_logic_vector(MAXTIME-1 downto 0);
                    tcpu            : in std_logic_vector(BITTASKS-1 downto 0);
                    param_reg       : in std_logic_vector(MAXDUNTIL-1 downto 0);
                    dq_read         : in std_logic_vector(MAXDUNTIL-1 downto 0);
                    dq_en           : out std_logic;
                    dq_we           : out std_logic;
                    dq_addr         : out std_logic_vector(BITTASKS-1 downto 0);
                    dq_assign       : out std_logic_vector(MAXDUNTIL-1 downto 0);
                    pe              : out std_logic_vector(BITTASKS-1 downto 0);
                    Status          : out std_logic;
                    dqackstate      : out dqa;
                    dqorgstate      : out dqo);
    end component;

    component dq_ram
            port (  clk             : in std_logic;
                    reset_n         : in std_logic;
                    en              : in std_logic;
                    we              : in std_logic;
                    addr            : in std_logic_vector(BITTASKS-1 downto 0);
                    din             : in std_logic_vector(MAXDUNTIL-1 downto 0);
                    read_ack        : out std_logic;
                    dout            : out std_logic_vector(MAXDUNTIL-1 downto 0));
    end component;

begin
    dq_state_routine : dq_state
            port map ( clk=>clk, reset_n=>reset_n, delayuntil=>delayuntil,
                    suspend=>suspend, qe=>qe, foundnext=>foundnext,
                    dq_read_ack=>read_ack, stime=>stime, tcpu=>tcpu,
                    param_reg=>param_reg, dq_read=>dq_read, dq_en=>dq_en,
                    dq_we=>dq_we, dq_addr=>dq_addr, dq_assign=>dq_assign, pe=>pe,
                    Status=>status, dqackstate=>dqackstate, dqorgstate=>dqorgstate);

        dq_ram_routine : dq_ram
                port map ( clk=>clk, reset_n=>reset_n, en=>dq_en, we=>dq_we, addr=>dq_addr,
                    din=>dq_assign, read_ack=>read_ack, dout=>dq_read);

end Behavioral;

-------------------------------------------------------------------------------------------------------------
-- NAME: dq_ram
--
-- INPUTS:   clk          FPGA clock
--           reset_n      FPGA reset, active low
--           en           enable signal for the RAM
--           we           write enable signal for the RAM, write high, read low
--           addr         location in RAM to access
--           din          data to put in addr on write
--
-- OUTPUTS: dout          data from addr on read
--           read_ack     notifies state machine accessing RAM that read data is available
--
-- DESCRIPTION: This simulates RAM to hold the delay queue data
-------------------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.variables.ALL;

entity DQ_RAM is
   Port ( clk          : in std_logic;
        reset_n        : in std_logic;
        en             : in std_logic;
        we             : in std_logic;
        addr           : in std_logic_vector(BITTASKS-1 downto 0);
        din            : in std_logic_vector(MAXDUNTIL-1 downto 0);
        dout           : out std_logic_vector(MAXDUNTIL-1 downto 0);
        read_ack       : out std_logic);
end DQ_RAM;

architecture Behavioral of DQ_RAM is
        type ram_type is array (NUMTASKS-1 downto 0) of std_logic_vector(MAXDUNTIL-1
                downto 0);
        signal ram_array : ram_type;
```

```
begin
        process(clk)
        begin
                if(clk'event and clk='1') then
                        if(en='1') then
                                if(we='1') then
                                        ram_array(conv_integer(addr))<=din;
                                        dout <= din;
                                else
                                        dout <= ram_array(conv_integer(addr));
                                end if;
                        end if;
                end if;
        end process;

        ackproc : process(clk, reset_n)
        begin
                if(reset_n='0') then
                        read_ack <= '0';
                elsif(clk'event and clk='1') then
                        if(en='1' and we='0') then
                                read_ack <= '1';
                        else
                                read_ack <= '0';
                        end if;
                end if;
        end process ackproc;
end Behavioral;

---------------------------------------------------------------------------------------------------
-- NAME: dq_state.vhd
--
-- INPUTS:    clk           FPGA clock
--            reset_n       FPGA reset, active low
--            sTime         value of VCounter
--            DelayUntil    signal specifying that a delay until command has been made
--            Tcpu          ID of running task
--            param_reg     time to which the task will delay until
--            Suspend       signal indicating that the current task is being suspended
--            QE            channel indicating that a task has awoken from a delay
--            FoundNext     signal indicating that a new task to run has been found
--            DQ_read       data read from the DQ_RAM
--            DQ_read_ack   signal indicating valid data is in DQ_read
--
-- OUTPUTS: DQ_en           enable for RAM
--              DQ_we           write enable for RAM
--              DQ_addr         address of RAM
--              DQ_assign       data to write to RAM
--              PE              ID of task that has awoken from delay
--              Status          status of this component
--              DQackState      state of DQASM
--              DQorgState      state of DQOSM
--
-- DESCRIPTION: the state machines to control the delay queue.  DQASM gets delay
--      information from the interface, and the DQOSM keeps track of tasks that have delayed
-------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.variables.ALL;

entity dq_state is
   Port ( clk           : in std_logic;
        reset_n         : in std_logic;
        sTime           : in std_logic_vector(MAXTIME-1 downto 0);
        DelayUntil      : in std_logic;
        Tcpu            : in std_logic_vector(BITTASKS-1 downto 0);
        param_reg       : in std_logic_vector(MAXDUNTIL-1 downto 0);
        Suspend         : in std_logic;
        QE              : in std_logic;
        FoundNext       : in std_logic;
        DQ_read         : in std_logic_vector(MAXDUNTIL-1 downto 0);
        DQ_read_ack     : in std_logic;
        DQ_en           : out std_logic;
        DQ_we           : out std_logic;
        DQ_addr         : out std_logic_vector(BITTASKS-1 downto 0);
        DQ_assign       : out std_logic_vector(MAXDUNTIL-1 downto 0);
        PE              : out std_logic_vector(BITTASKS-1 downto 0);
        Status          : out std_logic;
        DQackState      : out dqa;
        DQorgState      : out dqo);
end dq_state;

architecture Behavioral of dq_state is
        signal DQack            : dqa;
        signal DQorg            : dqo;
        signal PidNoDQ          : std_logic_vector(BITTASKS-1 downto 0);
        signal PidDQ            : std_logic_vector(BITTASKS-1 downto 0);
        signal DUntil           : std_logic_vector(MAXDUNTIL-1 downto 0);
        signal len              : std_logic_vector(BITTASKS-1 downto 0);
        signal t                : std_logic_vector(MAXTIME-1 downto 0);
        signal Q                : std_logic;
        signal addr             : std_logic_vector(BITTASKS-1 downto 0);

begin
        -- INTERNAL CHANNEL Q
        Q <= '1' when (DQack=a1 and DQorg=o0) else '0';

        -- THIS PROCESS IS ACTIVATED BY THE INTERFACE, WHEN A TASK
        -- SENDS A NEW DELAY UNTIL COMMAND
        ack_interface : process(clk, reset_n)
        begin
                if(reset_n='0') then
                        DQack <= a0;
                        PidNoDQ <= (others => '0');
                        DUntil <= (others => '0');
                        Status <= '0';
                elsif(clk'event and clk='1') then
                        case DQack is
                                when a0 =>
                                        -- CHECK IF RECEIVE NEW DELAY UNTIL
                                        if(DelayUntil='1' and param_reg > sTime) then
                                                PidNoDQ <= Tcpu;
                                                DUntil <= param_reg;
                                                Status <= '0';
                                                DQack <= a1;
                                        elsif(DelayUntil='1' and sTime >= param_reg) then
                                                Status <= '1';
                                                DQack <= a0;
                                        else
                                                Status <= '0';
                                                DQack <= a0;
                                        end if;
                                when a1 =>
                                        -- WAIT UNTIL WE SYNCH ON INTERNAL Q
                                        if(Q='1') then
                                                DQack <= a2;
                                        else
                                                DQack <= a1;
                                        end if;
                                when a2 =>
                                        -- WAIT UNTIL WE SYNCH ON CHANNEL SUSPEND
                                        if(Suspend='1') then
                                                DQack <= a3;
                                        else
                                        DQack <= a2;
                                end if;
                        when a3 =>
                                -- WAIT UNTIL READYQUEUE FINDS NEXT TASK
                                if(FoundNext='1') then
                                        DQack <= a0;
                                else
                                        DQack <= a3;
                                end if;
                        when others =>
                                DQack <= a0;
                end case;
        end if;
end process ack_interface;

-- THIS PROCESS ORGANIZES THE DELAY QUEUE: ADDS ELEMENTS TO RAM
-- CHECKS TO SEE IF DELAY TIME IS UP FOR A TASK
dq_organize : process(clk, reset_n)
variable i : std_logic_vector(BITTASKS-1 downto 0);
variable j : std_logic_vector(MAXDUNTIL-1 downto 0);
begin
        if(reset_n='0') then
                PE <= (others => '0');
                DQ_we <= '1';
                DQ_en <= '1';
                DQ_assign <= (others => '0');
                addr <= (others => '0');
                len <= (others => '0');
                t <= (others => '0');
                PidDQ <= (others => '0');
                i := (others => '0');
                j := (others => '0');
                DQorg <= reset_ramDQ;
        elsif(clk'event and clk='1') then
                case DQorg is
                        when reset_ramDQ =>
                                if(addr < (NUMTASKS-1)) then
                                        -- DQ_assign has already been set to 0
                                        addr <= addr+1;
                                        DQorg <= reset_ramDQ;
                                else
                                        DQ_en <= '0';
                                        DQ_we <= '0';
                                        DQorg <= o0;
                                end if;
                        when o0 =>
        if(Q='1' and len=0) then
                DQ_we <= '0';
                DQ_en <= '0';
                t <= DUntil;
                PidDQ <= PidNoDQ;
                len <= len+1;
                DQorg <= o0;
        elsif(Q='1' and len>0 and DUntil>=t) then
                DQ_we <= '1';
                DQ_en <= '1';
                addr <= PidNoDQ;
                DQ_assign <= DUntil;
                len <= len+1;
                DQorg <= o0;
        elsif(Q='1' and len>0 and DUntil<t) then
                DQ_we <= '1';
                DQ_en <= '1';
                addr <= PidDQ;
                DQ_assign <= t;
                DQorg <= o1;
        elsif(t<=sTime and len=1) then
                DQ_we <= '0';
                DQ_en <= '0';
                PE <= PidDQ;
                DQorg <= o2;
        elsif(t<=sTime and len>1) then
                DQ_we <= '0';
                DQ_en <= '0';
                t <= (others => '0');
                PE <= PidDQ;
                i := (others => '0');
                j := (others => '0');
                len <= len-1;
                DQorg <= o3;
        else
                DQ_we <= '0';
                DQ_en <= '0';
                DQorg <= o0;
        end if;
when o1 =>
        DQ_we <= '0';
        DQ_en <= '0';
        t <= DUntil;
        PidDQ <= PidNoDQ;
        len <= len+1;
        DQorg <= o0;
```

```
when o2 =>
        DQ_we <= '0';
        DQ_en <= '0';
        if(QE='1') then
                len <= (others => '0');
                t <= (others => '0');
                DQorg <= o0;
        else
                DQorg <= o2;
        end if;
when o3 =>
        DQ_we <= '0';
        DQ_en <= '0';
        if(QE='1') then
                DQorg <= o4;
        else
                DQorg <= o3;
        end if;
when o4 =>
        if(i=NUMTASKS-1) then
                DQ_we <= '0';
                DQ_en <= '1';
                PidDQ <= i;
                addr <= i;
                DQorg <= o5;
        elsif(i<NUMTASKS-1) then
                DQ_we <= '0';
                DQ_en <= '1';
                addr <= i;
                DQorg <= o6;
        else
                DQ_we <= '0';
                DQ_en <= '0';
                DQorg <= o4;
        end if;
when o5 =>
        if(DQ_read_ack='1') then
                t <= DQ_read;
                DQ_we <= '1';
                DQ_en <= '1';
                addr <= PidDQ;
                DQ_assign <= (others => '0');
                DQorg <= o0;
        else
                DQ_we <= '0';
                DQ_en <= '0';
                DQorg <= o5;
            end if;
        when o6 =>
            DQ_we <= '0';
            DQ_en <= '0';
            if(DQ_read_ack='1') then
                i := i+1;
                if(DQ_read>0) then
                    PidDQ <= addr;
                    j := DQ_read;
                    DQorg <= o7;
                else
                    DQorg <= o4;
                end if;
            else
                DQorg <= o6;
            end if;
        when o7=>
            DQ_we <= '0';
            DQ_en <= '1';
            addr <= i;
            if(i<NUMTASKS-1) then
                DQorg <= o8;
            else
                DQorg <= o9;
            end if;
        when o8 =>
            DQ_we <= '0';
            DQ_en <= '0';
            if(DQ_read_ack='1') then
                i := i+1;
                DQorg <= o7;
                if(DQ_read>0 and DQ_read<j) then
                    PidDQ <= addr;
                    j := DQ_read;
                end if;
            else
                DQorg <= o8;
            end if;
        when o9 =>
            if(DQ_read_ack='1') then
                DQorg <= o0;
                DQ_we <= '1';
                DQ_en <= '1';
                DQ_assign <= (others => '0');
                if(DQ_read>0 and DQ_read<j) then
```

```
                                                  PidDQ <= i;
                                                  t <= DQ_read;
                                                  addr <= i;
                                          else
                                                  t <= j;
                                                  addr <= PidDQ;
                                          end if;
                                  else
                                          DQorg <= o9;
                                  end if;
                          when others =>
                                  DQorg <= o0;
                  end case;
          end if;
  end process dq_organize;

  -- ASSIGN STATE BITS TO OUTPUT
  DQackState <= DQack;
  DQorgState <= DQorg;
  DQ_addr <= addr;

end Behavioral;
```

A.5 Timing

```
---------------------------------------------------------------------------------------------------------
-- NAME: timer.vhd
--
-- INPUTS:    clk           FPGA clock
--            reset_n       FPGA reset, active low
--            setfreq       signal specifying that set frequency command has been made
--            freq          frequency at which VCounter should increment at
--
-- OUTPUTS: status          status of this component
--            VCounter      counter value
--
-- DESCRIPTION:  This is the Ada time counter.  VCounter increases once every 'freq' clk
--      cycles
---------------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.variables.ALL;

entity timer is
   Port ( clk           : in std_logic;
        reset_n         : in std_logic;
        setfreq         : in std_logic;
        freq            : in std_logic_vector(FREQREG-1 downto 0);
        status          : out std_logic;
        VCounter        : out std_logic_vector(MAXTIME-1 downto 0));
end timer;

architecture Behavioral of timer is
        signal int_cntr : std_logic_vector(FREQREG-1 downto 0);
        signal int_freq : std_logic_vector(FREQREG-1 downto 0);
        signal int_time : std_logic_vector(MAXTIME-1 downto 0);

begin
        VCounter <= int_time;

        process(clk, reset_n)
        begin
                if(reset_n='0') then
                        int_time <= (others => '0');
                        int_cntr <= STARTFREQ;
                        int_freq <= (others => '0');
                        status <= '0';
                elsif(clk'event and clk='1') then
                        if(setfreq='1') then
                                status <= '1';
                                int_cntr <= STARTFREQ;
                                int_freq <= freq;
                                int_time <= (others => '0');
                        elsif(int_cntr=int_freq) then
                                status <= '0';
                                int_cntr <= STARTFREQ;
                                int_time <= int_time+1;
                        else
                                status <= '0';
                                int_cntr <= int_cntr+1;
                        end if;
                end if;
        end process;

end Behavioral;
```

A.6 Protected Objects

```
-----------------------------------------------------------------------------------------------
-- NAME: multipo.vhd
--
-- INPUTS:    clk           FPGA clock
--            reset_n       FPGA reset, active low
--            Pcpu          priority of task running on PPC
--            Tcpu          ID of task running on PPC
--            CeilPrio      ceiling priority of PO being accessed
--            FPs           array of FPs commands for all POs
--            Upe           array of Upe commands for all POs
--            Ufe           array of Ufe commands for all POs
--            Ufpx          array of Ufpx commands for all POs
--            UPxe          array of UPxe commands for all POs
--            UFxe          array of UFxe commands for all POs
--            Barrier       barrier value of PO being accessed
--            BarrierNew    signal indicating that the Barrier value is valid
--            FPe           array of FPe commands for all POs
--            FPx           array of FPx commands for all POs
--            Es            array of Es commands for all POs
--            UEb           array of UEb commands for all POs
--            UEe           array of UEe commands for all POs
--            UEx           array of UEx commands for all POs
--            Ex            array of Ex commands for all POs
--            RQState       state of RQSM
--
-- OUTPUTS: BarrierReq      request to get PO barrier values
--            Suspend       signal that task calling Entry must suspend itself
--            FindNew       signal to RQSM that a new task to run must be found
--            NPcpu         what priority should now be of task running on PPC
--            RQ_en         enable access to RQ_RAM
--            RQ_we         write enable to RQ_RAM
--            RQ_addr       address to access in RQ_RAM
--            RQ_assign     data to write to RQ_RAM
--            PO_Status     status of this component
--
-- DESCRIPTION: generates NUMPO protected objects from a single protected object, where
--      NUMPO is the total number of protected object, defined in myvariables.vhd
-----------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.variables.ALL;
```

```
entity multiPO is
   Port ( clk          : in std_logic;
         reset_n       : in std_logic;
         Pcpu          : in std_logic_vector(BITPRIO-1 downto 0);
         Tcpu          : in std_logic_vector(BITTASKS-1 downto 0);
         CeilPrio      : in std_logic_vector(BITPRIO-1 downto 0);
         FPs           : in std_logic_vector(NUMPO-1 downto 0);
         Upe           : in std_logic_vector(NUMPO-1 downto 0);
         Ufe           : in std_logic_vector(NUMPO-1 downto 0);
         Ufpx          : in std_logic_vector(NUMPO-1 downto 0);
         UPxe          : in std_logic_vector(NUMPO-1 downto 0);
         UFxe          : in std_logic_vector(NUMPO-1 downto 0);
         Barrier       : in std_logic;
         BarrierNew    : in std_logic;
         FPe           : in std_logic_vector(NUMPO-1 downto 0);
         FPx           : in std_logic_vector(NUMPO-1 downto 0);
         Es            : in std_logic_vector(NUMPO-1 downto 0);
         UEb           : in std_logic_vector(NUMPO-1 downto 0);
         UEe           : in std_logic_vector(NUMPO-1 downto 0);
         UEx           : in std_logic_vector(NUMPO-1 downto 0);
         Ex            : in std_logic_vector(NUMPO-1 downto 0);
         RQState       : in rqs;
         BarrierReq    : out std_logic_vector(NUMPO-1 downto 0);
         Suspend       : out std_logic_vector(NUMPO-1 downto 0);
         FindNew       : out std_logic_vector(NUMPO-1 downto 0);
         NPcpu         : out all_NPcpu;
         RQ_en         : out std_logic_vector(NUMPO-1 downto 0);
         RQ_we         : out std_logic_vector(NUMPO-1 downto 0);
         RQ_addr       : out all_RQaddr;
         RQ_assign     : out all_RQassign;
         PO_Status     : out std_logic_vector(NUMPO-1 downto 0));
end multiPO;

architecture Behavioral of multiPO is
         component PO_one
                 port (  clk          : in std_logic;
                         reset_n      : in std_logic;
                         Pcpu         : in std_logic_vector(BITPRIO-1 downto 0);
                         Tcpu         : in std_logic_vector(BITTASKS-1 downto 0);
                         CeilPrio     : in std_logic_vector(BITPRIO-1 downto 0);
                         FPs          : in std_logic;
                         Upe          : in std_logic;
                         Ufe          : in std_logic;
                         Ufpx         : in std_logic;
                         UPxe         : in std_logic;
                         UFxe         : in std_logic;
```

```
                        Barrier         : in std_logic;
                        BarrierNew      : in std_logic;
                        FPe             : in std_logic;
                        FPx             : in std_logic;
                        Es              : in std_logic;
                        UEb             : in std_logic;
                        UEe             : in std_logic;
                        UEx             : in std_logic;
                        Ex              : in std_logic;
                        RQState         : in rqs;
                        BarrierReq      : out std_logic;
                        Suspend         : out std_logic;
                        FindNew         : out std_logic;
                        NPcpu           : out std_logic_vector(BITPRIO-1 downto 0);
                        RQ_en           : out std_logic;
                        RQ_we           : out std_logic;
                        RQ_addr         : out std_logic_vector(BITTASKS-1 downto 0);
                        RQ_assign       : out std_logic_vector(RQDATASIZE-1 downto 0);
                        PO_Status       : out std_logic);
        end component;

for all: PO_one use entity work.PO_one(Behavioral);
begin

        PO_gen: for i in 0 to NUMPO-1 generate
                PO: PO_one
                        port map ( clk, reset_n, Pcpu(BITPRIO-1 downto 0),
                                Tcpu(BITTASKS-1 downto 0), CeilPrio(BITPRIO-1 downto 0),
                                FPs(i), Upe(i), Ufe(i), Ufpx(i), UPxe(i), UFxe(i),
                                Barrier, BarrierNew, FPe(i), FPx(i), Es(i), UEb(i),
                                UEe(i), UEx(i), Ex(i), RQState, BarrierReq(i),
                                Suspend(i), FindNew(i), NPcpu(i), RQ_en(i), RQ_we(i),
                                RQ_addr(i), RQ_assign(i), PO_Status(i));
        end generate PO_gen;

end Behavioral;

------------------------------------------------------------------------------------------------------------------
-- NAME: po_one.vhd
--
-- INPUTS:      clk             FPGA clock
--              reset_n         FPGA reset, active low
--              Pcpu            priority of task running on PPC
--              Tcpu            ID of task running on PPC
--              CeilPrio        ceiling priority of PO being accessed
--              FPs             array of FPs commands for all POs
--              Upe          array of Upe commands for all POs
--              Ufe          array of Ufe commands for all POs
--              Ufpx         array of Ufpx commands for all POs
--              UPxe         array of UPxe commands for all POs
--              UFxe         array of UFxe commands for all POs
--              Barrier      barrier value of PO being accessed
--              BarrierNew   signal indicating that the Barrier value is valid
--              FPe          array of FPe commands for all POs
--              FPx          array of FPx commands for all POs
--              Es           array of Es commands for all POs
--              UEb          array of UEb commands for all POs
--              UEe          array of UEe commands for all POs
--              UEx          array of UEx commands for all POs
--              Ex           array of Ex commands for all POs
--              RQState      state of RQSM
--
-- OUTPUTS: BarrierReq   request to get PO barrier values
--              Suspend      signal that task calling Entry must suspend itself
--              FindNew      signal to RQSM that a new task to run must be found
--              NPcpu        what priority should now be of task running on PPC
--              RQ_en        enable access to RQ_RAM
--              RQ_we        write enable to RQ_RAM
--              RQ_addr      address to access in RQ_RAM
--              RQ_assign    data to write to RQ_RAM
--              PO_Status    status of this component
--
-- DESCRIPTION:  combines the elements of one PO: entry, procedure/function, and channel
--       controller
------------------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.variables.ALL;

entity PO_one is
   Port ( clk             : in std_logic;
          reset_n         : in std_logic;
          Pcpu            : in std_logic_vector(BITPRIO-1 downto 0);
          Tcpu            : in std_logic_vector(BITTASKS-1 downto 0);
          CeilPrio        : in std_logic_vector(BITPRIO-1 downto 0);
          FPs             : in std_logic;
          Upe             : in std_logic;
          Ufe             : in std_logic;
          Ufpx            : in std_logic;
```

```
        UPxe          : in std_logic;
        UFxe          : in std_logic;
        Barrier       : in std_logic;
        BarrierNew    : in std_logic;
        FPe           : in std_logic;
        FPx           : in std_logic;
        Es            : in std_logic;
        UEb           : in std_logic;
        UEe           : in std_logic;
        UEx           : in std_logic;
        Ex            : in std_logic;
        RQState       : in rqs;
        BarrierReq    : out std_logic;
        Suspend       : out std_logic;
        FindNew       : out std_logic;
        NPcpu         : out std_logic_vector(BITPRIO-1 downto 0);
        RQ_en         : out std_logic;
        RQ_we         : out std_logic;
        RQ_addr       : out std_logic_vector(BITTASKS-1 downto 0);
        RQ_assign     : out std_logic_vector(RQDATASIZE-1 downto 0);
        PO_Status     : out std_logic);
end PO_one;

architecture Behavioral of PO_one is
        signal ecount           : std_logic;
        signal E_en             : std_logic;
        signal P_en             : std_logic;
        signal E_we             : std_logic;
        signal P_we             : std_logic;
        signal E_addr           : std_logic_vector(BITTASKS-1 downto 0);
        signal P_addr           : std_logic_vector(BITTASKS-1 downto 0);
        signal E_assign         : std_logic_vector(RQDATASIZE-1 downto 0);
        signal P_assign         : std_logic_vector(RQDATASIZE-1 downto 0);
        signal E_npcpu          : std_logic_vector(BITPRIO-1 downto 0);
        signal P_npcpu          : std_logic_vector(BITPRIO-1 downto 0);
        signal E_status         : std_logic;
        signal P_status         : std_logic;
        signal Eg               : std_logic;
        signal Ef               : std_logic;
        signal ProcState        : pstate;
        signal EntryState       : estate;
        signal FindNew_int      : std_logic;
        signal Suspend_int      : std_logic;
        signal BarrierReqP      : std_logic;
        signal BarrierReqE      : std_logic;
```

```
component ProcFunc
        port (  clk             : in std_logic;
                reset_n         : in std_logic;
                Pcpu            : in std_logic_vector(BITPRIO-1 downto 0);
                Tcpu            : in std_logic_vector(BITTASKS-1 downto 0);
                CeilPrio        : in std_logic_vector(BITPRIO-1 downto 0);
                FPs             : in std_logic;
                Upe             : in std_logic;
                Ufe             : in std_logic;
                Ufpx            : in std_logic;
                UPxe            : in std_logic;
                UFxe            : in std_logic;
                Barrier         : in std_logic;
                BarrierNew      : in std_logic;
                ECount          : in std_logic;
                Ef              : in std_logic;
                FindNew         : in std_logic;
                Status          : out std_logic;
                BarrierReq      : out std_logic;
                Eg              : out std_logic;
                NPcpu           : out std_logic_vector(BITPRIO-1 downto 0);
                RQ_en           : out std_logic;
                RQ_we           : out std_logic;
                RQ_addr         : out std_logic_vector(BITTASKS-1 downto 0);
                RQ_assign       : out std_logic_vector(RQDATASIZE-1 downto 0);
                ProcState       : out pstate);
end component;

component entry
        port (  clk             : in std_logic;
                reset_n         : in std_logic;
                Tcpu            : in std_logic_vector(BITTASKS-1 downto 0);
                Pcpu            : in std_logic_vector(BITPRIO-1 downto 0);
                CeilPrio        : in std_logic_vector(BITPRIO-1 downto 0);
                Es              : in std_logic;
                Barrier         : in std_logic;
                BarrierNew      : in std_logic;
                UEb             : in std_logic;
                UEx             : in std_logic;
                UEe             : in std_logic;
                Ex              : in std_logic;
                FindNew         : in std_logic;
                Suspend         : in std_logic;
                Eg              : in std_logic;
                Ef              : in std_logic;
                BarrierReq      : out std_logic;
```

```
                    ECount         : out std_logic;
                    Status         : out std_logic;
                    NPcpu          : out std_logic_vector(BITPRIO-1 downto 0);
                    RQ_en          : out std_logic;
                    RQ_we          : out std_logic;
                    RQ_addr        : out std_logic_vector(BITTASKS-1 downto 0);
                    RQ_assign      : out std_logic_vector(RQDATASIZE-1 downto 0);
                    EntryState     : out estate);
        end component;

        component channel_controller_po1
                port (  ProcState      : in pstate;
                        EntryState     : in estate;
                        RQState        : in rqs;
                        FPe            : in std_logic;
                        FPx            : in std_logic;
                        Ex             : in std_logic;
                        UEe            : in std_logic;
                        Ef             : out std_logic;
                        Suspend        : out std_logic;
                        FindNew        : out std_logic);
        end component;

begin
        proc_routine : ProcFunc
                port map (clk=>clk, reset_n=>reset_n, pcpu=>Pcpu, tcpu=>Tcpu,
                        ceilprio=>CeilPrio, fps=>FPs, upe=>Upe, ufe=>Ufe, ufpx=>Ufpx,
                        upxe=>Upxe, ufxe=>Ufxe, barrier=>Barrier, barrierNew=>BarrierNew,
                        ecount=>ecount, ef=>Ef, findnew=>FindNew_int, status=>P_status,
                        barrierReq=>BarrierReqP, eg=>Eg, npcpu=>P_npcpu, rq_en=>P_en,
                        rq_we=>P_we, rq_addr=>P_addr, rq_assign=>P_assign,
                        procstate=>ProcState);

        entry_routine : entry
                port map (clk=>clk, reset_n=>reset_n, tcpu=>Tcpu, pcpu=>Pcpu,
                        ceilprio=>CeilPrio, es=>Es, barrier=>Barrier, barrierNew=>BarrierNew,
                        ueb=>Ueb, uex=>Uex, uee=>Uee, ex=>Ex, findnew=>FindNew_int,
                        suspend=>Suspend_int, eg=>Eg, ef=>Ef, barrierReq=>barrierReqE,
                        ecount=>ecount, status=>E_status, npcpu=>E_npcpu, rq_en=>E_en,
                        rq_we=>E_we, rq_addr=>E_addr, rq_assign=>E_assign,
                        entrystate=>EntryState);

        channels : channel_controller_PO1
                port map (ProcState=>ProcState, EntryState=>EntryState, RQState=>RQState,
                        FPe=>FPe, FPx=>FPx, Ex=>Ex, UEe=>UEe, Ef=>Ef,
                        Suspend=>Suspend_int, FindNew=>FindNew_int);
        -- assign internal findnew and suspend to output, so readyqueue can access
        FindNew <= FindNew_int;
        Suspend <= Suspend_int;

        -- OR status lines to get PO_status
        PO_status <= P_status OR E_status;

        -- MUX RQ ram lines to determine which is valid    - use enable lines
        -- since design won't any two PO or any parts to access RAM at once
        -- (only 1 task runs at a time)
        RQ_en <= '1' when (P_en='1' or E_en='1') else '0';
        RQ_we <= '1' when (P_we='1' or E_we='1') else '0';
        RQ_addr <= P_addr when P_en='1' else E_addr when E_en='1' else (others => '0');
        RQ_assign <= P_assign when (P_en='1' and P_we='1') else
                        E_assign when (E_en='1' and E_we='1') else (others => '0');

        -- which NPcpu is valid :
        -- in one PO, they will never be set at same time.
        NPcpu <= P_npcpu when (P_npcpu >= E_npcpu) else E_npcpu;

        -- barrier request line
        BarrierReq <= BarrierReqP OR BarrierReqE;

end Behavioral;

----------------------------------------------------------------------------------------------------
-- NAME: channel_controller_PO1.vhd
--
-- INPUTS:     ProcState      state of FPSM
--             EntryState     state of ESM
--             RQState        state of RQSM
--             FPe            signal that FPe command has been made
--             FPx            signal that FPx command has been made
--             Ex             signal that Ex command has been made
--             UEe            signal that UEe command has been made
--
-- OUTPUTS:  Ef               channel use by ESM and FPSM
--             Suspend        channel indicating that a task is suspending
--             FindNew        signal to RQSM that a new task to run needs to be found
--
-- DESCRIPTION:  determines channels in a single PO
----------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.variables.ALL;

entity channel_controller_PO1 is
  Port ( ProcState      : in pstate;
         EntryState     : in estate;
         RQState        : in rqs;
         FPe            : in std_logic;
         FPx            : in std_logic;
         Ex             : in std_logic;
         UEe            : in std_logic;
         Ef             : out std_logic;
         Suspend        : out std_logic;
         FindNew        : out std_logic);
end channel_controller_PO1;

architecture Behavioral of channel_controller_PO1 is

begin
      Ef <= '1' when ((ProcState=P11 or ProcState=P5) and EntryState=E14) else '0';
      Suspend <= '1' when (EntryState=E7 and RQState=r0) else '0';
      FindNew <= '1' when (((EntryState=E5 and UEe='1') or
                            (EntryState=E6 and Ex='1')  or
                            (ProcState=P6 and FPe='1') or
                            (ProcState=P12 and FPx='1')) and RQState=r0) else '0';

end Behavioral;

------------------------------------------------------------------------------------------------------------
-- NAME: entry.vhd
--
-- INPUTS:    clk          FPGA clock
--            reset_n      FPGA reset, active low
--            Pcpu         priority of task running on PPC
--            Tcpu         ID of task running on PPC
--            CeilPrio     ceiling priority of PO being accessed
--            Es           signal that Es command has been made
--            Barrier      barrier value of PO being accessed
--            BarrierNew   signal indicating that the Barrier value is valid
--            UEb          signal that UEb command has been made
--            UEe          signal that UEe command has been made
--            UEx          signal that UEx command has been made
--            Ex           signal that Ex command has been made
--            FindNew      signal that indicates exit of PO routine, a new task must be found
--            Suspend      signal that task calling Entry must suspend itself
--              Eg              signal from FPSM to signal entry code should be executed
--              Ef              channel used between ESM and FPSM to return control back to
--                              FPSM after entry code is executed
--
-- OUTPUTS: BarrierReq      request to get PO barrier value
--              ECount          signal indicating a task is suspended on the entry queue
--              Status  status of this component
--              NPcpu           what priority should now be of task running on PPC
--              RQ_en           enable access to RQ_RAM
--              RQ_we           write enable to RQ_RAM
--              RQ_addr         address to access in RQ_RAM
--              RQ_assign       data to write to RQ_RAM
--              EntryState      state of ESM
--
-- DESCRIPTION:  this is a protected entry routine.  Checks barrier value at first; if equal to 1,
--       entry executes.  If equal to 0, task suspends itself and adds itself to entry queue
--       (ECount=1), and procedure must continue execution of code later
------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.variables.ALL;

entity Entry is
   Port ( clk           : in std_logic;
          reset_n       : in std_logic;
          Tcpu          : in std_logic_vector(BITTASKS-1 downto 0);
          Pcpu          : in std_logic_vector(BITPRIO-1 downto 0);
          CeilPrio      : in std_logic_vector(BITPRIO-1 downto 0);
          Es            : in std_logic;
          Barrier       : in std_logic;
          BarrierNew    : in std_logic;
          UEb           : in std_logic;
          UEx           : in std_logic;
          UEe           : in std_logic;
          Ex            : in std_logic;
          FindNew       : in std_logic;
          Suspend       : in std_logic;
          Eg            : in std_logic;
          Ef            : in std_logic;
          BarrierReq    : out std_logic;
          ECount        : out std_logic;
          Status        : out std_logic;
          NPcpu         : out std_logic_vector(BITPRIO-1 downto 0);
```

```
        RQ_en           : out std_logic;
        RQ_we           : out std_logic;
        RQ_addr         : out std_logic_vector(BITTASKS-1 downto 0);
        RQ_assign       : out std_logic_vector(RQDATASIZE-1 downto 0);
        EntryState      : out estate);
end Entry;

architecture Behavioral of Entry is
        signal E : estate;

begin
        process(clk, reset_n)
        variable Ep : std_logic_vector(BITPRIO-1 downto 0);
        variable Et : std_logic_vector(BITTASKS-1 downto 0);
        begin
                if(reset_n='0') then
                        Status <= '0';
                        RQ_en <= '0';
                        RQ_we <= '0';
                        RQ_addr <= (others => '0');
                        RQ_assign <= (others => '0');
                        NPcpu <= (others => '0');
                        Ep := (others => '0');
                        Et := (others => '0');
                        BarrierReq <= '0';
                        ECount <= '0';
                        E <= E0;
                elsif(clk'event and clk='1') then
                        case E is
                                when E0 =>
                                        RQ_en <= '0';
                                        RQ_we <= '0';
                                        Status <= '0';
                                        NPcpu <= (others => '0');
                                        -- check barrier
                                        if(Es='1') then
                                                BarrierReq <= '1';
                                                E <= E1;
                                        else
                                                BarrierReq <= '0';
                                                E <= E0;
                                        end if;
                                when E1 =>
                                        if(BarrierNew='1') then
                                                -- entry is executed
                                                BarrierReq <= '0';
```

```
                    if(Barrier='1') then
                            Status <= '1';
                            RQ_en <= '1';
                            RQ_we <= '1';
                            RQ_addr <= Tcpu;
                            RQ_assign(RQDATASIZE-1 downto 1) <=
                                    CeilPrio;
                            RQ_assign(0) <= '1';
                            Ep := Pcpu;
                            NPcpu <= CeilPrio;
                            E <= E2;
                    else -- Barrier='0', entry is suspended
                            Status <= '0';
                            RQ_en <= '0';
                            RQ_we <= '0';
                            ECount <= '1';
                            Et := Tcpu;
                            Ep := Pcpu;
                            E <= E7;
                    end if;
            else
                    RQ_en <= '0';
                    RQ_we <= '0';
                    Status <= '0';
                    NPcpu <= (others => '0');
                    E <= E1;
            end if;
    when E2 =>
            RQ_en <= '0';
            RQ_we <= '0';
            Status <= '0';
            E <= E3;
    when E3 =>
            RQ_en <= '0';
            RQ_we <= '0';
            if(UEb='1') then
                    Status <= '1';
                    E <= E4;
            else
                    Status <= '0';
                    E <= E3;
            end if;
    when E4 =>
            RQ_en <= '0';
            RQ_we <= '0';
            Status <= '0';
```

```
        E <= E5;
when E5 =>
        if(UEx='1') then
                RQ_en <= '0';
                RQ_we <= '0';
                Status <= '1';
                E <= E6;
        elsif(FindNew='1') then
                Status <= '0';
                RQ_en <= '1';
                RQ_we <= '1';
                RQ_addr <= Tcpu;
                RQ_assign(RQDATASIZE-1 downto 1) <= Ep;
                RQ_assign(0) <= '1';
                E <= E0;
        else
                RQ_en <= '0';
                RQ_we <= '0';
                Status <= '0';
                E <= E5;
        end if;
when E6 =>
        Status <= '0';
        if(FindNew='1') then
                RQ_en <= '1';
                RQ_we <= '1';
                RQ_addr <= Tcpu;
                RQ_assign(RQDATASIZE-1 downto 1) <= Ep;
                RQ_assign(0) <= '1';
                E <= E0;
        else
                RQ_en <= '0';
                RQ_we <= '0';
                E <= E6;
        end if;
when E7 =>
        RQ_en <= '0';
        RQ_we <= '0';
        Status <= '0';
        if(Suspend='1') then
                E <= E8;
        else
                E <= E7;
        end if;
when E8 =>
        RQ_en <= '0';
        RQ_we <= '0';
        Status <= '0';
        if(Eg='1') then
                ECount <= '0';
                E <= E9;
        else
                E <= E8;
        end if;
when E9 =>
        RQ_en <= '0';
        RQ_we <= '0';
        if(UEb='1') then
                Status <= '1';
                E <= E10;
        else
                Status <= '0';
                E <= E9;
        end if;
when E10 =>
        RQ_en <= '0';
        RQ_we <= '0';
        Status <= '0';
        E <= E11;
when E11 =>
        RQ_en <= '0';
        RQ_we <= '0';
        if(UEx='1') then
                Status <= '1';
                E <= E12;
        elsif(UEe='1') then
                Status <= '1';
                E <= E14;
        else
                Status <= '0';
                E <= E11;
        end if;
when E12 =>
        RQ_en <= '0';
        RQ_we <= '0';
        Status <= '0';
        E <= E13;
when E13 =>
        RQ_en <= '0';
        RQ_we <= '0';
        if(Ex='1') then
                Status <= '1';
```

```
                                        E <= E14;
                                else
                                        Status <= '0';
                                        E <= E13;
                                end if;
                        when E14 =>
                                Status <= '0';
                                if(Ef='1') then
                                        RQ_en <= '1';
                                        RQ_we <= '1';
                                        RQ_addr <= Et;
                                        RQ_assign(RQDATASIZE-1 downto 1) <= Ep;
                                        -- make suspended task valid again here
                                        RQ_assign(0) <= '1';
                                        E <= E0;
                                else
                                        RQ_en <= '0';
                                        RQ_we <= '1';
                                        E <= E14;
                                end if;
                        when others =>
                                E <= E0;
                end case;
            end if;
        end process;

        -- ASSIGN STATE BITS TO OUTPUT
        EntryState <= E;
end Behavioral;

----------------------------------------------------------------------------------------------------
-- NAME: procfunc.vhd
--
-- INPUTS:     clk            FPGA clock
--             reset_n        FPGA reset, active low
--             Pcpu           priority of task running on PPC
--             Tcpu           ID of task running on PPC
--             CeilPrio       ceiling priority of PO being accessed
--             FPs            signal that FPs command has been made
--             Upe            signal that Upe command has been made
--             Ufe            signal that Ufe command has been made
--             Ufpx           signal that Ufpx command has been made
--             UPxe           signal that UPxe command has been made
--             UFxe           signal that UFxe command has been made
--             Barrier        barrier value of PO being accessed
--             BarrierNew     signal indicating that the Barrier value is valid
--              ECount        signal indicating if a task is suspended on the entry queue
--              Ef            channel used between ESM and FPSM to return control back to
--                            FPSM after entry code is executed
--              FindNew       signal indicating that a PO is finishing, new task must be found
--
-- OUTPUTS: Status            status of this component
--              BarrierReq    request to get PO barrier values
--              Eg            signal to ESM that entry code should be executed
--              NPcpu         what priority should now be of task running on PPC
--              RQ_en         enable access to RQ_RAM
--              RQ_we         write enable to RQ_RAM
--              RQ_addr       address to access in RQ_RAM
--              RQ_assign     data to write to RQ_RAM
--              ProcState     state of FPSM
--
-- DESCRIPTION: This is a protected procedure/protected function routine. Monitors
--      execution of procedure or function. Difference between them is that before exit,
--      procedure checks barrier value and executes entry code if a task is on the entry queue
--      function does not do this
----------------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.variables.ALL;

entity ProcFunc is
  Port ( clk          : in std_logic;
      reset_n         : in std_logic;
      Pcpu            : in std_logic_vector(BITPRIO-1 downto 0);
      Tcpu            : in std_logic_vector(BITTASKS-1 downto 0);
      CeilPrio        : in std_logic_vector(BITPRIO-1 downto 0);
      FPs             : in std_logic;
      Upe             : in std_logic;
      Ufe             : in std_logic;
      Ufpx            : in std_logic;
      UPxe            : in std_logic;
      UFxe            : in std_logic;
      Barrier         : in std_logic;
      BarrierNew      : in std_logic;
      ECount          : in std_logic;
      Ef              : in std_logic;
      FindNew         : in std_logic;
      Status          : out std_logic;
      BarrierReq      : out std_logic;
```

```
        Eg          : out std_logic;
        NPcpu       : out std_logic_vector(BITPRIO-1 downto 0);
        RQ_en       : out std_logic;
        RQ_we       : out std_logic;
        RQ_addr     : out std_logic_vector(BITTASKS-1 downto 0);
        RQ_assign   : out std_logic_vector(RQDATASIZE-1 downto 0);
        ProcState   : out pstate);
end ProcFunc;

architecture Behavioral of ProcFunc is
        signal P : pstate;

begin
        process(clk, reset_n)
        variable prio : std_logic_vector(BITPRIO-1 downto 0);
        begin
                if(reset_n='0') then
                        status <= '0';
                        BarrierReq <= '0';
                        RQ_en <= '0';
                        RQ_we <= '0';
                        RQ_addr <= (others => '0');
                        RQ_assign <= (others => '0');
                        NPcpu <= (others => '0');
                        BarrierReq <= '0';
                        Eg <= '0';
                        prio := (others => '0');
                        P <= P0;
                elsif(clk'event and clk='1') then
                        case P is
                                when P0 =>
                                        if(FPs='1') then
                                                RQ_en <= '1';
                                                RQ_we <= '1';
                                                RQ_addr <= Tcpu;
                                                RQ_assign(RQDATASIZE-1 downto 1) <=
                                                        CeilPrio;
                                                -- must be valid, or wouldn't be running
                                                RQ_assign(0) <= '1';
                                                prio := Pcpu;
                                                NPcpu <= CeilPrio;
                                                Status <= '1';
                                                P <= P1;
                                        else
                                                RQ_en <= '0';
                                                RQ_we <= '0';
                        Status <= '0';
                        NPcpu <= (others => '0');
                        P <= P0;
                end if;
        when P1 =>
                Status <= '0';
                RQ_en <= '0';
                RQ_we <= '0';
                P <= P2;
        when P2 =>
                RQ_en <= '0';
                RQ_we <= '0';
                if(UPe='1') then
                        Status <= '0';
                        P <= P3;
                elsif(UFe='1') then
                        Status <= '1';
                        P <= P6;
                elsif(Ufpx='1') then
                        Status <= '1';
                        P <= P7;
                else
                        Status <= '0';
                        P <= P2;
                end if;
        when P3 =>
                RQ_en <= '0';
                RQ_we <= '0';
                if(ECount='0') then
                        P <= P6;
                        Status <= '1';
                else
                        BarrierReq <= '1';
                        Status <= '0';
                        P <= P4;
                end if;

        when P4 =>
                if(BarrierNew='1') then
                        BarrierReq <= '0';
                        Status <= '1';
                        if(Barrier='0') then
                                P <= P6;
                        else
                                P <= P5;
                                Eg <= '1';
                end if;
        else
                P <= P4;
                BarrierReq <= '1';
                Status <= '0';
        end if;
when P5 =>
        Eg <= '0';
        RQ_en <= '0';
        RQ_we <= '0';
        Status <= '0';
        if(Ef='1') then
                P <= P6;
        else
                P <= P5;
        end if;
when P6 =>
        Status <= '0';
        if(FindNew='1') then
                RQ_en <= '1';
                RQ_we <= '1';
                RQ_addr <= Tcpu;
                RQ_assign(RQDATASIZE-1 downto 1) <= prio;
                RQ_assign(0) <= '1';
                P <= P0;
        else
                RQ_en <= '0';
                RQ_we <= '0';
                P <= P6;
        end if;
when P7 =>
        RQ_en <= '0';
        RQ_we <= '0';
        Status <= '0';
        P <= P8;
when P8 =>
        RQ_en <= '0';
        RQ_we <= '0';
        if(UPxe='1') then
                Status <= '0';
                P <= P9;
        elsif(UFxe='1') then
                Status <= '1';
                P <= P12;
        else
                Status <= '0';
                P <= P8;
        end if;
when P9 =>
        RQ_en <= '0';
        RQ_we <= '0';
        if(ECount='0') then
                P <= P12;
                Status <= '1';
        else
                BarrierReq <= '1';
                Status <= '0';
                P <= P10;
        end if;
when P10 =>
        if(BarrierNew='1') then
                BarrierReq <= '0';
                Status <= '1';
                if(Barrier='0') then
                        P <= P12;
                else
                        P <= P11;
                        Eg <= '1';
                end if;
        else
                P <= P10;
                BarrierReq <= '1';
                Status <= '0';
        end if;
when P11 =>
        Eg <= '0';
        RQ_en <= '0';
        RQ_we <= '0';
        Status <= '0';
        if(Ef='1') then
                P <= P12;
        else
                P <= P11;
        end if;
when P12 =>
        Status <= '0';
        if(FindNew='1') then
                RQ_en <= '1';
                RQ_we <= '1';
                RQ_addr <= Tcpu;
                RQ_assign(RQDATASIZE-1 downto 1) <= prio;
                RQ_assign(0) <= '1';
```

```
                                        P <= P0;
                                else
                                        RQ_en <= '0';
                                        RQ_we <= '0';
                                        P <= P12;
                                end if;
                        when others =>
                                P <= P0;
                end case;
        end if;
    end process;

    -- ASSIGN STATE BITS TO OUTPUT
    ProcState <= P;

end Behavioral;
```

A.7 Task Arbitration

```
-----------------------------------------------------------------------------------------------------------
-- NAME: arbitrate_cpu.vhd
--
-- INPUTS:     NPcpu_po     priority of the task the PO thinks should be running
--             Tcpu         ID of the running task
--             NPcpu_rq     priority of the task the RQSM thinks should be running
--             NTcpu_rq     ID of the task the RQSM thinks should be running
--
-- OUTPUTS:    NPcpu        priority of task to run on the PPC
--             NTcpu        ID of the task to run on the PPC
--
-- DESCRIPTION: this checks to see what process is requesting to run a task of highest priority,
--      in the case that the readyqueue wants to run a task that just woke from a delay at the same
--      time an entry or procedure routine starts and raises pcpu to ceilprio.  only need to account
--      for PO because only one PO process can occur at once.  kernel_internal passes to this the
--      data from the current PO being used, based on POId

--      for equal prio, PO always gets preference -- since it's already running, and task
--      from readyqueue is just waking up.
-----------------------------------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.variables.ALL;

entity arbitrate_cpu is
   Port ( NPcpu_po    : in std_logic_vector(BITPRIO-1 downto 0);
          Tcpu        : in std_logic_vector(BITTASKS-1 downto 0);
          NPcpu_rq    : in std_logic_vector(BITPRIO-1 downto 0);
          NTcpu_rq    : in std_logic_vector(BITTASKS-1 downto 0);
          NPcpu       : out std_logic_vector(BITPRIO-1 downto 0);
          NTcpu       : out std_logic_vector(BITTASKS-1 downto 0));
end arbitrate_cpu;

architecture Behavioral of arbitrate_cpu is

begin
       NPcpu <= NPcpu_po when (NPcpu_po >= NPcpu_rq) else NPcpu_rq;
       NTcpu <= Tcpu when ((NPcpu_po >= NPcpu_rq) and NPcpu_po/=0) else NTcpu_rq;

end Behavioral;
```

A.8 Variables and Constants

```
-- package file of user specified variables, constants (myvariables.vhd)

library IEEE;
use IEEE.STD_LOGIC_1164.all;

package variables is

----------------------------------
-- BUS interface constants

  -- length of actual in/out reg
  constant REG_LENGTH : integer := 32;
-- number of registers
  constant NREGS : integer := 8;

-- register addresses
  constant addr_reg1 : std_logic_vector(7 downto 0) := "00000100";        -- 4
  constant addr_reg2 : std_logic_vector(7 downto 0) := "00001000";        -- 8
  constant addr_reg3 : std_logic_vector(7 downto 0) := "00001100";        -- C
  constant addr_reg4 : std_logic_vector(7 downto 0) := "00010000";        -- 10
  constant addr_reg5 : std_logic_vector(7 downto 0) := "00010100";        -- 14
  constant addr_reg6 : std_logic_vector(7 downto 0) := "00011000";        -- 18
  constant addr_reg7 : std_logic_vector(7 downto 0) := "00011100";        -- 1C
  constant addr_reg8 : std_logic_vector(7 downto 0) := "00100000";        -- 20

----------------------------------

  constant NUMTASKS : integer := 4;         -- TOTAL NUMBER OF TASKS
  constant BITTASKS : integer := 2;         -- BITS NEEDED TO REPRESENT NUMTASKS
  constant BITPRIO : integer := 3;          -- BITS NEEDED TO REPRESENT TOTAL
                                            -- NUMBER OF PRIORITIES
  constant MAXDUNTIL : integer := 64;       -- SIZE OF THE PARAMETER REGISTER
  constant MAXTIME : integer := 64;         -- MAX BITS TIME WILL COUNT TO
  constant RQDATASIZE : integer := 4;       -- SIZE OF DATA HELD IN RQ RAM, should be
                                            -- BITPRIO+1 (for valid bit)
  constant NUMPO : integer := 4;            -- number of PROTECTED OBJECTS routines
  constant BITPO : integer := 2;            -- number of BITS to idx POs
  constant CMDREG : integer := 5;           -- number of BITS in COMMAND REGISTER
  constant PARAMREG : integer := 32;        -- number of BITS in EACH PARAM REGISTER
  constant STATREG : integer := 3;          -- number of BITS in status reg, output to software
  constant FREQREG : integer := 8;          -- number of BITS in freq reg
  constant STARTFREQ : std_logic_vector := "00000001"; -- what to initialize freq cntr at
                                                        -- need to put here, since FREQREG
                                                        -- might be variable - this changes
  -- commands:
  constant CREATEi          : std_logic_vector := "10000";
  constant DELAYUNTILi      : std_logic_vector := "00001";
  constant FPSi             : std_logic_vector := "00010";
  constant UPEi             : std_logic_vector := "00011";
  constant UFEi             : std_logic_vector := "10001";
  constant FPEi             : std_logic_vector := "00100";
  constant UFPXi            : std_logic_vector := "00101";
  constant UPXEi            : std_logic_vector := "00110";
  constant UFXEi            : std_logic_vector := "10010";
  constant FPXi             : std_logic_vector := "00111";
  constant ESi              : std_logic_vector := "01000";
  constant UEBi             : std_logic_vector := "01001";
  constant UEEi             : std_logic_vector := "01010";
  constant UEXi             : std_logic_vector := "01011";
  constant EXi              : std_logic_vector := "01100";
  constant GETTIMEi         : std_logic_vector := "01101";
  constant SETFREQi         : std_logic_vector := "01110";
  constant FINDTASKi        : std_logic_vector := "01111";

  -- status bits:
  constant NOSTAT           : std_logic_vector := "000";
  constant BADCMD           : std_logic_vector := "001";
  constant CMDDONE          : std_logic_vector := "010";
  constant GETBARR          : std_logic_vector := "100";

  -- to pass state bits between entities
  type pstate is (P0, P1, P2, P3, P4, P5, P6, P7, P8, P9, P10, P11, P12);
  type estate is (E0, E1, E2, E3, E4, E5, E6, E7, E8, E9, E10, E11, E12, E13, E14);
  type rqs is (reset_ram, r0, r1, r2, r3, r4, r5, r6, r7, r8);
  type dqa is (a0, a1, a2, a3);
  type dqo is (reset_ramDQ, o0, o1, o2, o3, o4, o5, o6, o7, o8, o9);

  -- to pass 2-d arrays----------
  subtype rqdata is std_logic_vector(RQDATASIZE-1 downto 0);
  type all_RQassign is array(NUMPO-1 downto 0) of rqdata;

  subtype taskaddr is std_logic_vector(BITTASKS-1 downto 0);
  type all_RQaddr is array(NUMPO-1 downto 0) of taskaddr;

  subtype newpcpu is std_logic_vector(BITPRIO-1 downto 0);
  type all_NPcpu is array(NUMPO-1 downto 0) of newpcpu;

end variables;
```

A.9 Constraints

-- constraints file (myconstraints.ucf) used for implementation and testing in ISE-------------------

```
OFFSET = IN 6 ns BEFORE "clk";
OFFSET = OUT 9 ns AFTER "clk";
NET "clk" TNM_NET = "clk";
TIMESPEC "TS_clk" = PERIOD "clk" 80 MHz HIGH 50 %;
INST "addr<0>" TNM = "addrs";
INST "addr<1>" TNM = "addrs";
INST "addr<2>" TNM = "addrs";
INST "addr<3>" TNM = "addrs";
INST "addr<4>" TNM = "addrs";
INST "addr<5>" TNM = "addrs";
INST "addr<6>" TNM = "addrs";
INST "addr<7>" TNM = "addrs";
TIMEGRP "addrs" OFFSET = IN 6 ns BEFORE "clk";
INST "write_n" TNM = "write_n";
TIMEGRP "write_n" OFFSET = IN 6 ns BEFORE "clk";
```

-- constraints file system.ucf, used in EDK for specifying clock and pin constraints

```
#
#--------------------------------------------------------------------------
#
# Constraints For Virtex II - Pro Design
#
#--------------------------------------------------------------------------
CONFIG PROHIBIT = RAMB16_X0Y0;

#--------------------------------------------------------------------------
# Timing Ignore Constraints
#--------------------------------------------------------------------------

#--------------------------------------------------------------------------
# Clock Period Constraints
#--------------------------------------------------------------------------

# 100 MHz Ref Clk to DCM Produces PLB(1X), CPU(FX=3X)
# (Over-Constrain Period by 250 ps to allow for Jitter, Skew, Noise, Etc)
NET "sys_clk" PERIOD = 10.00;

#--------------------------------------------------------------------------
# Multicycle Path Constraints for DCR
#--------------------------------------------------------------------------
```

```
#--------------------------------------------------------------------------
# IO Pad Location Constraints (2VP4/7 AFX Board)
#--------------------------------------------------------------------------

NET "sys_clk"          LOC = v12;
NET "sys_rst"          LOC = v15;

#UART
NET rx                 LOC = U9;
NET tx                 LOC = W7;

NET "plb_sdram_1_sdram_bankaddr<0>"  LOC = L4;
NET "plb_sdram_1_sdram_bankaddr<1>"  LOC = R4;
NET "plb_sdram_1_sdram_casn"  LOC = K3;
NET "plb_sdram_1_sdram_cke"  LOC = J3;
NET "plb_sdram_1_sdram_clk"  LOC = E3;
NET "plb_sdram_1_sdram_csn"  LOC = P3;
NET "plb_sdram_1_sdram_dqm<0>"  LOC = T3;
NET "plb_sdram_1_sdram_dqm<1>"  LOC = F4;
NET "plb_sdram_1_sdram_rasn"  LOC = T4;
NET "plb_sdram_1_sdram_wen"  LOC = R3;

NET "plb_sdram_1_sdram_addr<13>"       LOC = E4;
NET "plb_sdram_1_sdram_addr<12>"  LOC = G4;
NET "plb_sdram_1_sdram_addr<11>"  LOC = F3;
NET "plb_sdram_1_sdram_addr<10>"  LOC = N3;
NET "plb_sdram_1_sdram_addr<9>"  LOC = K4;
NET "plb_sdram_1_sdram_addr<8>"  LOC = H4;
NET "plb_sdram_1_sdram_addr<7>"  LOC = G3;
NET "plb_sdram_1_sdram_addr<6>"  LOC = L3;
NET "plb_sdram_1_sdram_addr<5>"  LOC = H3;
NET "plb_sdram_1_sdram_addr<4>"  LOC = J4;
NET "plb_sdram_1_sdram_addr<3>"  LOC = M3;
NET "plb_sdram_1_sdram_addr<2>"  LOC = N4;
NET "plb_sdram_1_sdram_addr<1>"  LOC = P4;
NET "plb_sdram_1_sdram_addr<0>"  LOC = M4;
NET "plb_sdram_1_sdram_dq<31>"  LOC = R1;
NET "plb_sdram_1_sdram_dq<30>"  LOC = P2;
NET "plb_sdram_1_sdram_dq<29>"  LOC = R2;
NET "plb_sdram_1_sdram_dq<28>"  LOC = P1;
NET "plb_sdram_1_sdram_dq<27>"  LOC = T1;
NET "plb_sdram_1_sdram_dq<26>"  LOC = N2;
NET "plb_sdram_1_sdram_dq<25>"  LOC = T2;
NET "plb_sdram_1_sdram_dq<24>"  LOC = N1;
NET "plb_sdram_1_sdram_dq<23>"  LOC = Y2;
NET "plb_sdram_1_sdram_dq<22>"  LOC = W2;
NET "plb_sdram_1_sdram_dq<21>"  LOC = Y1;
NET "plb_sdram_1_sdram_dq<20>"  LOC = V2;
NET "plb_sdram_1_sdram_dq<19>"  LOC = W1;
NET "plb_sdram_1_sdram_dq<18>"  LOC = U2;
NET "plb_sdram_1_sdram_dq<17>"  LOC = V1;
NET "plb_sdram_1_sdram_dq<16>"  LOC = U1;
NET "plb_sdram_1_sdram_dq<15>"  LOC = E2;
NET "plb_sdram_1_sdram_dq<14>"  LOC = D1;
NET "plb_sdram_1_sdram_dq<13>"  LOC = E1;
NET "plb_sdram_1_sdram_dq<12>"  LOC = D2;
NET "plb_sdram_1_sdram_dq<11>"  LOC = F2;
NET "plb_sdram_1_sdram_dq<10>"  LOC = G5;
NET "plb_sdram_1_sdram_dq<9>"  LOC = F1;
NET "plb_sdram_1_sdram_dq<8>"  LOC = F5;
NET "plb_sdram_1_sdram_dq<7>"  LOC = K1;
NET "plb_sdram_1_sdram_dq<6>"  LOC = J1;
NET "plb_sdram_1_sdram_dq<5>"  LOC = K2;
NET "plb_sdram_1_sdram_dq<4>"  LOC = H1;
NET "plb_sdram_1_sdram_dq<3>"  LOC = J2;
NET "plb_sdram_1_sdram_dq<2>"  LOC = G1;
NET "plb_sdram_1_sdram_dq<1>"  LOC = H2;
NET "plb_sdram_1_sdram_dq<0>"  LOC = G2;
```

Appendix B
Kernel Tests

B.1 ISE Tests

located in: ~kernel/

Test code name (or series of tests)	*Purpose*
create3.vhd	Creates 3 tasks
test_timer.vhd	Sets VCounter and gets the time
tfunction.vhd	Executes protected function
tprocedure.vhd	Executes protected procedure
tentry.vhd	Executes protected entry
tdq_1.vhd	Tests multiple tasks making delay until calls
tpcallinge.vhd	Entry causes task to suspends; Procedure calls entry code
int_proc_1.vhd through int_proc_4.vhd	Procedure executes and is interrupted at various points
int_entry_1.vhd through int_entry_3.vhd	Entry executes and is interrupted at various points
int_pcallinge_1.vhd through int_pcallinge_7.vhd	Entry causes task to suspend; Procedure calls entry code, and is interrupted at various points

B.2 EDK Tests

located in: ~/gurkh/sw/

Test code name (or series of tests)	*Purpose*
Create3.c	Creates 3 tasks
Test_func.c	Executes protected function
Test_proc.c	Executes protected procedure
Test_entry.c	Executes protected entry
Delay.c	Tests multiple tasks making delay until calls
Pcallinge.c	Entry causes task to suspends; Procedure calls entry code
Int_proc_1.c through int_proc_3.c	Procedure executes and is interrupted at various points
Int_entry_1.c through int_entry_3.c	Entry executes and is interrupted at various points
Int_pcallinge_1.c through in_pcallinge_7.c	Entry causes task to suspend; Procedure calls entry code, and is interrupted at various points
Test_proc_board.c	Executes protected procedure, and writes data to the computer screen
Test_entry_board.c	Executes protected entry, and writes data to the computer screen