HARDWARE IMPLEMENTATION OF REAL-TIME OPERATING SYSTEM'S THREAD CONTEXT SWITCH

by

Deepak Kumar Gauba

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Engineering

Boise State University

August 2010

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Deepak Kumar Gauba

Thesis Title: Hardware Implementation of Real-Time Operating System's Thread Context Switch

Date of Final Oral Examination: 10 May 2010

The following individuals read and discussed the thesis submitted by student Deepak Kumar Gauba, and they evaluated his presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Nader Rafla, Ph.D.	Chair, Supervisory Committee
Jennifer A. Smith, Ph.D.	Member, Supervisory Committee
James R. Buffenbarger, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Nader Rafla, Ph.D., Chair of the Supervisory Committee. The thesis was approved for the Graduate College by John R. Pelton, Ph.D., Dean of the Graduate College.

To my father…

ACKNOWLEDGEMENTS

I would like to thank my professors and colleagues at Boise State University for their support, guidance and encouragement. In particular, I would like to sincerely thank my advisor, Dr. Nader Rafla, for his valuable guidance and support while completing my graduate education. The thesis could never have been completed without him.

I would also like to thank Dr. James R. Buffenbarger and Dr. Jennifer A. Smith for being on my thesis committee, and guiding and encouraging me throughout my research work. I am very grateful to Dr. James R. Buffenbarger for his guidance and valuable suggestions during my research work, which helped me, finish my work on time.

Finally, I would like to thank my family for their unwavering support and encouragement. Thank you all.

ABSTRACT

Increasingly, embedded real-time applications use multi-threading. The benefits of multi-threading include greater throughput, improved responsiveness, and ease of development and maintenance. However, there are costs and pitfalls associated with multi-threading.

In some of hard real-time applications, with very precise timing requirements, multi-threading itself becomes an overhead cost mainly due to scheduling and context-switching components of the real-time operating system (RTOS). Different scheduling algorithms have been suggested to improve the overall system performance. However, context-switching still consumes much of the processor's time and becomes a major overhead cost especially for hard real-time embedded systems.

A typical RTOS context switch consumes 50 to 80 processor clock cycles (depending on processor architecture and context size) to store and restore the thread context. If a real-time application needs to respond to an event repeatedly less than this time, then the overall system performance may not be acceptable. The suggested approach in this thesis improves the context-switching time drastically. This technique has been implemented in hardware, as part of the processor state along with new central processing unit (CPU) instructions to take care of the context-switching process without interacting with external memory. With the suggested approach, the thread context-switch can be achieved in 4 CPU clock cycles independent of context size. This is a significant improvement to thread context switching.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 – INTRODUCTION

Context switching is a very important part of any multi-tasking operating system. In most hard real-time operating systems, running with time-critical applications, context-switching becomes an overhead due to its timing requirements. Two factors contribute to this overhead cost: direct and indirect. The direct cost of context switching includes moving contents of the Central Processing Unit (CPU) registers to and from external memory or cache. Indirect cost includes perturbation of cache, CPU pipeline, etc. [1]. In general, it is difficult to measure the total cost of context-switching. In the case of a hard real-time operating system (RTOS), with linear memory architecture, the direct cost of context switching constitutes a major part of the total context-switching cost. Many algorithms have been developed to reduce the direct cost of context switching [2, 3, 4]. These algorithms are either executed on specific high performance processors with cache or suggest examining the processor state and then deciding whether or not it is actually required to save the entire processor state/context [3, 4]. This latter approach is good for improving overall system performance. But if an application needs to save and restore the complete state frequently with hard real-time requirements, then this approach would not be effective. In addition, even if the context size is reduced, the basic context registers (like the program counter, stack pointer, global pointer, etc.) still need to be saved and restored. As context size changes from thread to thread, it would be difficult to design a deterministic system, which is another basic requirement of a hard RTOS. Since these

registers are being saved and restored to and from external memory or cache, this will consume some clock cycles depending upon the number of these registers.

This thesis is divided into two major components: hardware and software. The hardware component mainly involves the implementation of a number of register files to hold the operating system's thread contexts inside the processor and the development of the hardware support for new instructions to store and restore the contexts to and from the newly implemented register files. This concept is verified by actually implementing the register files in a Very high speed integrated circuit Hardware Description Language (VHDL) and executing the new CPU instructions to ensure the correct data movement.

The software component has been further divided into two sub-parts. The first software sub-part is the implementation of a small co-operative operating system that executes the threads in a round-robin fashion, and to develop test applications. These applications need to call the operating system's scheduler function whenever it needs to switch to the next thread. This co-operative operating system and test applications have been executed on a modified, as well as on a traditional, MIPS processor for a proof of concept and to measure the performance improvement.

The second software sub-part is the addition of newly implemented MIPS instructions to the MIPS assembler so that the correct executable file can be generated automatically to include the newly implemented context-switch instructions on the modified processor hardware.

1.1 Organization

This thesis is organized in seven chapters. Chapter 1 gives a brief introduction and outline. Chapter 2 explains the MIPS processor architecture and its instruction types; RTOS's basic design with its scheduler and context-switch components; and, co-operative operating system design. Chapter 3 provides the problem statement and a description of the proposed approach. Chapter 4 talks about the hardware implementation, which includes modifications in the MIPS processor architecture and details of newly implemented context-switch CPU instructions. Chapter 5 talks about software components, which includes details of the newly implemented co-operative operating system and MIPS assembler modifications. Chapter 6 explores and analyzes the test results generated from the hardware simulation and software test applications. Finally, Chapter 7 concludes the research and describes some future work based on this thesis.

1.2 Contributions of This Thesis

This thesis provides a review and an in-depth discussion of the different techniques and components required for the context-switching of a RTOS. It also provides a core framework design for context-switching implementation in the hardware along with the instructions used by the CPU for this purpose. A test operating system is developed to prove the proposed concept, which is expandable to general operating systems.

CHAPTER 2 – RTOS OVERVIEW

A typical real-time embedded system basically consists of two components: an embedded processor and a RTOS. To achieve a required performance, specially designed applications need to be developed as per the processor and RTOS. In this chapter, the architecture of a commonly used processor and the basic functionality of a RTOS are described.

For this research, a Plasma MIPS processor implementation [6] has been chosen for experimentation. The hardware implementation of the Plasma MIPS processor is done in VHDL for Xilinx© and Altera© Field Programmable Gate Array (FPGA) boards. The development work and implementation are carried out on a Xilinx Spartan-3E Starter kit board due to its availability.

2.1 Plasma MIPS Processor Architecture

The Plasma MIPS processor architecture is divided into two parts. The first part describes the hardware architecture, which deals with different modules, buses, and registers. The second part describes the CPU instruction formats and types of instructions.

2.1.1 CPU Hardware Architecture

Figure 2.1 shows a block diagram of the plasma MIPS processor architecture. The "*control"* module is the heart of this implementation. The "*mem_control*" module fetches

instructions from external memory at the address specified by the program counter (PC) register and sends it out in the form of a 32-bit code to the "*control"* module. The control module processes the code and creates a 60-bit Very Large Word Instruction (VLWI) code and sends it out to other processor modules such as "*Reg_Bank*", "*Bus_Mux*", "*PC_Next*", and "*MUL-ALU-SHIFT*" for further processing. Out of the 60-bit VLWI code generated by the "*control*" module, 26 bits are sent out to the "*Bus_Mux*" module, which includes 16 bits for immediate data "*imm_out*", 7-bit bus control signals, and 3 bits to indicate the type of branch instructions. The "*MUL-ALU-SHIFT*" unit performs the necessary arithmetic operations selected by the 16-bit signal "*op_select*" received from the "*control*" module.

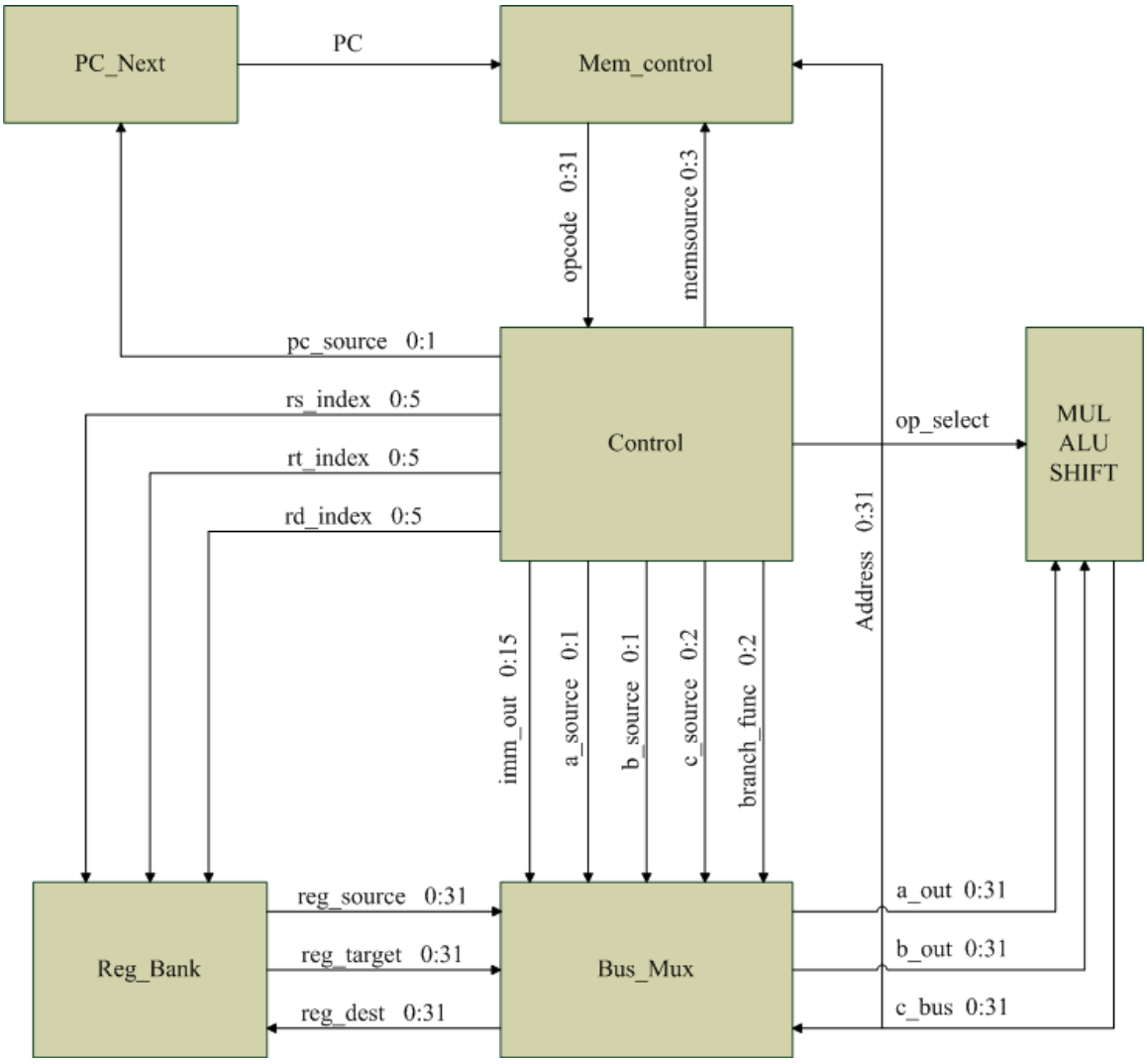

Figure 2. 1: Plasma MIPS Processor Architecture

The "*reg_bank*" module implements all the registers, including program counter (pc), stack pointer (sp), and global pointer (gp), as described in Table 2.1. In the case of the Plasma implementation, all registers in the "*reg_bank*" module have been implemented in the FPGA's block random access memory (RAM). So, only one register can be accessed at a time.

The "*control"* module also generates three sets of signals (rs_index, rt_index, and rd_index); each is 6 bits wide. These are used to access the registers in the "*reg_bank*" module. The "*reg_bank*" module outputs the requested data to the "*bus_mux"* module from the requested register.

Table 2. 1: MIPS Registers

Register	Register Name	Function
$0	zero	Always contains 0
$at	at	Assembler temporary
$2 - $3	v0 – v1	Function return value
$4 - $7	a0 – a3	Function parameters
$8 - $15	t0 – t7	Function temporary values
$16 - $23	s0 – s7	Saved registers across function calls
$24 - $25	t8 – t9	Function temporary values
$26 - $27	k0 – k1	Reserved for interrupt handler
$28	gp	Global Pointer
$29	sp	Stack Pointer
$30	s8	Saved register across function calls
$31	ra	Return address from function call
HI-LO	lo-hi	Multiplication/division results
PC	Program Counter	Points at 8 bytes past current instruction
EPC	epc	Exception program counter return address

2.1.2 MIPS Instruction Format

There are three types of basic instruction formats in the MIPS processor. These are defined as R-Type, I-Type, and J-Type. The R-Type instruction format is used to create instructions with register operations like load and store instructions; the I-Type instruction format is used to implement instructions that involve immediate data; and, the J-Type is used to implement jump or branch instructions. In the R-Type instruction format, the 6 high order bits (opcode) are 0 and the 6 low order ones (funct) define the function being performed by the instruction. The middle bits indicate indexes of different registers. This is detailed in Table 2.2.

Table 2. 2: Instruction Format for Instruction Type R

Instruction Format for Type R						
Bit Position	31 - 26	25 - 21	20 -16	15 - 11	10 - 6	5 - 0
Name	opcode	rs	rt	rd	shamt	funct

Table 2.3 shows the basic Type-I instruction format. The lower-order 16 bits are the immediate data and 6 high order bits (opcode) define the operation. Middle bits represent indexes of source and target registers.

Table 2. 3: Instruction Format for Instruction Type I

Instruction Format for Type I				
Bit Position	31 – 26	25 - 21	20 -16	15 - 0
Name	opcode	rs	rt	immediate data

Table 2.4 shows the Type-J instruction format where the lower-order 26 bits indicates the address and remaining bits (opcode) define the operation.

Table 2. 4: Instruction Format for Instruction Type J

Instruction Format for Type J		
Bit Position	31 – 26	25 - 0
Name	opcode	Address

2.2 Real-Time Operating System

In general, an operating system **(OS)** is responsible for managing the hardware resources of a computer and hosting applications that run on the computer. A **RTOS** is a specialized type of OS that performs different tasks, but is specially designed to run applications with very precise timing and a high degree of reliability. They are intended for real-time applications. Such applications include embedded systems such as programmable thermostats, household-appliance controllers, industrial robots, spacecrafts, and industrial-control and scientific-research equipment.

Furthermore, RTOS can be divided into two categories, **hard real-time** and **soft real-time** operating systems. In a **hard real-time** or **immediate real-time** operating system, the completion of an operation, after its deadline, is considered useless and this may cause a critical failure of the complete system and can lead to an accident. The ECU (**E**ngine **C**ontrol **U**nit) of a car and CNC (**C**omputer **N**umeric **C**ontrol) machine control are some of the examples of hard real-time systems. On the other hand, a **soft real-time system** will tolerate such lateness and may respond with decreased service quality.

Mobile-phone application and ink-jet printers are examples of a typical soft real-time system.

2.2.1 RTOS Functionality

A RTOS always contains multi-tasking, also known as multi-threading. Multi-tasking is a technique used for processor time allocation. Applications are divided into logical pieces commonly called threads and a kernel (core of the operating system) that coordinates their execution. A thread can be defined as an executing instance of an application and its context is the contents of the CPU registers and program counter at any instant of time. A register is a small fast memory inside a CPU (as opposed to the slower memory outside of the CPU) and is used to speed up the execution of programs by providing quick access to commonly used values, generally those in the midst of a calculation.

All threads in a hard RTOS are not equal. Some threads of an application have greater importance or priority than others. The high-priority threads must meet their deadlines; otherwise, the system may lead to a complete failure or a deadly accident. A scheduler, in the real-time operating system's kernel, schedules the threads based upon their priority. It also arranges a list of threads ready for execution, based upon their priority, and schedules them from the top of the list. There are many scheduler algorithms available to perform the scheduling activity efficiently and fairly. The most commonly used scheduler algorithm for RTOS is priority-based preemptive scheduling. This algorithm enables the scheduler to preempt the current running thread if a high-priority

thread becomes ready to execute. If a number of threads in the system have the same priority, then the scheduler will schedule these threads in a round-robin fashion.

2.2.2 Context Switching

Scheduling threads is done based upon their priority and the scheduler invokes the context-switching module of the OS. The context-switching module suspends the current running thread and starts executing the next eligible thread from the scheduler's ready queue. The context-switching activity can be described in slightly more detail as the kernel performing the following activities with regard to threads on the CPU:

1. Suspends the progression of one thread and store the CPU's state (context) for that thread somewhere in memory.
2. Retrieves the context of the next thread in the scheduler's ready list from memory and restore it in the CPU's registers.
3. Returns to the location indicated by the program counter (the point at which the thread was suspended in an earlier context switch) in order to resume the thread's execution.

Accordingly, the context switch can be described as the kernel suspending execution of one thread on the CPU and resuming execution of some other thread of a higher or same priority. Context switching is an essential feature of multi-tasking operating systems. So by definition, a multi-tasking operating system is one in which multiple threads execute on a single CPU seemingly simultaneously and without interfering with each other. This illusion of concurrency is achieved by means of context switches that are occurring in rapid succession (tens or hundreds of times per second).

Context switching occurs as a result of threads voluntarily relinquishing their time in the CPU or as a result of the scheduler making the switch when a process has used up its allocated CPU time slice. A context switch can also occur as a result of a hardware interrupt, which is a signal from a hardware device to the kernel indicating that an event has occurred.

2.2.3 Co-Operative Operating System

A hard real-time operating system cannot be designed without priority-based preemptive scheduling. On the other hand, soft real-time systems can be designed using round-robin scheduling in which all threads are scheduled in round-robin fashion and threads relinquish their CPU time voluntarily after reaching a logical end of the task or after executing for a fixed amount of time. Operating systems with this type of scheduler design are called co-operative operating systems, as threads cooperate with others running on the system.

2.3 Summary

RTOSs are the most common component of today's typical real-time embedded systems. RTOSs use multi-threading to share the CPU time to achieve multi-tasking. The scheduler module of a RTOS does the required context switching to achieve a specified CPU time sharing among all threads. There are performance issues with the tradition context- switching implementations. Those issues, and a possible solution, will be discussed in the following chapters.

CHAPTER 3 – PROBLEM STATEMENT AND SOLUTION

As described in the previous chapter, context switching is an important part of any multi-tasking OS. It is computationally intensive because it requires considerable processor time. Thus, context switching represents a substantial cost to the system in terms of CPU time. The cost of context switching goes even higher for a hard real-time system, as that makes it difficult to meet the thread's deadline.

3.1 Cost of Context Switching

Research has been previously done to measure the cost of context switching in general [1]. The direct cost of context switching includes saving the CPU register data to/from external memory or cache and indirect cost includes perturbation of cache, CPU pipeline, etc [1]. That makes it difficult to measure the total cost of context switching. For a typical hard real-time system, with linear memory architecture, the direct cost of context switching constitutes the major part of the total context-switching cost. That is why many algorithms have been implemented to reduce the direct cost of context switching [2, 3, 4].

The direct cost of context switching depends upon the CPU architecture and the OS design. It is directly proportional to the number of CPU temporary (scratch) registers to be saved and restored, and the OS design. For example, in the case of the Advanced RISC Machine (ARM) processor, there are sixteen temporary registers (including program counter), apart from one status register, which are required to be saved and

restored during the context switching [1]. Similarly, for the MIPS processor, there are eleven registers that need to be saved, apart from program counter. So, context-switching time varies from processor to processor.

Similarly, a number of commercially available RTOSs like VxWorks, ThreadX, and QNX claim different context-switching times in terms of micro-seconds for different processors. Because of these variables, it is difficult to measure the total context-switching time in general. That is why, for this research, measurement of context-switching time is presented in terms of clock cycles.

To explain context-switching overhead, a hypothetical application that requires frequent context switching in a small amount of time is considered. Assume a real-time system with three threads; A, B and, C. Thread ‘A’ reads and samples input data from an analog input; thread ‘B’ processes each sampled datum and generates some control signals; and, thread ‘C’ generates the output signals based upon the control signals generated by thread ‘B’. The sequence of operations is described as follows:

1. Thread ‘A’ reads analog input data and releases CPU control by issuing an OS system call.
2. The Kernel saves the context of thread ‘A’ somewhere in external memory by copying CPU temporary registers, the stack pointer, the program counter, etc. These registers are saved individually in sequence as the CPU is only able to generate one address at a time.
3. The Kernel restores the context of thread ‘B’ from the external memory, by restoring CPU temporary registers, the program counter and the stack pointer in

sequence from external memory. After restoring the context, thread ‘B’ starts executing and processing the received sample and then generates the control data. Finally, thread ‘B’ releases the CPU control by executing a system call.

4. Now, the kernel saves the context of thread ‘B’ and restores the context of thread ‘C’ to and from the external memory, respectively. Thread ‘C’ then sends out the control data and releases the CPU control to thread ‘A’ to read the next sample.

If input analog data must be sampled at a higher frequency rate, and each sample is very important, then the context switch poses a large overhead, as each context switch would consume many CPU cycles. A typical context switch consumes 50 – 80 clock cycles and if a system needs to respond to an event in less than context-switching time, then that can be done by implementing the event response in an interrupt service routine (ISR) and useing an interrupt-driven system instead. But if those events are happening continuously, then the overall performance of the system will be degraded tremendously, as most threads may not get a chance to execute.

Since the registers are being saved in external memory, only one register could be saved at a time. Consequently, for N registers to be saved and restored, this approach would take at least 2 x 2 x N CPU clock cycles for a complete context switch. Practically, during these many clock cycles, the CPU normally remains idle, not doing any work assigned by the applications. That reduces the overall efficiency of the system.

3.2 Current Approaches

To improve responsiveness, the context-switching time needs to be reduced. Many software and hardware based solutions have been proposed to reduce context-

switching time [2, 4, 5]. The software-based solutions mainly reduce the average context-switching time by reducing the context size. The approach suggested by Xiangrong Zhou and Peter Petrov suggests achieving a low cost context-switching by using compiler, micro-architecture, and an OS kernel [4]. This technique identifies the switch points in the executing code, at compile time, at which a minimum number of context registers needs to be saved. If an interrupt occurs and the system needs to do a context switch, then the scheduler defers the context switch untill the execution reaches the next switch point where less context-registers need to be saved. The compiler identifies switch points and also provides the custom software routines to kernel for context switching. The authors used an example to explain the approach, as shown in Figure 3.1[4]. If an interrupt occurs at time *t1*, then all registers need to be saved as all registers are live at that point [4]. But if the context switching is deferred untill time *t2*, then only 3 registers need to be saved, as shown.

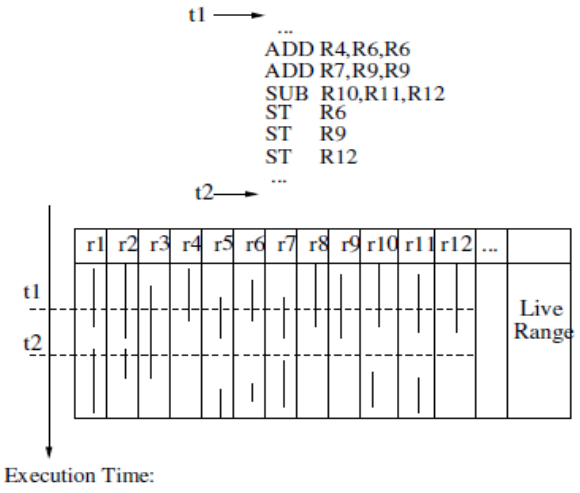

Figure 3. 1: Context Switching using Variable Context Size [4]

The good part of this approach is that the system is doing the actual task between time *t1* and *t2*, but the duration between *t1* and *t2* is not fixed and can be longer as interrupt is an asynchronous event and a RTOS thread may miss a deadline. That can be taken care by a good RTOS kernel design but that would lead to a complex and non-deterministic system.

The other software-based approach is for a specific RTOS but can be implemented for any RTOS. The approach suggested by Zhaohui Wu, Hong Li, Zhigang Gao, Jie Sun, and Jiang Li, creates new thread states, and based upon the thread's state, the scheduler decides if context save or restore is actually required [2]. It helps in reducing the average context-switch time of the system.

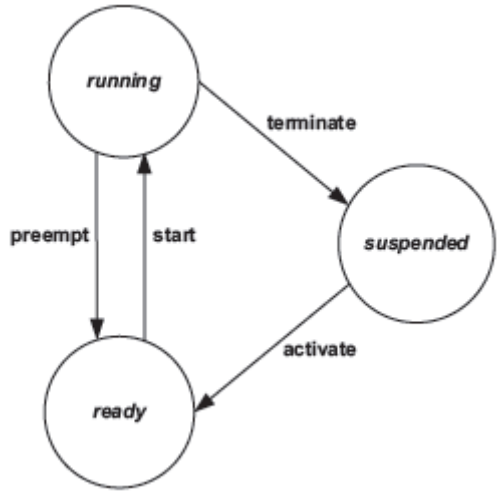

Figure 3. 2: Thread State Diagram of OSEK Operating System [2]

Figure 3.2 shows the thread's state diagram of the OSEK OS. The original OSEK OS thread's state diagram shows three states. If a thread is waiting for a resource or if a

thread is terminated, then it goes to a *suspended* state. The authors suggested one new state called *waiting* state, in which the thread goes into waiting state if the thread is waiting for a resource, and it goes into suspended state when it is being terminated. The thread's *ready* state is further divided into two states: *initial* and *intermediate*.

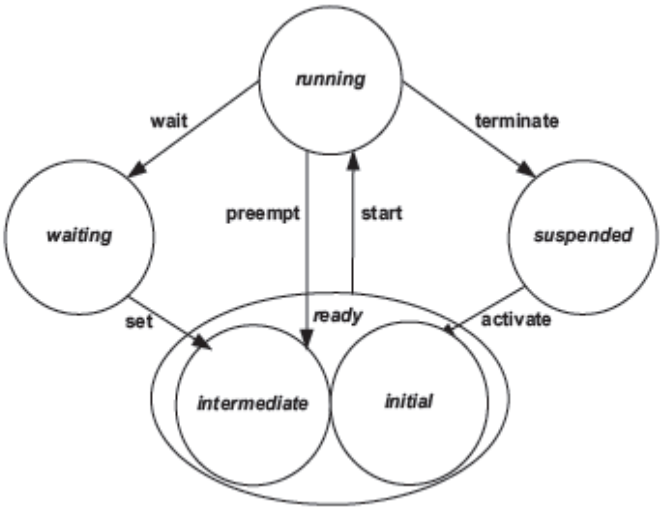

Figure 3. 3: Modified Thread State Diagram of OSEK Operating System [2]

As shown in Figure 3.3, when a thread becomes active in the suspended state, it will be in the *initial* state, and when a thread becomes ready from the waiting state, it will be in the *intermediate* state. When a thread's state changes from a running state to a *suspended* state, then there is no need to save the context as in that case the thread gets terminated. On the other hand, when a thread's state changes from *suspended* to *ready*/*initial* state, then there is no need to restore the context as thread would start executing from the beginning. This approach improves the average context-switch time

and thus the overall throughput of the system. As the context-switching time would be reduced in some cases, designing a deterministic system with this approach would lead to a very complex design.

Current hardware-based solution use reconfigurable hardware. Research by Hyden Kwok-Hay So, at Berkeley University, uses BORPH (Berkeley Operating system for ReProgrammable Hardware) OS [5]. This operating system is specially designed for FPGA-based reconfigurable hardware. The BORPH kernel supports FPGA applications similar to conventional OS support for programs. The FPGA resources are managed by a kernel similar to other system resources.

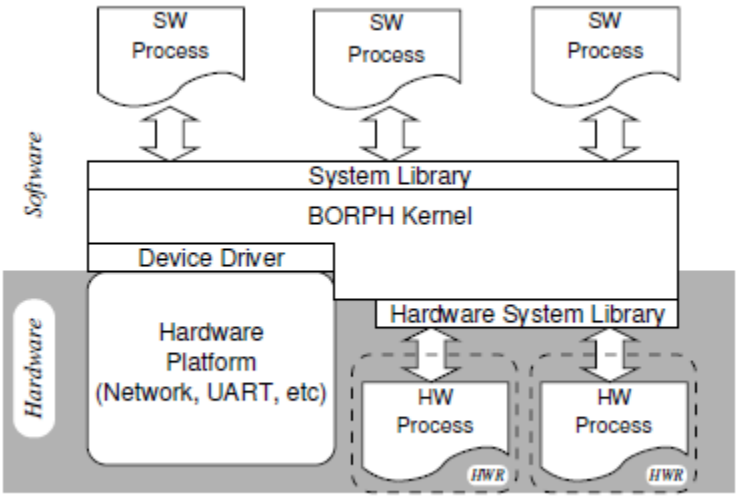

Figure 3. 4: Modified Thread State Diagram of OSEK Operating System [5]

The BORPH is a UNIX-based OS with a virtual file system that allows users to communicate with a running gateware design through UNIX file system access. Figure

3.4 shows the basic block diagram of this reconfigurable hardware-based approach. Since different processes can be created in hardware, which can execute in parallel and keep their state, there is no need of real context-switching for the processes running in the hardware. The OS tasks that require fast context-switching can be implemented as hardware processes to avoid context-switching overhead. This approach is good for hard real-time systems and this would provide good throughput, and also would be deterministic. But this approach would take lot of hardware resources and it would be difficult to design and maintain such a system.

3.2 Proposed Solution

The proposed approach reduces the context-switch time drastically to a fixed number of clock cycles independent of the number of context registers, because the context is saved in a newly created context register file. These context-register files are implemented in the processor hardware itself, as a part of the register bank module. The proposed modified MIPS processor architecture is shown in Figure 3.5.

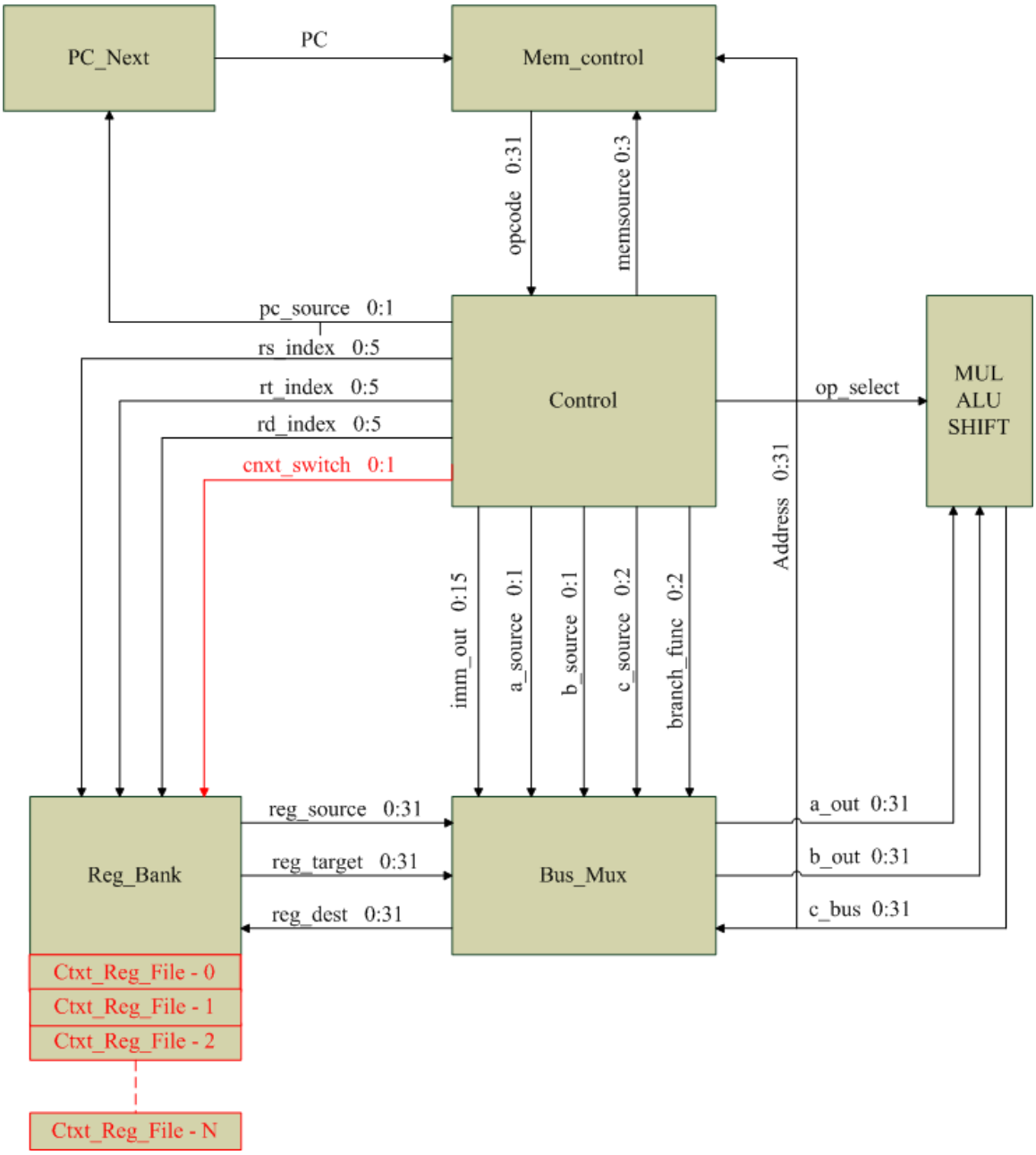

Figure 3. 5: Modified MIPS Processor Architecture

To save and restore a thread's context in the internal register files, two new CPU instructions have been implemented in the processor hardware. Additional software needs

to be developed to use these instructions to exploit the suggested hardware design. Since new instructions are being added to the processor architecture, the MIPS assembler has to be modified to support the new instructions.

3.3 Summary

Context switching is a major bottleneck for hard RTOS-based systems, especially for applications that require frequent context switching, and have stringent deadlines for different threads. Many software approaches try to reduce context-switching overhead, but this leads to making the system non-deterministic, although a deterministic system is one of the basic requirements of a hard RTOS.

As per the suggested approach, the context-switching module of the RTOS is implemented in the processor hardware. That does not only reduce context switching but also makes the system more reliable. The hardware implementation of the suggested approach is described in detail in the next chapter.

CHAPTER 4 – HARDWARE DESIGN AND IMPLEMENTATION

Hardware implementation for the proposed architecture is divided into two parts. The first part describes the implementation of context register files to save the CPU state in the processor itself and the second part describes the hardware implementation of save-context (*scxt*) and restore-context (*rcxt*) CPU instructions.

4.1 Register File Design

The Plasma MIPS processor, used for this thesis, implements the *"reg_bank"* module in the FPGA's block RAM [9]. This design won't work with the suggested approach, as all the context registers must be saved on a register file in one CPU clock cycle, and that cannot be achieved if registers are implemented as RAM locations. To achieve this task, the original Plasma MIPS design is modified by implementing all the "*reg_bank"* registers in FPGA's logic blocks. This design requires more FPGA logic resources but provides fast access to registers as compared to the original design.

To prove this concept, only 4 register files are implemented in the "*reg_bank"* module. These register files are indexed from 0 to 3. Each register file can hold up to 12 registers, which is the size of a thread's context for the MIPS architecture. A thread's context includes 9 saved or temporary registers ($16 - $23 and $30), the stack pointer register ($28), the global pointer register ($29), and the link register ($31). As previously shown, in Figure 3.5, new register files have been added in the "*reg_bank"* module to save the context registers. Their registers are initialized to 0 at reset.

The number of register files can be extended, depending on the available FPGA resources, to accommodate more threads. To access the register files, two context-switch instructions have been implemented in the hardware. The design of these context-switch instructions is described in the next section, and the VHDL source code for the *"reg_bank"* module is placed in APPENDIX A-1.

4.2 Context-Switching Instruction Design

The "*mem_control"* module of the processor fetches 32-bit machine-code instructions from the memory and passes them to the "*control"* module for further processing. The control module generates a 62-bit VLWI instruction code, which includes two additional bits for the "*cnxt_switch"* signal. As shown in Figure 3.5, the 2-bit signal (*cnxt_switch* 0: 1) is sent out to "*reg_bank*" module that processes "*scxt*" and "*rcxt*" context-switch instructions, based upon the "*cnxt_switch"* signal value. Table 4.1 shows the "*cnxt_switch*" signal's bit map.

Table 4. 1: The *cnxt_switch* Signal Bit Map

Instruction	**cnxt_switch**	
	Bit 1	Bit 0
scxt	0	1
rcxt	1	0
Other instructions	0	0

Before executing the "*scxt*" and "*rcxt*" instructions, software needs to save the index of the register file in any temporary register (*$t*), and then execute these instructions with the corresponding *$t* register as an operand of the instruction. For example, if temporary register *$4* contains the index of the register file in which the

context needs to be saved, then the instruction to save the context would be "*scxt $4*"; and, similarly, if register *$4* contains the index of the register file from which the context needs to be restored, then the instruction to restore the context would be "*rcxt $4*".

4.3 Context-Switching Instruction Hardware Implementation

As discussed in Chapter 2, three formats of MIPS processor instructions are available. Since we need to store the index of the register file in a temporary register and pass the register as an operand of the instruction, the newly implemented context-switch instructions should be in instruction format Type-R. Table 4.2 shows the bit map designed for the "*scxt*" instruction where register *"rt"* contains the index of the register file in which the current context needs to be saved.

Table 4. 2: Save Context Instruction Bit Map

Save Context Instruction (scxt rt)						
Bit Position	31 - 26	25 - 21	20 -16	15 - 11	10 - 6	5 - 0
Name	opcode	rs	rt	rd	shamt	funct
Bit Values	000000	00000	00001 - 10111	00000	00000	111100

Similarly, Table 4.3 shows the designed bit map for the "*rcxt*" instruction. Again, register *"rt"* contains the index of the register file from which the next thread's context needs to be restored.

Table 4. 3: Restore Context Instruction Bit Map

Restore Context Instruction (rcxt rt)						
Bit Position	31 - 26	25 - 21	20 -16	15 - 11	10 - 6	5 - 0
Name	opcode	rs	rt	rd	shamt	funct
Bit Values	000000	00000	00001 - 10111	00000	00000	111101

The software must set the register *"rt"* correctly, by setting the operand value within the range of available register files, before executing these instructions. Since we have implemented four register files to save contexts, if the value in register *"rt"* is set out of range (greater than 3), then these instructions would be executed as a No Operation (NOP) instruction and would not harm or change any thread's context data. This will ensure the overall system's continuity.

4.4 Hardware Synthesis and Implementation

The Plasma MIPS VHDL implementation for Xilinx Spartan-3E FPGA board is chosen to implement the suggested approach [9]. The implementation and verification is done using ISE version 10.1 and ModelSim XE III 6.3C, respectively. As discussed, the "*control*" and "*reg_bank*" modules are modified to implement the suggested MIPS processor architecture. The context-switch register files have been implemented in the "*reg_bank"* module to the maximum capacity of the available FPGA resources. With four register files, the suggested MIPS architecture has consumed 99% of the slices and 89% of the 4-input LUTs (Look-Up Table) of the FPGA resources. This addition has reduced the maximum speed of the processor from 27.421 MHz to 25.716 MHz. To accommodate the four context-switch register files and control logic along with the

original MIPS processor in the FPGA, the "*Ethernet*" peripheral module of the original Plasma MIPS implementation has been removed from the design, as this peripheral is not a part of the processor design, nor needed for our intended usage. Table 4.4 compares the FPGA resource usage between the original and proposed MIPS architecture, respectively.

Table 4. 4: FPGA Device Resource Usage for Original Plasma MIPS Architecture and Modified MIPS Architecture

Device Resource Type	**Total**	**Device Utilization for Original Architecture**		**Device Utilization for Modified Architecture**	
		Used	**Utilization**	**Used**	**Utilization**
Number of Slice Flip Flops	9312	783	8%	3017	32%
Number of 4 input LUTs	9312	3754	40%	8329	89%
Number of occupied Slices	4656	2066	44%	4654	99%
Minimum Period		36.468 ns		38.886 ns	
Maximum Frequency		27.421 MHz		25.716 MHz	

For simplicity, the system has been implemented as XIP (execute in place), as the processor reads the software instructions directly from the FPGA's block RAM and executes. To place an executable in block RAM, specially designed tools ("*convert_bin*" and "*toimage*") have been used [9]. The *"convert_bin"* tool reads the MIPS executable

file and generates a text file containing MIPS hexadecimal instructions in ASCII (American Standard Code for Information Interchange) format. Then, the "*toimage*" tool reads the MIPS 32-bit instructions from the text file, breaks them into four 8-bit segments, and places them in the block RAM source file ("*ram_image.vhd"*) at appropriate locations. This VHDL file is used to synthesize and implement the processor design, using Xilinx ISE tools, with the software executable in the FPGA block RAM.

4.5 Summary

To reduce context-switching time, register files have been implemented in the processor hardware to store the CPU state of a thread. Special instructions have been implemented to allow the access of these register files through software. Details of this hardware and instruction implementation have been described in this chapter. Software needed to exploit this newly implemented processor feature is explained in the following chapter.

CHAPTER 5 – SOFTWARE DESIGN

Software design for this research can be divided into two parts. The first part describes the design and development of a co-operative operating system, which uses the newly implemented CPU instructions for context switching, and three test applications to measure the performance improvement. The second part deals with the modifications in the GNU MIPS tool-chain to support "*scxt*" and "*rcxt*" context-switch instructions. For this thesis, these instructions are added in the GNU-MIPS assembler.

5.1 Co-Operative Operating System Design

To prove the validity of the suggested approach and measure the performance improvement, a small basic co-operative OS for the modified MIPS processor has been developed. This OS provides interface functions for application development mainly to initialize, create, and schedule different threads. Table 5.1 lists the interface functions.

Since it is a co-operative OS, each thread does its allocated task and releases the thread's control voluntarily by calling the OS interface function "*schedule"*. This function does the actual context switching by saving the current thread's context, and then restoring the context of the next thread in the queue. The OS supports context switching using external memory, as well as internal register files.

Table 5. 1: Operating System Interface Functions

Function Name	Description
initOS( )	This function initializes the Task structure for each thread in the system to the default values.
CreateThread( )	This function initializes the Task structure of the thread for the thread's requirements. It takes parameters as follows; *TaskID* - Thread ID *funcptr* - Pointer to thread's starting function *FastCtxtSwitch* - Context switching property setting of the thread
Schedule( )	This function schedules the next thread in the queue

5.1.1 Operating System Design

The current implementation of the OS supports four threads but can be easily extended as needed. These threads are initialized by the application. The application needs to call "*InitOS*" to initialize each thread's "*Task*" structure. Figure 5.1 shows the operating system's "*Task*" structure. The "*FastCtxtSwitch*" member of the "*Task*" structure identifies the context-switching property of a thread. If "*FastCtxtSwitch*" is set to a value greater than 0, then the OS save/restore the context of that thread to/from internal register files, respectively. Otherwise, external memory is used by the context switching for that particular thread. This feature is required for applications with more than four threads. In that case, an application can decide whether a thread needs fast context switching or not. For our experimentation, this feature is used to compare the

performance between the traditional context-switching approach and the suggested approach.

```
typedef struct Task
 {
     void (*TaskPtr)();       // Pointer to Thread Starting Function
     int *State;              // context
     unsigned char Executed;  // 1 – thread has started, 0 otherwise
     unsigned char TaskID;    // Task ID
     unsigned char FastCtxtSwitch;  // 1 – Require fast context switch,
                                    // 0 otherwise
 }Task;
```

Figure 5. 1: Task Structure

At the time of system initialization, all four threads are initialized with "*FastCtxtSwitch*" to 0. To achieve the context switch using internal register files, an application has to set the "*FastCtxtSwitch*" member of the "*Task*" structure, for that thread, to a value greater than 0. So, it is the application's responsibility to use the context-switching method carefully, based upon the the thread's requirements and the number of internal register files available in the processor hardware.

5.1.2 Operating System Operation

After initializing the OS by calling "*InitOS*", the application needs to call the "*createTask*" OS function to create different threads. While creating threads, the application needs to pass parameters: a thread identification number, a pointer to the thread's starting function, and the context-switching property of the thread. After creating

all threads, the application needs to call the "*schedule*" OS function to start executing threads.

The "*schedule*" function schedules threads for execution in round-robin fashion. It picks the next thread in the queue and calls the thread's starting function if the thread is going to be executed for the first time. Otherwise, it saves the context of the current thread and restores the context of the next thread in the queue. Again, if the restored thread is going to execute for the first time, then the scheduler calls the thread's starting function. Otherwise, it restores the context of the next thread from the internal register files or from external memory depending on the "*FastCtxtSwitch*" value of the thread's task structure.

After completing the assigned task, a thread must call "*schedule*" to release the CPU control voluntarily. Figure 5.2 shows a detailed flow chart for our co-operative OS. The source code is listed in APPENDIX B-1 to APPENDIX B-5.

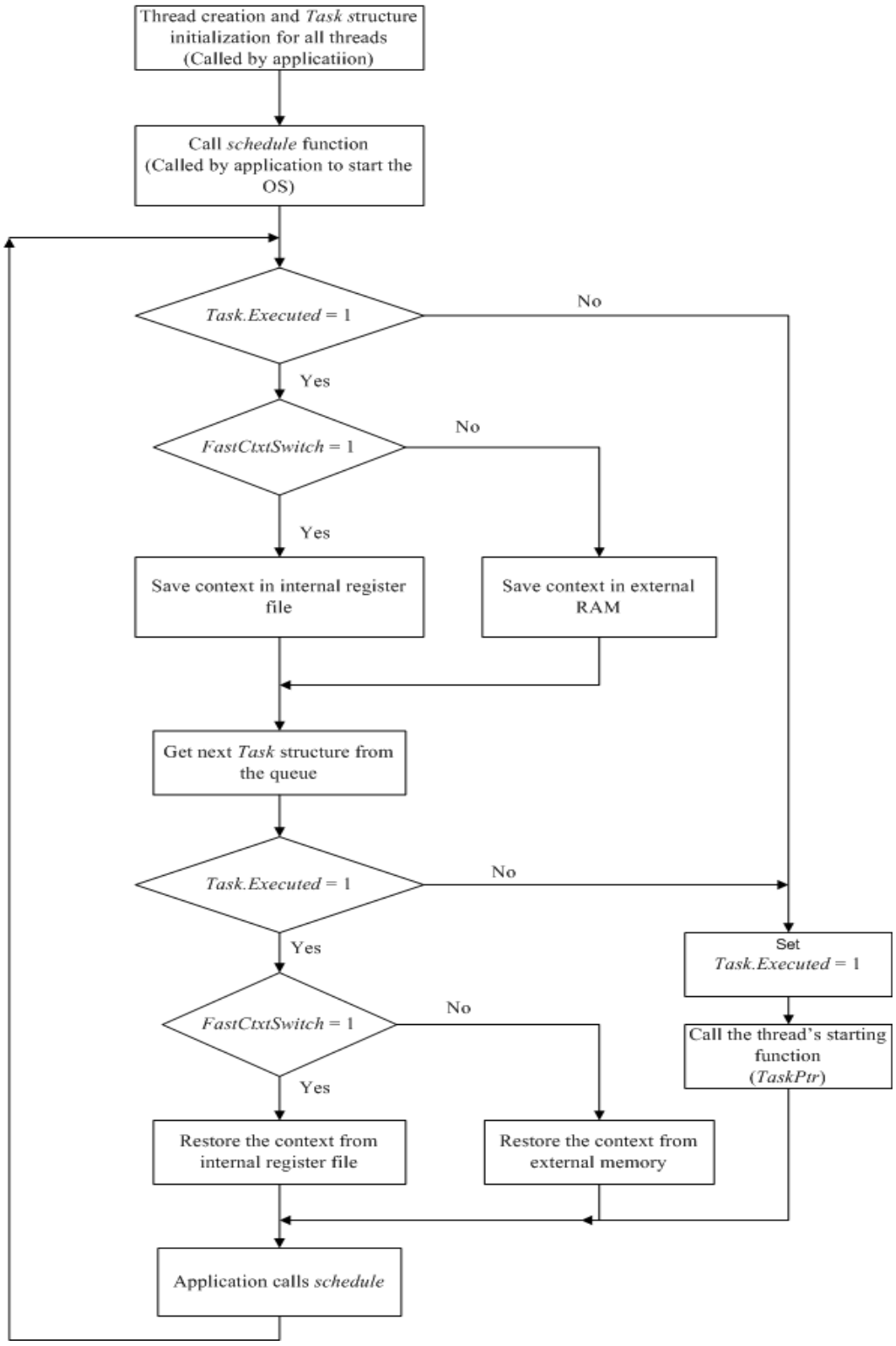

Figure 5. 2: Co-Operative Operating System's Flow Chart

5.2 Assembler Modifications

The GNU tool chain for the MIPS processor is used to compile the co-operative OS and the test applications. To automate the build process, the newly implemented context- switch instructions are added to the GNU MIPS assembler.

These instructions are added to GNU "*binutils*" version 2.19 [9]. The *"binutils-2.19/gas"* (GNU assembler) folder contains the source code for the MIPS assembler. The file "*mips-opc.c*" in "*binutils-2.19/opcode*" contains all the instructions supported by the MIPS processor. The new instructions have been added in the file "*mips-opc.c*".

The GNU MIPS assembler is implemented in the 'C' programming language. Figure 5.3 describes the basic structure of MIPS instructions implementation in the assembler.

```
const struct mips_opcode
{
      name, args, match, mask, pinfo, pinfo2, membership
}
```

Figure 5. 3: MIPS Instruction Structure Format in GNU Assembler

The "*mips-opc.c*" file defines an array of *"mips_opcode"* structures and each array element contains one MIPS machine OP-code. Table 5.2 describes the "*mips-opcode*" structure elements and Table 5.3 describes the "*scxt*" and "*rcxt*" instruction implementation in the array of *"mips_opcode"* structures.

Table 5. 2: *"mips_opcode"* Structure Data Members

Structure Member	Description
name	Instruction string e.g. "add"
args	A string describing the arguments to the instruction.
match	Match hex value of the instruction
mask	Bit mask of the instruction
pinfo	A collection of additional bits describing the instruction.
pinfo2	Additional bits describing the instruction.
membership	MIPS version information

Table 5. 3: 'scxt' and 'rcxt' Instruction Assembler Values

Structure Member	Instruction scxt values	Instruction rcxt values
name	"scxt"	"rcxt"
args	"t"	"t"
match	0x0000003c	0x0000003d
mask	0xffffffff	0xffffffff
pinfo	RD_t	RD_t
pinfo2	0	0
membership	1	1

A message is also added to the assembler source code that gets printed on the screen when using the modified assembler.

5.3 Software System Implementation

The software components developed in this thesis have been implemented in ‘C’ and the MIPS assembly programming language. The source code for the co-operative OS and test applications are compiled using the GNU MIPS tool-chain under cygwin environment on a Windows-based computer.

To debug the software, debug messages are added. The debug messages are sent to the Universal Asynchronous Receiver Transmitter (UART) serial port. The terminal program on the computer connected to the Xilinx FPGA board through serial cable receives and displays these messages on a computer screen. The same debug serial port is used to send the test application’s results for analysis.

5.4 Summary

A multi-threaded OS is required to test the suggested approach of context switching. A small co-operative OS that supports four threads has been implemented. The context switching between these threads can be achieved using internal register files as well as external memory based on the context-switching property setting in the “*Task*” structure of that thread. The GNU MIPS assembler also has been modified to support the newly implemented context-switching instructions.

To measure the performance improvement with the suggested approach, test applications are required. These applications execute on top of the implemented OS and

exploit the context-switching features supported by the OS and the processor. The next chapter describes these test applications and analyzes the test results.

CHAPTER 6 – EXPERIMENTAL RESULTS AND ANALYSIS

This chapter describes the verification process of the hardware implementation of the proposed approach. It also explains the software used for testing the complete system. The performance of the suggested approach is also evaluated and the results are compared with traditional context-switching methods.

6.1 Hardware Verification

To verify the correct operation of the context-switch instructions, software using the "*scxt*" and "*rcxt*" instructions was developed in the MIPS assembly language and executed on the modified processor in a simulation environment. This verification software first initializes all nine temporary registers of the thread's context with values from 1 to 9. Register $4 is then set to 2, which is the index of the register file in which the context will be stored, and then the "*scxt*" instruction is executed. It is expected that the instruction should move all the contents of the context registers to context register file 2 (*ctxt_reg2*) in 2 clock cycles. Simulation for the "*scxt $4*" instruction verified this expectation as shown in Figure 6.1

To verify the "*rcxt*" instruction, the values 2 through 10 are stored in the 9 temporary registers of the processor. Then, the "*rcxt*" instruction is executed with register $4 value set to 2, indicating the context needs to be restored from the context-register file 2. The contents of register file 2 are moved into the CPU context registers and the previously saved context is restored in 2 clock cycles. Figure 6.2 shows the simulation

waveform for the "*rcxt $4*" instruction. As expected, the values of the correct context registers replace the previous context.

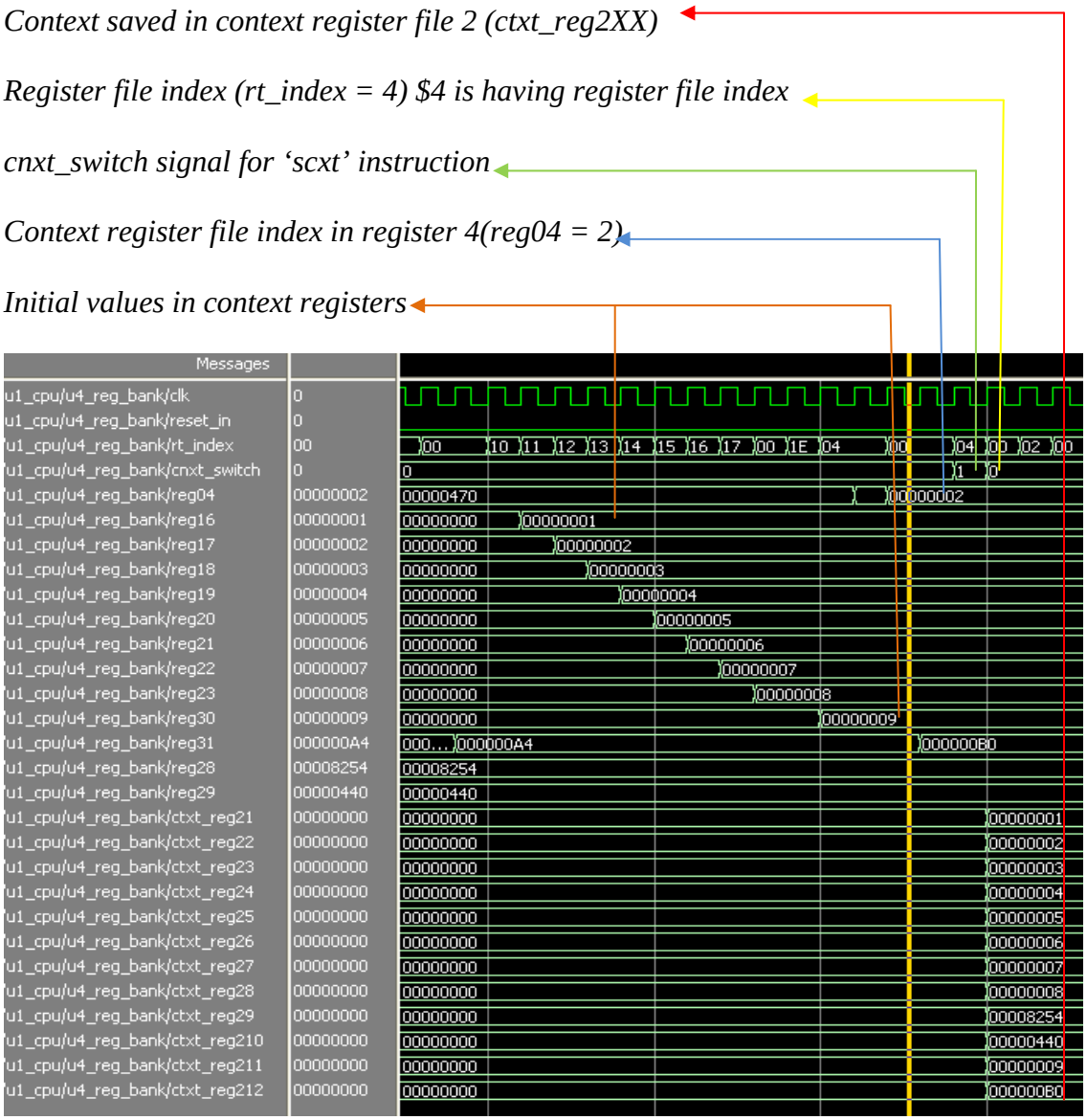

Figure 6. 1: Waveform for 'scxt $4' Instruction

Context restored from context register file 2 (ctxt_reg2XX)

Register index (rt_index = 4) that is holding register file index

"cnxt_switch" signal for 'rcxt' instruction

Context register file index in register 4(reg04 = 2)

Initial values in context registers

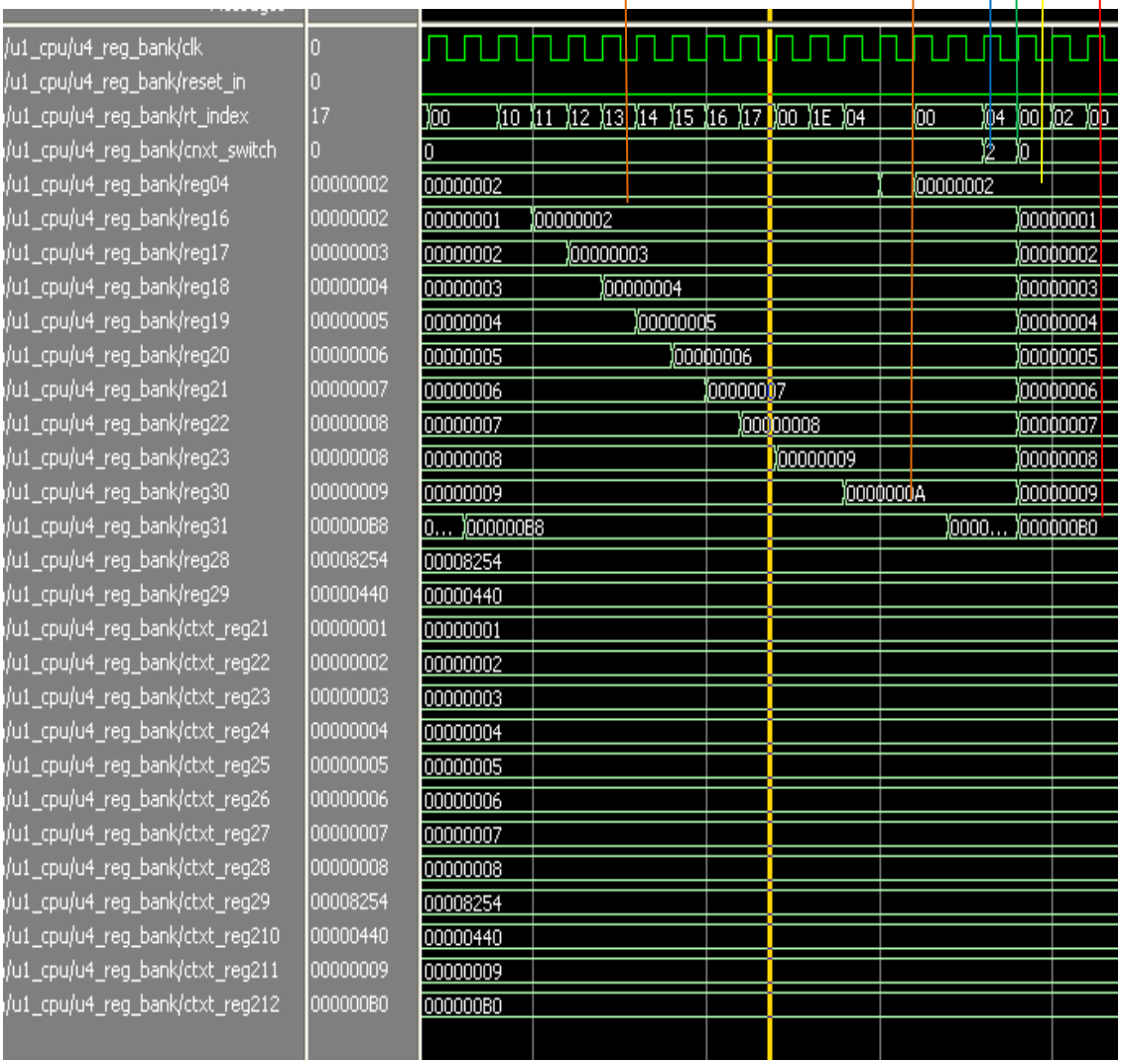

Figure 6. 2: Waveform for 'rcxt $4' Instruction

As discussed in Chapter 4, if the context-switch instructions executed with the operand register have an out-of-range value, then these instructions are executed as NOP instructions and do not change context state. A test case is designed to verify this as folows: register $4 is initialized with the value 1 and register $5 with 7 before executing the *"scxt"* instruction. These instructions are then executed in sequence. Since the value 7 is loaded into register $5, the operand of the "rcxt" instruction, is out of range. No change in context registers is expected for this test case. Figure 6.3 verifies this functionality. The *"rcxt"* instruction (0x0005003D) is executed to restore the context from register file index 7 as specified in register $5. After the execution of this instruction, there is no change in the context registers.

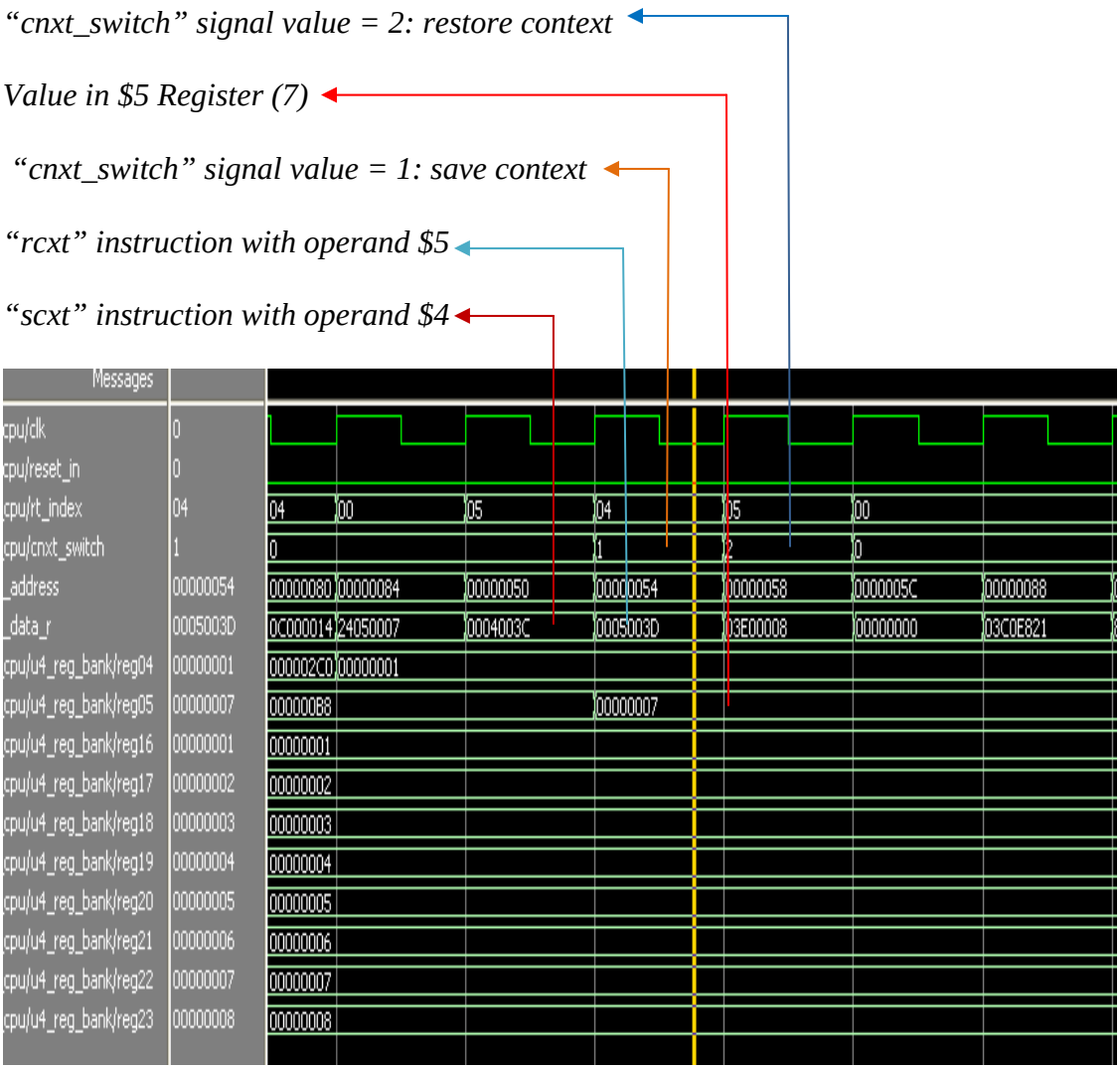

Figure 6. 3: Waveform to Verify an Out-of-Range Instruction Operand

To determine the number of clock cycles consumed by a complete context switch, a test program is implemented and executed on the modified MIPS processor soft-core in the simulation environment. This program initializes the CPU context registers with known non-zero values and then executes "*scxt*" and "*rcxt*" instructions in sequence. The "*scxt*" instruction saves the CPU context in register file 1 and the "*rcxt*" instruction restores the context from register file 2. As shown in the simulation output in Figure 6.4,

the *“scxt”* and *“rcxt”* instructions consume 2 clock cycles each to store and restore the context in the register file 1 (ctxt_reg 1) and from register file 2 (cntxt_reg 2), respectively. So the complete context switch takes place in 4 clock cycles. This value is independent of the number of registers used by the context.

Figure 6.4 also shows that the scratch registers (reg16 to reg23) are initialized with values 1 to 8, respectively, and context-register file 1 is initialized with 0s. The figure also shows that register 4 is initialized with the value of 1, the index of the context-register file. After executing the *“scxt”* instruction (0x0004003C) the context is saved in register file 1, as expected. The *“rcxt”* instruction (0x0005003D) is executed next showing that the register-file index from which context needs to be restored is saved in register 5. Since register 5 is initialized with the value 2, the context needs to be restored from the register file 2 that was initialized with 0 at reset. The figure shows that the context registers are loaded with 0s after *“rcxt”* execution.

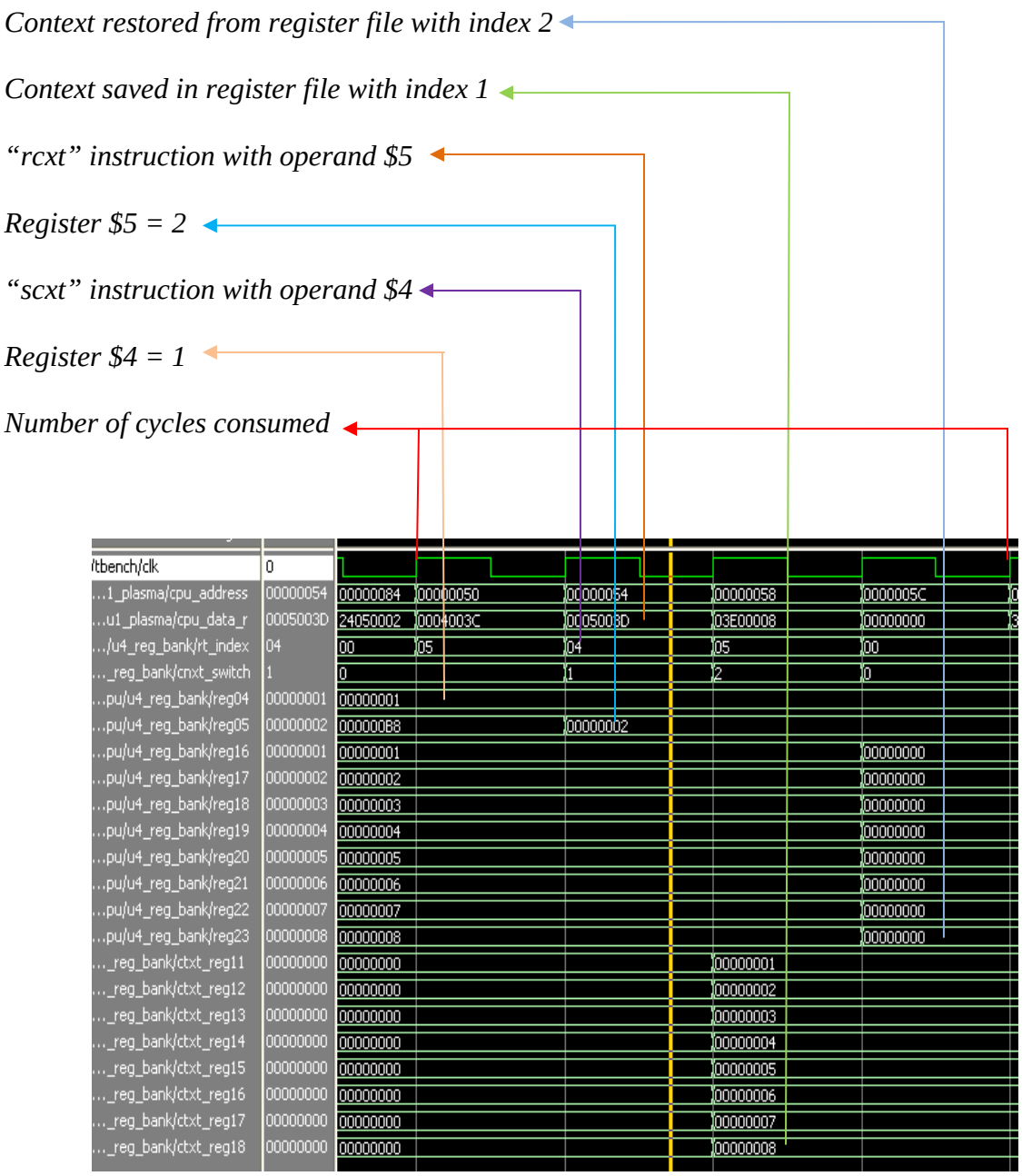

Figure 6. 4: Context Switch Instructions Waveform

6.3 Test Applications

As discussed earlier, it is difficult to measure the actual cost of context switching due to variables like processor speed, processor architecture, RTOS design, and etc. So, the actual cost of context switching may vary among different systems. The overall impact on system performance, due to context-switching overhead, also depends on the type of application. If an application requires frequent context switching, then the system will spend more time in managing and switching the threads and that will degrade overall performance.

For this thesis, three test applications have been implemented, which require frequent context switching, and each application tests and measures the different aspects of the suggested approach. These test applications use the interface functions provided by the co-operative OS to access the proposed hardware features. Each application implements four threads and each thread is running in a never-ending loop. Each thread executes a specific task by manipulating global variables in a loop and calls the operating system's "*schedule*" function to release control to the next thread. These applications are designed to test the functionality and measure the performance improvement of the proposed approach. The following sections describe the test applications and their results in detail.

6.3.1 Test Application – 1

This application tests the successful operation of the proposed approach by switching four threads using internal register files. This test is used to ensure that data

between threads is not corrupted and thread's context switching is correct. The flow chart of this application is shown in Figure 6.5.

In this application, the first thread, with *TaskID=0*, assigns/modifies four global variables and calls the OS function "*schedule*" to voluntarily release CPU control. This is analogous to thread 'A' of our hypothetical application discussed in Chapter 3, which reads analog data. The second thread, with *TaskID=1*, manipulates the data by summing the variables and storing the result in another global variable. This is analogous to thread 'B' that processes the analog data to generate control signals. The third thread, with *TaskID=2*, sends the data to the debug serial port, which is analogous to thread 'C' that sends the control signal to output port. Finally, one more thread, with *TaskID=3*, calculates and prints the number of clock cycles consumed to process one data sample. Since each thread is sending out messages to the debug serial port, the output log received on the debug terminal, as shown in Figure 6.6, confirms the successful operation of the suggested approach. The source code for test application 1 is in APPENDIX C-1.

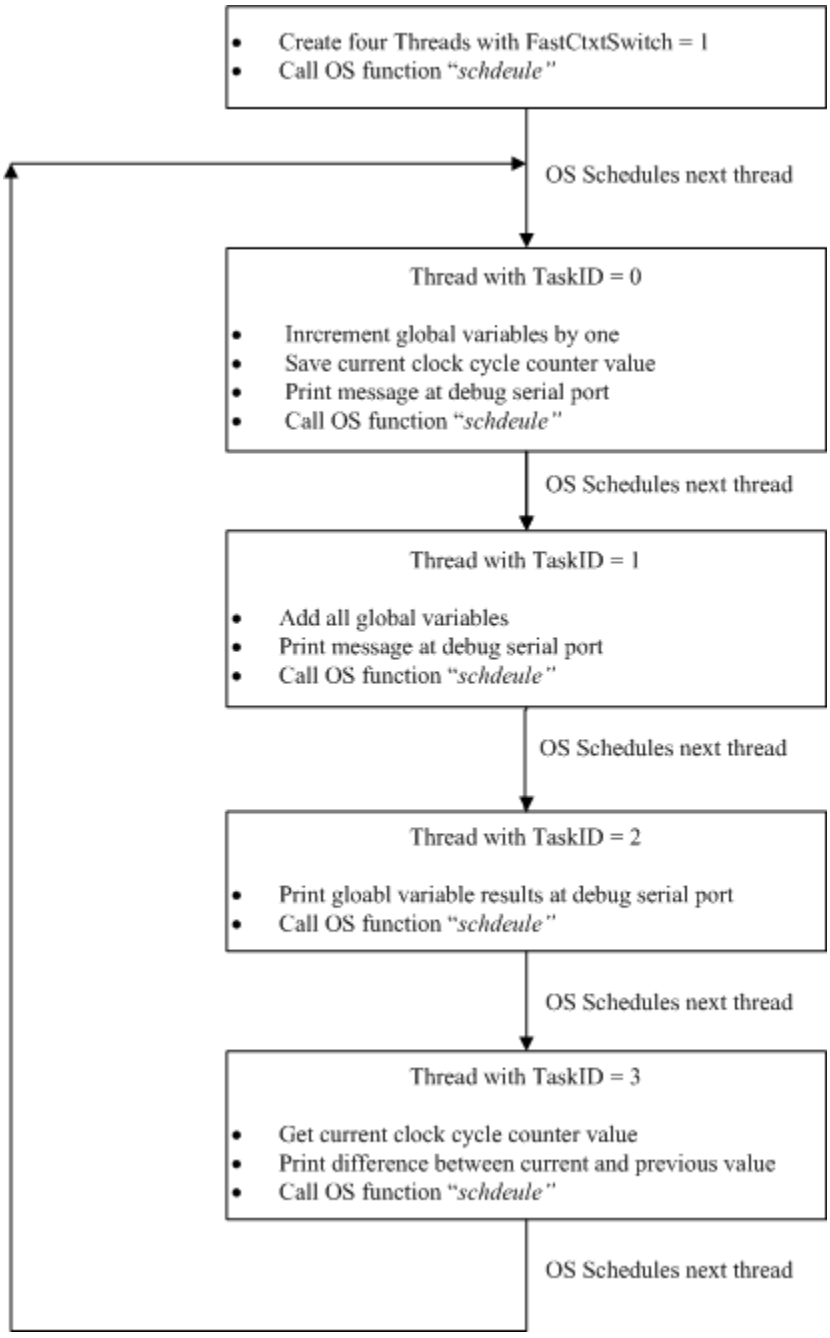

Figure 6. 5: Flowchart for Test Application – 1

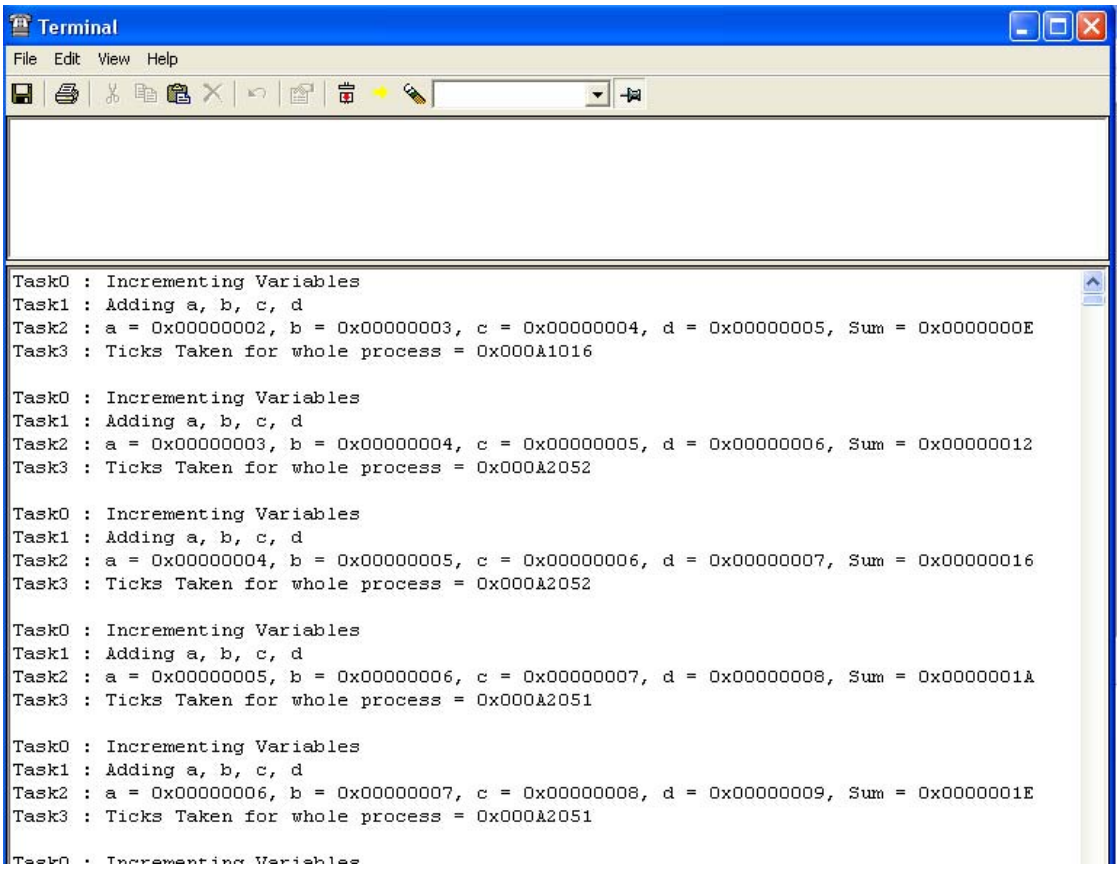

Figure 6. 6: Serial Debug Log from Test Application – 1

6.3.2 Test Application – 2

The second application is designed to measure the performance improvement, in clock cycles. It creates four threads that execute in never-ending loops. The first thread, with *TaskID=0*, stores the current clock cycle counter value in a global variable and does the context switch to the next thread using an internal register file. The second thread, with *TaskID=1*, reads the new current clock cycle counter value, calculates the clock cycles consumed by these two threads, saves the result in another global variable, and releases the control to next thread. The threads with *TaskID=2* and *TaskID=3* repeat the

process with the same code as the first two threads, but context-switch using external memory. As the threads *TaskID=0* and *TaskID=1* does the context-switching using internal register files and *TaskID=2* and *TaskID=3* does context-switching using external memory, the difference in the clock cycles consumed by these two sets of threads determines the performance improvement per context-switch in clock cycles. Thread with *TaskID*=3 additionally does this calculation and sends the results on the debug serial port in hexadecimal format.

Messages sent to the debug port include: number of clock cycles consumed by threads with *TaskID=0* and *TaskID=1*; number of clock cycles consumed by threads with *TaskID=2* and *TaskID=3*; and, finally, the difference between these two. Since the threads are executing in never-ending loops, the application will keep on sending this information to the debug serial port. Figure 6.7 shows the output log for this test Application – 2. The output shows that the suggested approach saves **0x46 (70)** clock cycles per context switch when using the suggested Plasma MIPS processor architecture as compared to a regular MIPS processor. The source code for this test application is in APPENDIX C-2.

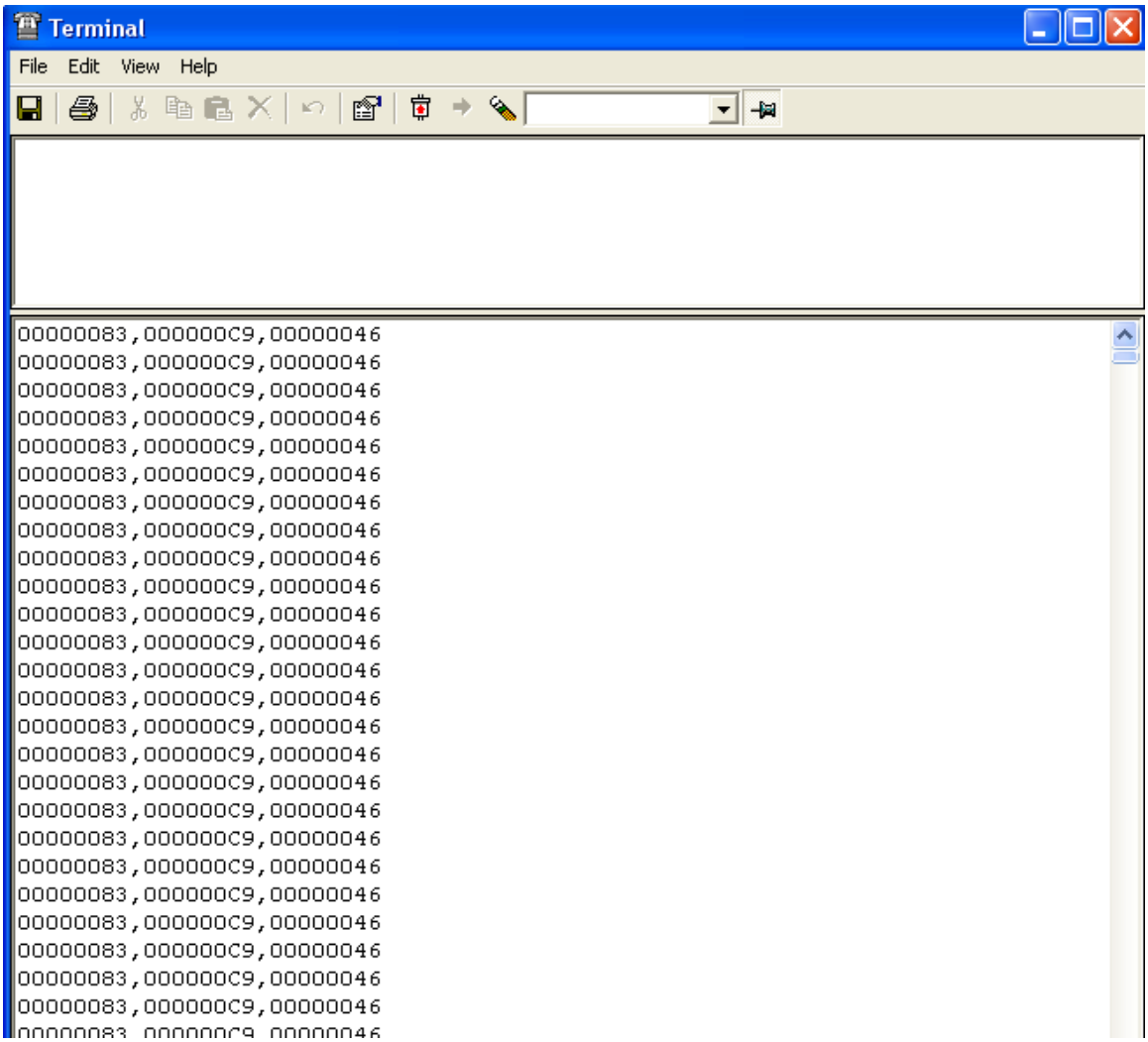

Figure 6. 7: Serial Debug Log from Test Application – 2

6.3.3 Test Application – 3

This test application has been developed to calculate the percentage of performance improvement for our hypothetical application that continuously performs frequent context switching. This application measures the number of data samples processed in a fixed number of clock cycles (0x70000) under both context-switching conditions. It has two parts. The first part (test Application – 3A) executes test

Application– 1 for 0x70000 clock cycles without printing any message/results on the debug serial port. After executing the application for 0x70000 clock cycles, the thread with *TaskID=3* prints the number of data samples processed during this period. In the second part (test Application – 3B), the process is repeated but with context switching using external memory. The difference in the results of these two executions can be used to calculate overall system performance percentage improvement in terms of number of data samples processed or percentage improvement in data throughput.

As shown in Figure 6.8, test Application-3A processes 0x327D (12925) data samples in the allocated 0x70000 clock cycles, and Figure 6.9 shows test Application-3B processed 0x21D2 (8658) data samples in the same amount of time. Therefore, the suggested approach processes 12925 – 8658 = 4267 additional samples in the same amount of time, which gives 4267 / 8658 * 100 = **49.28%** performance improvement for this test application. Since the test application is not doing any additional work, this can be interpreted as the maximum performance improvement possible for any application running on the suggested MIPS processor architecture. As different applications would have more functionality with less context switching, this performance percentage improvement would be reduced for those types of applications.

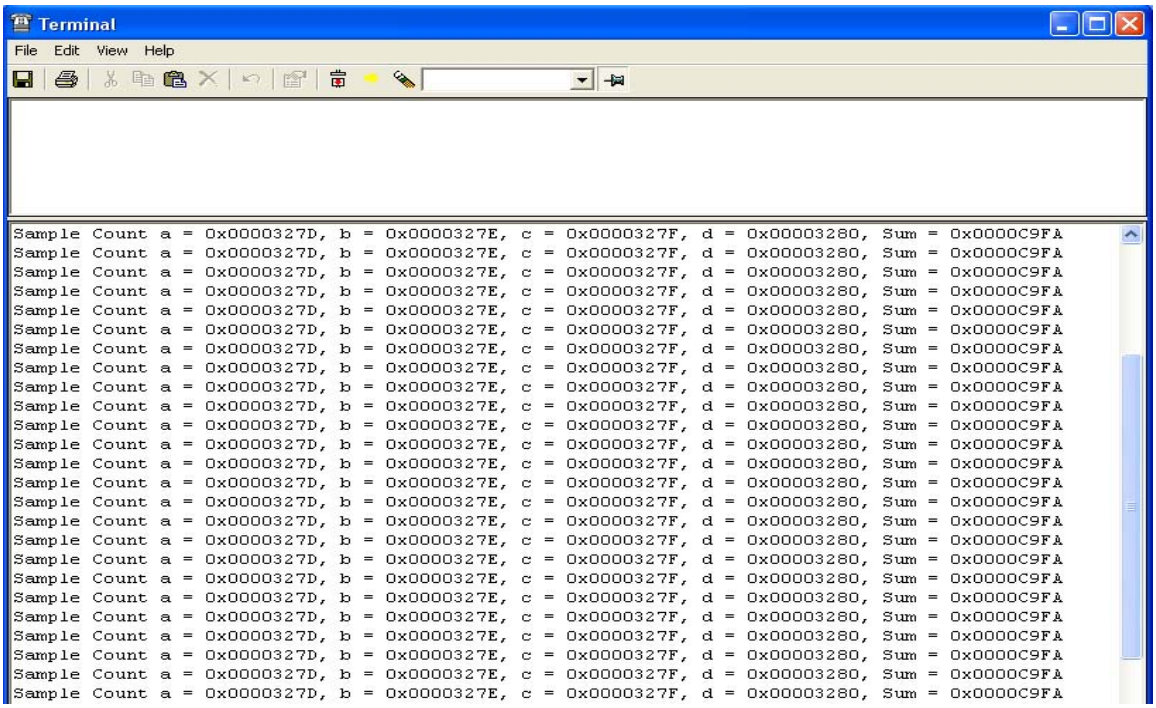

Figure 6. 8: Serial Debug Log from Test Application – 3A

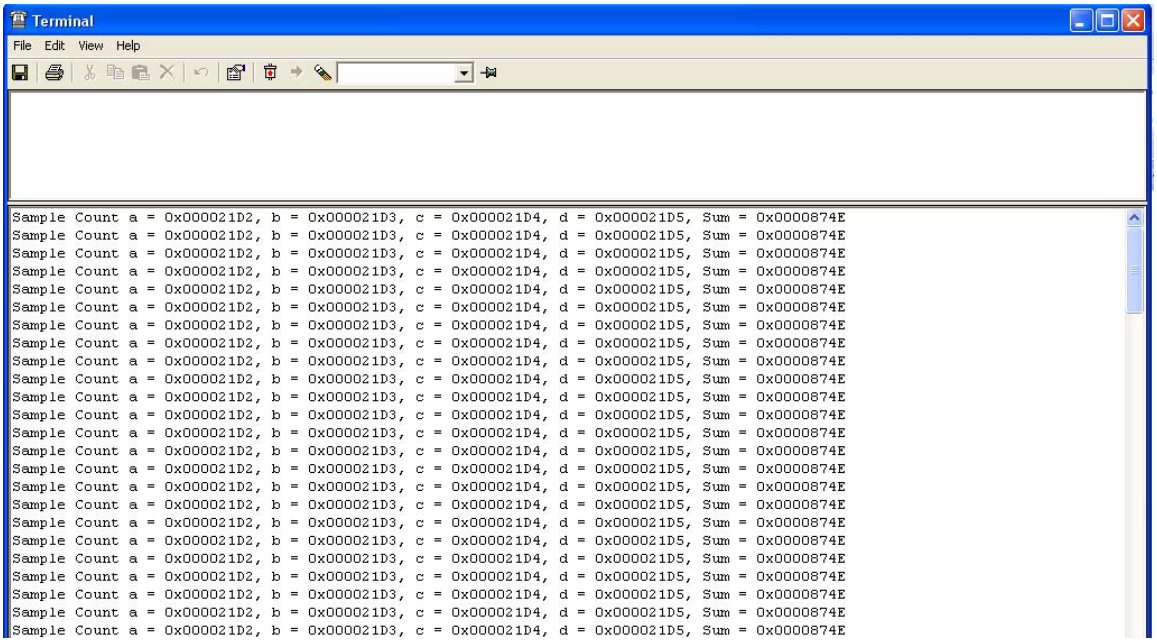

Figure 6. 9: Serial Debug Log from Test Application – 3B

CHAPTER 7 – CONCLUSIONS AND FUTURE WORK

This thesis proposed a hardware solution to reduce context-switching overhead in a RTOS. To reduce the context-switching time, context-switch register files were implemented within the processor architecture. The size of the each register file was equal to the number of CPU context registers. Two special context-switch CPU instructions, to handle saving and restoring the context, were implemented in the hardware. Each of these instructions consumed two clock cycles to move the CPU context registers to or from a context-register file.

The GNU assembler was also modified to support these newly implemented context- switch instructions. A basic co-operative operating system and three test applications were developed to test and measure the performance of the suggested approach.

The proposed approach allowed the RTOS to achieve the context switching in just 4 clock cycles, independent of the number of context registers. This improved the ability of hard RTOSs to meet their basic requirements. Based upon the observations and experimental results, we can draw the following conclusions:

- Hard real-time systems, in which frequent context switching is required, can benefit greatly from this approach.
- The proposed approach improves RTOS-based system performance drastically and makes the system deterministic in meeting the thread’s deadlines.

- The suggested approach increases the efficiency of a RTOS-based system as the system spends less time in managing the threads and therefore uses CPU time more efficiently.

There are multiple improvements possible to the current suggested approach. These improvements can be implemented as per a system requirement or to simplify system functionality. Some of them are listed here.

1. In the current approach, if the '*scxt*' and '*rcxt*' instruction's operand contains an out-of-range register file index, then the instructions behaves as NOP instructions. An exception can be generated to indicate that the context switch has not been completed. In the case of this exception, context-switching can be done using external memory by the exception handler software.
2. A purely software-based solution can also be implemented for an out-of-range register file index. In that case, software needs to check the operand register value before calling context-switch instructions. If the value is out-of-range, then the context would be saved/restored from external memory.
3. For simplicity, the suggested approach can be implemented without adding new instructions. One new register can be implemented in the hardware. The software can write a pre-defined bit pattern to achieve context switch in internal register files.
4. This approach can be implemented for reconfigurable hardware. Register files can be created at run-time, under software control. In this case, the RTOS kernel needs to manage the context-switch hardware.

This approach can also be used for soft RTOS and regular operating systems to improve system throughput. In case of regular operating systems where threads are created at run-time, it is difficult to know the number of threads at system design time. Therefore, it may not be possible to implement register files for all threads in the system. The threads, with frequent context switching, can be set for fast context switching using internal register files by a specially designed scheduler algorithm. This thesis is a step forward in moving the RTOS kernel to hardware.

Another expansion to this research is to attempt to save the CPU context during hardware interrupts as that will reduce the interrupt latency of the system, which is also an important factor for hard real-time systems.

BIBLIOGRAPHY

[1] Francis M. David, Jeffery C. Carlyle, Roy H. Campbell "*Context Switch Overheads for Linux on ARM Platforms*" ExpCS, San Diego, California, Article No.: 3, 14-15 June, 2007

[2] Zhaohui Wu, Hong Li, Zhigang Gao, Jie Sun, Jiang Li "*An Improved Method of Task Context Switching in OSEK Operating System*", Advanced Information Networking and Applications, 2006. AINA 2006. 20th International Conference, pp. 6, Publication date: 18-20 April 2006

[3] Jeffrey S. Snyder, David B. Whalley, Theodore P. Baker "*Fast Context Switches: Compiler and Architectural support for Preemptive Scheduling*" Microprocessors and Microsystems, pp. 35-42, 1995. Available:citesser.ist.psu.edu/33707.html

[4] Xiangrong Zhou, Peter Petrov "*Rapid and low-cost context-switch through embedded processor customization for real-time and control applications* " DAC San Francisco, CA, Pages: 352 - 357 24-28, July, 2006.

[5] Hyden Kwok-Hay So, Robert W. Broderson"BORPH: *An Operating System for FPGA-Based Reconfigurable Computers"* DAC University of California, Berkeley, Technical Report No. UCB/EECS-2007-92, 20 July, 2007

[6] Gilles Chanteperdrix, Richard Cochran "*The ARM Fast Context Switch Extension for Linux*" Papers from the Real Time Linux Workshop, October 14, 2009

[7] MIPS Assembly Language Programmer's Guide, ASM – 01-DOC, PartNumber 02-0036-005 October, 1992

[8] Express Logic Inc. "*Using Event Trace to Analyze Real-Time System Behavior and Increase Throughput*", http://www.rtos.com/PDFs/AnalyzingReal-TimeSystemBehavior.pdf

[9] Xilinx Corp, "Spartan 3E Starter Kit board user Guide" March 9, 2006

[10] Plasma MIPS Processor Design, http://www.opencores.org/project,plasma

[11] GNU compiler and assembler for MIPS, http://ftp.gnu.org/gnu/binutils/

[12] Abraham Silberschatz, Peter Baer Galvin "*Operating System Concepts"* – Fifth Edition: WILEY, Singapore, 1997

[13] David A. Patterson, John L. Hennessy "*Computer Organization and Design"* Third Edition: Morgan Kaufmann Publications San Francisco, 2005

[14] Pater J. Ashenden "*The Designer's Guide to VHDL"* Third Edition, Morgan Kaufmann Publications San Francisco, 2008

[15] http://www.cs.unibo.it/~solmi/teaching/arch_2002-2003/AssemblyLanguageProgDoc.pdf

APPENDIX A-1

```
--------------------------------------------------------------------
--    File    : reg_bank.vhd
--
--    This file implements a register bank with 32 registers that are
--    32-bits wide.
--    These register are implemented as FPGA logic. This file also
--    implements 4 context switch register files which are used to
--    save the operating systems's thread'd context.
--------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use work.mlite_pack.all;

entity reg_bank is
    port(clk            : in  std_logic;
        reset_in       : in  std_logic;
        pause          : in  std_logic;
        rs_index       : in  std_logic_vector(5 downto 0);
        rt_index       : in  std_logic_vector(5 downto 0);
        rd_index       : in  std_logic_vector(5 downto 0);
        reg_source_out : out std_logic_vector(31 downto 0);
        reg_target_out : out std_logic_vector(31 downto 0);
        reg_dest_new   : in  std_logic_vector(31 downto 0);
        intr_enable    : out std_logic;
            cnxt_switch    : in cnxt_switch_func_type);
end; --entity reg_bank

architecture logic of reg_bank is
   signal reg31, reg01, reg02, reg03 : std_logic_vector(31 downto 0);

   signal reg04, reg05, reg06, reg07 : std_logic_vector(31 downto 0);
   signal reg08, reg09, reg10, reg11 : std_logic_vector(31 downto 0);
   signal reg12, reg13, reg14, reg15 : std_logic_vector(31 downto 0);
   signal reg16, reg17, reg18, reg19 : std_logic_vector(31 downto 0);
   signal reg20, reg21, reg22, reg23 : std_logic_vector(31 downto 0);
   signal reg24, reg25, reg26, reg27 : std_logic_vector(31 downto 0);
   signal reg28, reg29, reg30        : std_logic_vector(31 downto 0);
   signal reg_epc                    : std_logic_vector(31 downto 0);
   signal reg_status                 : std_logic;

--    Context switch register files

   signal ctxt_reg01, ctxt_reg02, ctxt_reg03, ctxt_reg04 :
std_logic_vector(31 downto 0);
```

```
   signal ctxt_reg05, ctxt_reg06, ctxt_reg07, ctxt_reg08 :
std_logic_vector(31 downto 0);
   signal ctxt_reg09, ctxt_reg010, ctxt_reg011, ctxt_reg012 :
std_logic_vector(31 downto 0);

   signal ctxt_reg11, ctxt_reg12, ctxt_reg13, ctxt_reg14 :
std_logic_vector(31 downto 0);
   signal ctxt_reg15, ctxt_reg16, ctxt_reg17, ctxt_reg18 :
std_logic_vector(31 downto 0);
   signal ctxt_reg19, ctxt_reg110, ctxt_reg111, ctxt_reg112 :
std_logic_vector(31 downto 0);

   signal ctxt_reg21, ctxt_reg22, ctxt_reg23, ctxt_reg24 :
std_logic_vector(31 downto 0);
   signal ctxt_reg25, ctxt_reg26, ctxt_reg27, ctxt_reg28 :
std_logic_vector(31 downto 0);
   signal ctxt_reg29, ctxt_reg210, ctxt_reg211, ctxt_reg212 :
std_logic_vector(31 downto 0);

   signal ctxt_reg31, ctxt_reg32, ctxt_reg33, ctxt_reg34 :
std_logic_vector(31 downto 0);
   signal ctxt_reg35, ctxt_reg36, ctxt_reg37, ctxt_reg38 :
std_logic_vector(31 downto 0);
   signal ctxt_reg39, ctxt_reg310, ctxt_reg311, ctxt_reg312 :
std_logic_vector(31 downto 0);

begin

reg_proc: process(clk, rs_index, rt_index, rd_index, reg_dest_new,
   reg31, reg01, reg02, reg03, reg04, reg05, reg06, reg07,
   reg08, reg09, reg10, reg11, reg12, reg13, reg14, reg15,
   reg16, reg17, reg18, reg19, reg20, reg21, reg22, reg23,
   reg24, reg25, reg26, reg27, reg28, reg29, reg30,
   reg_epc, reg_status, reset_in, cnxt_switch)

   variable RegFileIndex  : std_logic_vector(31 downto 0);

begin

   if clk'event and clk = '1' then
      case rd_index is
      when "000001" => reg01 <= reg_dest_new;
      when "000010" => reg02 <= reg_dest_new;
      when "000011" => reg03 <= reg_dest_new;
      when "000100" => reg04 <= reg_dest_new;
      when "000101" => reg05 <= reg_dest_new;
      when "000110" => reg06 <= reg_dest_new;
      when "000111" => reg07 <= reg_dest_new;
      when "001000" => reg08 <= reg_dest_new;
      when "001001" => reg09 <= reg_dest_new;
      when "001010" => reg10 <= reg_dest_new;
      when "001011" => reg11 <= reg_dest_new;
      when "001100" => reg12 <= reg_dest_new;
```

```
when "001101" => reg13 <= reg_dest_new;
when "001110" => reg14 <= reg_dest_new;
when "001111" => reg15 <= reg_dest_new;
when "010000" => reg16 <= reg_dest_new;
when "010001" => reg17 <= reg_dest_new;
when "010010" => reg18 <= reg_dest_new;
when "010011" => reg19 <= reg_dest_new;
when "010100" => reg20 <= reg_dest_new;
when "010101" => reg21 <= reg_dest_new;
when "010110" => reg22 <= reg_dest_new;
when "010111" => reg23 <= reg_dest_new;
when "011000" => reg24 <= reg_dest_new;
when "011001" => reg25 <= reg_dest_new;
when "011010" => reg26 <= reg_dest_new;
when "011011" => reg27 <= reg_dest_new;
when "011100" => reg28 <= reg_dest_new;
when "011101" => reg29 <= reg_dest_new;
when "011110" => reg30 <= reg_dest_new;
when "011111" => reg31 <= reg_dest_new;
when "101100" => reg_status <= reg_dest_new(0);
when "101110" => reg_epc <= reg_dest_new;  --CP0 14
                 reg_status <= '0';        --disable interrupts
when others =>
end case;

      -- Initialise all the registers
 if reset_in = '1' then
       reg_status <= '0';
 reg_epc <= x"00000000";
 RegFileIndex := x"00000000";
 reg01 <= x"00000000";
 reg02 <= x"00000000";
 reg03 <= x"00000000";
 reg04 <= x"00000000";
 reg05 <= x"00000000";
 reg06 <= x"00000000";
 reg07 <= x"00000000";
 reg08 <= x"00000000";
 reg09 <= x"00000000";
 reg10 <= x"00000000";
 reg11 <= x"00000000";
 reg12 <= x"00000000";
 reg13 <= x"00000000";
 reg14 <= x"00000000";
 reg15 <= x"00000000";
 reg16 <= x"00000000";
 reg17 <= x"00000000";
 reg18 <= x"00000000";
 reg19 <= x"00000000";
 reg20 <= x"00000000";
 reg21 <= x"00000000";
 reg22 <= x"00000000";
 reg23 <= x"00000000";
```

```
reg24 <= x"00000000";
reg25 <= x"00000000";
reg26 <= x"00000000";
reg27 <= x"00000000";
reg28 <= x"00000000";
reg29 <= x"00000000";
reg30 <= x"00000000";
reg31 <= x"00000000";

      -- Initialize context switch Register files
   ctxt_reg01  <= x"00000000";
ctxt_reg02  <= x"00000000";
ctxt_reg03  <= x"00000000";
ctxt_reg04  <= x"00000000";
ctxt_reg05  <= x"00000000";
ctxt_reg06  <= x"00000000";
ctxt_reg07  <= x"00000000";
ctxt_reg08  <= x"00000000";
ctxt_reg09  <= x"00000000";
ctxt_reg010 <= x"00000000";
ctxt_reg011 <= x"00000000";
ctxt_reg012 <= x"00000000";

ctxt_reg11  <= x"00000000";
ctxt_reg12  <= x"00000000";
ctxt_reg13  <= x"00000000";
ctxt_reg14  <= x"00000000";
ctxt_reg15  <= x"00000000";
ctxt_reg16  <= x"00000000";
ctxt_reg17  <= x"00000000";
ctxt_reg18  <= x"00000000";
ctxt_reg19  <= x"00000000";
ctxt_reg110 <= x"00000000";
ctxt_reg111 <= x"00000000";
ctxt_reg112 <= x"00000000";

ctxt_reg21  <= x"00000000";
ctxt_reg22  <= x"00000000";
ctxt_reg23  <= x"00000000";
ctxt_reg24  <= x"00000000";
ctxt_reg25  <= x"00000000";
ctxt_reg26  <= x"00000000";
ctxt_reg27  <= x"00000000";
ctxt_reg28  <= x"00000000";
ctxt_reg29  <= x"00000000";
ctxt_reg210 <= x"00000000";
ctxt_reg211 <= x"00000000";
ctxt_reg212 <= x"00000000";

ctxt_reg31  <= x"00000000";
ctxt_reg32  <= x"00000000";
ctxt_reg33  <= x"00000000";
ctxt_reg34  <= x"00000000";
```

```
      ctxt_reg35  <= x"00000000";
      ctxt_reg36  <= x"00000000";
      ctxt_reg37  <= x"00000000";
      ctxt_reg38  <= x"00000000";
      ctxt_reg39  <= x"00000000";
      ctxt_reg310 <= x"00000000";
      ctxt_reg311 <= x"00000000";
      ctxt_reg312 <= x"00000000";
    end if;

case cnxt_switch is
    when SAVE_CNXT =>
      --case rt_index is
      case RegFileIndex(2 downto 0) is
             when "000" =>
            ctxt_reg01  <=  reg16;
            ctxt_reg02  <=  reg17;
            ctxt_reg03  <=  reg18;
            ctxt_reg04  <=  reg19;
            ctxt_reg05  <=  reg20;
            ctxt_reg06  <=  reg21;
            ctxt_reg07  <=  reg22;
            ctxt_reg08  <=  reg23;
            ctxt_reg09  <=  reg28;
            ctxt_reg010 <=  reg29;
            ctxt_reg011 <=  reg30;
            ctxt_reg012 <=  reg31;

        when "001" =>
            ctxt_reg11  <= reg16;
            ctxt_reg12  <= reg17;
            ctxt_reg13  <= reg18;
            ctxt_reg14  <= reg19;
            ctxt_reg15  <= reg20;
            ctxt_reg16  <= reg21;
            ctxt_reg17  <= reg22;
            ctxt_reg18  <= reg23;
            ctxt_reg19  <= reg28;
            ctxt_reg110 <= reg29;
            ctxt_reg111 <= reg30;
            ctxt_reg112 <= reg31;

        when "010" =>
            ctxt_reg21  <= reg16;
            ctxt_reg22  <= reg17;
            ctxt_reg23  <= reg18;
            ctxt_reg24  <= reg19;
            ctxt_reg25  <= reg20;
            ctxt_reg26  <= reg21;
            ctxt_reg27  <= reg22;
            ctxt_reg28  <= reg23;
            ctxt_reg29  <= reg28;
            ctxt_reg210 <= reg29;
```

```
            ctxt_reg211 <= reg30;
            ctxt_reg212 <= reg31;

        when "011" =>
            ctxt_reg31  <= reg16;
            ctxt_reg32  <= reg17;
            ctxt_reg33  <= reg18;
            ctxt_reg34  <= reg19;
            ctxt_reg35  <= reg20;
            ctxt_reg36  <= reg21;
            ctxt_reg37  <= reg22;
            ctxt_reg38  <= reg23;
            ctxt_reg39  <= reg28;
            ctxt_reg310 <= reg29;
            ctxt_reg311 <= reg30;
            ctxt_reg312 <= reg31;
          when others =>
          end case;

    when RSTR_CNXT =>
          --case rt_index is
          case RegFileIndex(2 downto 0) is
              when "000" =>
             reg16 <= ctxt_reg01;
             reg17 <= ctxt_reg02;
             reg18 <= ctxt_reg03;
             reg19 <= ctxt_reg04;
             reg20 <= ctxt_reg05;
             reg21 <= ctxt_reg06;
             reg22 <= ctxt_reg07;
             reg23 <= ctxt_reg08;
             reg28 <= ctxt_reg09;
             reg29 <= ctxt_reg010;
             reg30 <= ctxt_reg011;
             reg31 <= ctxt_reg012;

               when "001" =>
              reg16 <= ctxt_reg11;
              reg17 <= ctxt_reg12;
              reg18 <= ctxt_reg13;
              reg19 <= ctxt_reg14;
              reg20 <= ctxt_reg15;
              reg21 <= ctxt_reg16;
              reg22 <= ctxt_reg17;
              reg23 <= ctxt_reg18;
              reg28 <= ctxt_reg19;
              reg29 <= ctxt_reg110;
              reg30 <= ctxt_reg111;
              reg31 <= ctxt_reg112;

           when "010" =>
                reg16 <= ctxt_reg21;
                reg17 <= ctxt_reg22;
```

```
                reg18 <= ctxt_reg23;
                reg19 <= ctxt_reg24;
                reg20 <= ctxt_reg25;
                reg21 <= ctxt_reg26;
                reg22 <= ctxt_reg27;
                reg23 <= ctxt_reg28;
                reg28 <= ctxt_reg29;
                reg29 <= ctxt_reg210;
                reg30 <= ctxt_reg211;
                reg31 <= ctxt_reg212;

            when "011" =>
                reg16 <= ctxt_reg31;
                reg17 <= ctxt_reg32;
                reg18 <= ctxt_reg33;
                reg19 <= ctxt_reg34;
                reg20 <= ctxt_reg35;
                reg21 <= ctxt_reg36;
                reg22 <= ctxt_reg37;
                reg23 <= ctxt_reg38;
                reg28 <= ctxt_reg39;
                reg29 <= ctxt_reg310;
                reg30 <= ctxt_reg311;
                reg31 <= ctxt_reg312;
             when others =>
             end case;

       when others =>
       end case;
    end if;

    case rs_index is
    when "000000" => reg_source_out <= ZERO;
    when "000001" => reg_source_out <= reg01;
    when "000010" => reg_source_out <= reg02;
    when "000011" => reg_source_out <= reg03;
    when "000100" => reg_source_out <= reg04;
    when "000101" => reg_source_out <= reg05;
    when "000110" => reg_source_out <= reg06;
    when "000111" => reg_source_out <= reg07;
    when "001000" => reg_source_out <= reg08;
    when "001001" => reg_source_out <= reg09;
    when "001010" => reg_source_out <= reg10;
    when "001011" => reg_source_out <= reg11;
    when "001100" => reg_source_out <= reg12;
    when "001101" => reg_source_out <= reg13;
    when "001110" => reg_source_out <= reg14;
    when "001111" => reg_source_out <= reg15;
    when "010000" => reg_source_out <= reg16;
    when "010001" => reg_source_out <= reg17;
    when "010010" => reg_source_out <= reg18;
    when "010011" => reg_source_out <= reg19;
    when "010100" => reg_source_out <= reg20;
```

```
   when "010101" => reg_source_out <= reg21;
   when "010110" => reg_source_out <= reg22;
   when "010111" => reg_source_out <= reg23;
   when "011000" => reg_source_out <= reg24;
   when "011001" => reg_source_out <= reg25;
   when "011010" => reg_source_out <= reg26;
   when "011011" => reg_source_out <= reg27;
   when "011100" => reg_source_out <= reg28;
   when "011101" => reg_source_out <= reg29;
   when "011110" => reg_source_out <= reg30;
   when "011111" => reg_source_out <= reg31;
   when "101100" => reg_source_out <= ZERO(31 downto 1) & reg_status;
   when "101110" => reg_source_out <= reg_epc;     --CP0 14
   when "111111" => reg_source_out <= '1' & ZERO(30 downto 0); --intr
vector
   when others =>   reg_source_out <= ZERO;
   end case;

   case rt_index is
   when "000000" => RegFileIndex := ZERO;
   when "000001" => RegFileIndex := reg01;
   when "000010" => RegFileIndex := reg02;
   when "000011" => RegFileIndex := reg03;
   when "000100" => RegFileIndex := reg04;
   when "000101" => RegFileIndex := reg05;
   when "000110" => RegFileIndex := reg06;
   when "000111" => RegFileIndex := reg07;
   when "001000" => RegFileIndex := reg08;
   when "001001" => RegFileIndex := reg09;
   when "001010" => RegFileIndex := reg10;
   when "001011" => RegFileIndex := reg11;
   when "001100" => RegFileIndex := reg12;
   when "001101" => RegFileIndex := reg13;
   when "001110" => RegFileIndex := reg14;
   when "001111" => RegFileIndex := reg15;
   when "010000" => RegFileIndex := reg16;
   when "010001" => RegFileIndex := reg17;
   when "010010" => RegFileIndex := reg18;
   when "010011" => RegFileIndex := reg19;
   when "010100" => RegFileIndex := reg20;
   when "010101" => RegFileIndex := reg21;
   when "010110" => RegFileIndex := reg22;
   when "010111" => RegFileIndex := reg23;
   when "011000" => RegFileIndex := reg24;
   when "011001" => RegFileIndex := reg25;
   when "011010" => RegFileIndex := reg26;
   when "011011" => RegFileIndex := reg27;
   when "011100" => RegFileIndex := reg28;
   when "011101" => RegFileIndex := reg29;
   when "011110" => RegFileIndex := reg30;
   when "011111" => RegFileIndex := reg31;
   when others =>   RegFileIndex := ZERO;
   end case;
```

```
   intr_enable <= reg_status;
   reg_target_out <= RegFileIndex;
end process;

end; --architecture logic
```

APPENDIX A-2

```
------------------------------------------------------------------------
-- File : Control.vhd
--
-- DESCRIPTION:
--    Controls the CPU by decoding the opcode and generating control
--    signals to the rest of the CPU.
--    This file has been modifed to for OS context switch
--    instructions implementation in the hardware.
------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use work.mlite_pack.all;

entity control is
   port(opcode       : in  std_logic_vector(31 downto 0);
        intr_signal  : in  std_logic;
        rs_index     : out std_logic_vector(5 downto 0);
        rt_index     : out std_logic_vector(5 downto 0);
        rd_index     : out std_logic_vector(5 downto 0);
        imm_out      : out std_logic_vector(15 downto 0);
        alu_func     : out alu_function_type;
        shift_func   : out shift_function_type;
        mult_func    : out mult_function_type;
        branch_func  : out branch_function_type;
        a_source_out : out a_source_type;
        b_source_out : out b_source_type;
        c_source_out : out c_source_type;
        pc_source_out: out pc_source_type;
        mem_source_out:out mem_source_type;
        exception_out: out std_logic;
        -- added for OS context switch
        cnxt_switch  : out cnxt_switch_func_type);
end; --entity control

architecture logic of control is
begin

control_proc: process(opcode, intr_signal)
   variable op, func       : std_logic_vector(5 downto 0);
   variable rs, rt, rd     : std_logic_vector(5 downto 0);
   variable rtx            : std_logic_vector(4 downto 0);
   variable imm            : std_logic_vector(15 downto 0);
   -- Adding Context switch variable - Deepak
   variable cnxt_sw        : cnxt_switch_func_type;
   -- change ends - Deepak
   variable alu_function   : alu_function_type;
```

```
   variable shift_function : shift_function_type;
   variable mult_function  : mult_function_type;
   variable a_source       : a_source_type;
   variable b_source       : b_source_type;
   variable c_source       : c_source_type;
   variable pc_source      : pc_source_type;
   variable branch_function: branch_function_type;
   variable mem_source     : mem_source_type;
   variable is_syscall     : std_logic;
begin
   alu_function := ALU_NOTHING;
   shift_function := SHIFT_NOTHING;
   mult_function := MULT_NOTHING;
   a_source := A_FROM_REG_SOURCE;
   b_source := B_FROM_REG_TARGET;
   c_source := C_FROM_NULL;
   pc_source := FROM_INC4;
   branch_function := BRANCH_EQ;
   mem_source := MEM_FETCH;
   op := opcode(31 downto 26);
   rs := '0' & opcode(25 downto 21);
   rt := '0' & opcode(20 downto 16);
   rtx := opcode(20 downto 16);
   rd := '0' & opcode(15 downto 11);
   func := opcode(5 downto 0);
   imm := opcode(15 downto 0);
   is_syscall := '0';

   cnxt_sw := NO_CNXT_SW;

   case op is
   when "000000" =>   --SPECIAL
      case func is
      when "000000" =>   --SLL   r[rd]=r[rt]<<re;   0
         a_source := A_FROM_IMM10_6;
         c_source := C_FROM_SHIFT;
         shift_function := SHIFT_LEFT_UNSIGNED;

      when "000010" =>   --SRL   r[rd]=u[rt]>>re;  2
         a_source := A_FROM_IMM10_6;
         c_source := C_FROM_shift;
         shift_function := SHIFT_RIGHT_UNSIGNED;

      when "000011" =>   --SRA   r[rd]=r[rt]>>re;   3
         a_source := A_FROM_IMM10_6;
         c_source := C_FROM_SHIFT;
         shift_function := SHIFT_RIGHT_SIGNED;

      when "000100" =>   --SLLV  r[rd]=r[rt]<<r[rs]; 4
         c_source := C_FROM_SHIFT;
         shift_function := SHIFT_LEFT_UNSIGNED;

      when "000110" =>   --SRLV  r[rd]=u[rt]>>r[rs];  6
```

```
   c_source := C_FROM_SHIFT;
   shift_function := SHIFT_RIGHT_UNSIGNED;

when "000111" =>   --SRAV  r[rd]=r[rt]>>r[rs];  7
   c_source := C_FROM_SHIFT;
   shift_function := SHIFT_RIGHT_SIGNED;

when "001000" =>   --JR    s->pc_next=r[rs];      8
   pc_source := FROM_BRANCH;
   alu_function := ALU_ADD;
   branch_function := BRANCH_YES;

when "001001" =>   --JALR  r[rd]=s->pc_next; s->pc_next=r[rs];
   c_source := C_FROM_PC_PLUS4;
   pc_source := FROM_BRANCH;
   alu_function := ALU_ADD;
   branch_function := BRANCH_YES;

when "001100" =>   --SYSCALL   12
   is_syscall := '1';

when "001101" =>   --BREAK s->wakeup=1;   13
   is_syscall := '1';

--when "001111" =>   --SYNC  s->wakeup=1;  15

when "010000" =>   --MFHI  r[rd]=s->hi;    16
   c_source := C_FROM_MULT;
   mult_function := MULT_READ_HI;

when "010001" =>   --FTHI  s->hi=r[rs];      17
   mult_function := MULT_WRITE_HI;

when "010010" =>   --MFLO  r[rd]=s->lo;      18
   c_source := C_FROM_MULT;
   mult_function := MULT_READ_LO;

when "010011" =>   --MTLO  s->lo=r[rs];      19
   mult_function := MULT_WRITE_LO;

when "011000" =>   --MULT  s->lo=r[rs]*r[rt]; s->hi=0;    24
   mult_function := MULT_SIGNED_MULT;

when "011001" =>   --MULTU s->lo=r[rs]*r[rt]; s->hi=0;     25
   mult_function := MULT_MULT;

when "011010" =>   --DIV  s->lo=r[rs]/r[rt]; s->hi=r[rs]%r[rt];
   mult_function := MULT_SIGNED_DIVIDE;

when "011011" =>   --DIVU  s->lo=r[rs]/r[rt]; s->hi=r[rs]%r[rt];
   mult_function := MULT_DIVIDE;

when "100000" =>   --ADD   r[rd]=r[rs]+r[rt];    32
```

```
            c_source := C_FROM_ALU;
            alu_function := ALU_ADD;

         when "100001" =>   --ADDU  r[rd]=r[rs]+r[rt];    33
            c_source := C_FROM_ALU;
            alu_function := ALU_ADD;

         when "100010" =>   --SUB   r[rd]=r[rs]-r[rt];    34
            c_source := C_FROM_ALU;
            alu_function := ALU_SUBTRACT;

         when "100011" =>   --SUBU  r[rd]=r[rs]-r[rt];   35
            c_source := C_FROM_ALU;
            alu_function := ALU_SUBTRACT;

         when "100100" =>   --AND   r[rd]=r[rs]&r[rt];  36
            c_source := C_FROM_ALU;
            alu_function := ALU_AND;

         when "100101" =>   --OR    r[rd]=r[rs]|r[rt];   37
            c_source := C_FROM_ALU;
            alu_function := ALU_OR;

         when "100110" =>   --XOR   r[rd]=r[rs]^r[rt];    38
            c_source := C_FROM_ALU;
            alu_function := ALU_XOR;

         when "100111" =>   --NOR   r[rd]=~(r[rs]|r[rt]);  39
            c_source := C_FROM_ALU;
            alu_function := ALU_NOR;

         when "101010" =>   --SLT   r[rd]=r[rs]<r[rt];  42
            c_source := C_FROM_ALU;
            alu_function := ALU_LESS_THAN_SIGNED;

         when "101011" =>   --SLTU  r[rd]=u[rs]<u[rt];   43
            c_source := C_FROM_ALU;
            alu_function := ALU_LESS_THAN;

         when "101101" =>   --DADDU r[rd]=r[rs]+u[rt];   45
            c_source := C_FROM_ALU;
            alu_function := ALU_ADD;

         --when "110001" =>   --TGEU      49
         --when "110010" =>   --TLT        50
         --when "110011" =>   --TLTU      51
         --when "110100" =>   --TEQ       52
         --when "110110" =>   --TNE       54

         -- Adding instructions for Context save and Restore -- Deepak
         when "111100" => -- Save context to rt register file     60
--          rt := opcode(20 downto 16);
            cnxt_sw := SAVE_CNXT;
```

```
     when "111101" =>  --  Restore context from rs register file   61
--          rt := opcode(20 downto 16);
          cnxt_sw := RSTR_CNXT;

    -- Instruction Addition Ends --  Deepak
     when others =>
     end case;

  when "000001" =>   --REGIMM
     rt := "000000";
     rd := "011111";
     a_source := A_FROM_PC;
     b_source := B_FROM_IMMX4;
     alu_function := ALU_ADD;
     pc_source := FROM_BRANCH;
     branch_function := BRANCH_GTZ;
     --if(test) pc=pc+imm*4

     case rtx is
     when "10000" =>   --BLTZAL  r[31]=s->pc_next; branch=r[rs]<0;
        c_source := C_FROM_PC_PLUS4;
        branch_function := BRANCH_LTZ;

     when "00000" =>   --BLTZ    branch=r[rs]<0;
        branch_function := BRANCH_LTZ;

     when "10001" =>   --BGEZAL  r[31]=s->pc_next; branch=r[rs]>=0;
        c_source := C_FROM_PC_PLUS4;
        branch_function := BRANCH_GEZ;

     when "00001" =>   --BGEZ    branch=r[rs]>=0;
        branch_function := BRANCH_GEZ;

       when others =>
       end case;

  when "000011" =>
     c_source := C_FROM_PC_PLUS4;
     rd := "011111";
     pc_source := FROM_OPCODE25_0;

  when "000010" =>   --J      s->pc_next=(s->pc&0xf0000000)|target;
     pc_source := FROM_OPCODE25_0;

  when "000100" =>   --BEQ    branch=r[rs]==r[rt];
     a_source := A_FROM_PC;
     b_source := B_FROM_IMMX4;
     alu_function := ALU_ADD;
     pc_source := FROM_BRANCH;
     branch_function := BRANCH_EQ;

  when "000101" =>   --BNE    branch=r[rs]!=r[rt];
```

```
      a_source := A_FROM_PC;
      b_source := B_FROM_IMMX4;
      alu_function := ALU_ADD;
      pc_source := FROM_BRANCH;
      branch_function := BRANCH_NE;

   when "000110" =>   --BLEZ   branch=r[rs]<=0;
      a_source := A_FROM_PC;
      b_source := b_FROM_IMMX4;
      alu_function := ALU_ADD;
      pc_source := FROM_BRANCH;
      branch_function := BRANCH_LEZ;

   when "000111" =>   --BGTZ   branch=r[rs]>0;
      a_source := A_FROM_PC;
      b_source := B_FROM_IMMX4;
      alu_function := ALU_ADD;
      pc_source := FROM_BRANCH;
      branch_function := BRANCH_GTZ;

   when "001000" =>   --ADDI   r[rt]=r[rs]+(short)imm;
      b_source := B_FROM_SIGNED_IMM;
      c_source := C_FROM_ALU;
      rd := rt;
      alu_function := ALU_ADD;

   when "001001" =>   --ADDIU  u[rt]=u[rs]+(short)imm;
      b_source := B_FROM_SIGNED_IMM;
      c_source := C_FROM_ALU;
      rd := rt;
      alu_function := ALU_ADD;

   when "001010" =>   --SLTI   r[rt]=r[rs]<(short)imm;
      b_source := B_FROM_SIGNED_IMM;
      c_source := C_FROM_ALU;
      rd := rt;
      alu_function := ALU_LESS_THAN_SIGNED;

   when "001011" =>   --SLTIU  u[rt]=u[rs]<(unsigned long)(short)imm;
      b_source := B_FROM_IMM;
      c_source := C_FROM_ALU;
      rd := rt;
      alu_function := ALU_LESS_THAN;

   when "001100" =>   --ANDI   r[rt]=r[rs]&imm;
      b_source := B_FROM_IMM;
      c_source := C_FROM_ALU;
      rd := rt;
      alu_function := ALU_AND;

   when "001101" =>   --ORI    r[rt]=r[rs]|imm;
      b_source := B_FROM_IMM;
      c_source := C_FROM_ALU;
```

```
      rd := rt;
      alu_function := ALU_OR;

   when "001110" =>   --XORI   r[rt]=r[rs]^imm;
      b_source := B_FROM_IMM;
      c_source := C_FROM_ALU;
      rd := rt;
      alu_function := ALU_XOR;

   when "001111" =>   --LUI    r[rt]=(imm<<16);
      c_source := C_FROM_IMM_SHIFT16;
      rd := rt;

   when "010000" =>   --COP0
      alu_function := ALU_OR;
      c_source := C_FROM_ALU;
      if opcode(23) = '0' then  --move from CP0
         rs := '1' & opcode(15 downto 11);
         rt := "000000";
         rd := '0' & opcode(20 downto 16);
      else                       --move to CP0
         rs := "000000";
         rd(5) := '1';
         pc_source := FROM_BRANCH;   --delay possible interrupt
         branch_function := BRANCH_NO;
      end if;

   when "100000" =>   --LB     r[rt]=*(signed char*)ptr;
      a_source := A_FROM_REG_SOURCE;
      b_source := B_FROM_SIGNED_IMM;
      alu_function := ALU_ADD;
      rd := rt;
      c_source := C_FROM_MEMORY;
      mem_source := MEM_READ8S;    --address=(short)imm+r[rs];

   when "100001" =>   --LH     r[rt]=*(signed short*)ptr;
      a_source := A_FROM_REG_SOURCE;
      b_source := B_FROM_SIGNED_IMM;
      alu_function := ALU_ADD;
      rd := rt;
      c_source := C_FROM_MEMORY;
      mem_source := MEM_READ16S;   --address=(short)imm+r[rs];

   when "100010" =>   --LWL    //Not Implemented
      a_source := A_FROM_REG_SOURCE;
      b_source := B_FROM_SIGNED_IMM;
      alu_function := ALU_ADD;
      rd := rt;
      c_source := C_FROM_MEMORY;
      mem_source := MEM_READ32;

   when "100011" =>   --LW     r[rt]=*(long*)ptr;
      a_source := A_FROM_REG_SOURCE;
```

```
   b_source := B_FROM_SIGNED_IMM;
   alu_function := ALU_ADD;
   rd := rt;
   c_source := C_FROM_MEMORY;
   mem_source := MEM_READ32;

when "100100" =>   --LBU    r[rt]=*(unsigned char*)ptr;
   a_source := A_FROM_REG_SOURCE;
   b_source := B_FROM_SIGNED_IMM;
   alu_function := ALU_ADD;
   rd := rt;
   c_source := C_FROM_MEMORY;
   mem_source := MEM_READ8;    --address=(short)imm+r[rs];

when "100101" =>   --LHU    r[rt]=*(unsigned short*)ptr;
   a_source := A_FROM_REG_SOURCE;
   b_source := B_FROM_SIGNED_IMM;
   alu_function := ALU_ADD;
   rd := rt;
   c_source := C_FROM_MEMORY;
   mem_source := MEM_READ16;    --address=(short)imm+r[rs];

--when "100110" =>   --LWR    //Not Implemented

when "101000" =>   --SB     *(char*)ptr=(char)r[rt];
   a_source := A_FROM_REG_SOURCE;
   b_source := B_FROM_SIGNED_IMM;
   alu_function := ALU_ADD;
   mem_source := MEM_WRITE8;   --address=(short)imm+r[rs];

when "101001" =>   --SH     *(short*)ptr=(short)r[rt];
   a_source := A_FROM_REG_SOURCE;
   b_source := B_FROM_SIGNED_IMM;
   alu_function := ALU_ADD;
   mem_source := MEM_WRITE16;

when "101010" =>   --SWL    //Not Implemented
   a_source := A_FROM_REG_SOURCE;
   b_source := B_FROM_SIGNED_IMM;
   alu_function := ALU_ADD;
   mem_source := MEM_WRITE32;  --address=(short)imm+r[rs];

when "101011" =>   --SW     *(long*)ptr=r[rt];
   a_source := A_FROM_REG_SOURCE;
   b_source := B_FROM_SIGNED_IMM;
   alu_function := ALU_ADD;
   mem_source := MEM_WRITE32;  --address=(short)imm+r[rs];

when others =>
end case;

if c_source = C_FROM_NULL then
   rd := "000000";
```

```
    end if;

    if intr_signal = '1' or is_syscall = '1' then
       rs := "111111";  --interrupt vector
       rt := "000000";
       rd := "101110";  --save PC in EPC
       alu_function := ALU_OR;
       shift_function := SHIFT_NOTHING;
       mult_function := MULT_NOTHING;
       branch_function := BRANCH_YES;
       a_source := A_FROM_REG_SOURCE;
       b_source := B_FROM_REG_TARGET;
       c_source := C_FROM_PC;
       pc_source := FROM_LBRANCH;
       mem_source := MEM_FETCH;
       exception_out <= '1';
    else
       exception_out <= '0';
    end if;

    rs_index <= rs;
    rt_index <= rt;
    rd_index <= rd;
    imm_out <= imm;
    alu_func <= alu_function;
    shift_func <= shift_function;
    mult_func <= mult_function;
    branch_func <= branch_function;
    a_source_out <= a_source;
    b_source_out <= b_source;
    c_source_out <= c_source;
    pc_source_out <= pc_source;
    mem_source_out <= mem_source;
    -- context switch signal
    cnxt_switch <= cnxt_sw;

end process;

end; --logic
```

APPENDIX B-1

```
////////////////////////////////////////////////////////////
// File Name     : co_op_rtos.c
//
// Author        : Deepak Gauba
//
// Date          : 19th December, 2009
//
// Desription   :   This file implements a Basic co-operative
//                  Operating System which creates
//                  and switch tasks in roung robin fashion.
////////////////////////////////////////////////////////////

#include "plasma.h"

#define CONTXT_SIZE 15

typedef void (*TaskFunc)(void);

extern int setjmp(int *env);    // save the context on env (array)
extern void longjmp(int *env);  // restore the context from env (array)
extern int fast_setjmp(int val);   // Save context in internal register
files
extern void fast_longjmp(int val); // Restore context from internal
register files

#define MAX_THREADS 4  // Number of threads that this operating system
supports

int Context[MAX_THREADS * CONTXT_SIZE];

// Task structure
typedef struct Task
 {
     void (*TaskPtr)();          // Pointer to Thread Starting Function
     int *State;                 // context
     unsigned char Executed;     // 1 - thread has started, 0 otherwise
     unsigned char TaskID;       // Task ID
     unsigned char FastCtxtSwitch;  // 1 - Require fast context switch,
                                    // 0 otherwise
 }Task;

 Task Threads[MAX_THREADS];

int TaskNext = 0; // start TaskID 0
```

```
////////////////////////////////////////////////////////
// Function     :   createTask()
//
// Parameters   :   int num - Task identification number
//                  void * - Pointer to function
//                  unsigned char - 0 Fast context switch
//                                  1 otherwise
//
// Return       :   void
//
// Desription   :   This function craetes and initialize
//                  the task structure object. The task
//                  structure member "Executed" is
//                  initialized with 0 indiacted that this
//                  thread has not executed yet.Once the
//                  scheduler schedule this task, the
//                  "Executed" will be set to 1 indicating
//                  that task has started the execution.
//                  if FastCtxtSwitch is set to 1 then the
//                  thread's context is saved and restored
//                  from internal register file and if it
//                  is set to 0 then the contest is saved
//                  and restored from external RAM.
////////////////////////////////////////////////////////
void createTask(int TaskID, void *funcptr, unsigned char cnxt_type)
 {
     Threads[TaskID].TaskID = TaskID;
     Threads[TaskID].Executed = 0;
     Threads[TaskID].TaskPtr = (TaskFunc)funcptr;

     if(cnxt_type == 0)
     {
         // Context switch using internal register files
         Threads[TaskID].State = 0;
         Threads[TaskID].FastCtxtSwitch = 1;

     }else
     {
         // Context switch using external RAM
         Threads[TaskID].State = Context + (TaskID * CONTXT_SIZE);
         Threads[TaskID].FastCtxtSwitch = 0;
     }
     return;
 }
```

```
//////////////////////////////////////////////////////////////
// Function     :   schedule()
//
// Parameters   :   -   void
//
// Return       :   -   void
//
// Desription   :   This function is heart of this co-operative
//                  real time operating system. This function
//                  actually starts the OS and does the context
//                  switching between tasks in round robin fashion.
//////////////////////////////////////////////////////////////////
 void schedule(void)
 {
     int ret;
     void (*fp)();


     if(Threads[TaskNext].Executed == 0)
     {
         // we are going to execute this task first time
         // so start this task from the task function received
         // at the time of task creation.
         fp = Threads[TaskNext].TaskPtr;

         Threads[TaskNext].Executed = 1;
         fp = Threads[TaskNext].TaskPtr;
         fp();

     }else
     {
         if(Threads[TaskNext].FastCtxtSwitch == 0)
         {
             // save context on the external RAM
             ret = setjmp(Threads[TaskNext].State);
         }else
         {
             // Save context on the Internal register files
             ret = fast_setjmp(TaskNext);
         }

         if(ret)
         {
             // we just returned from Longjmp so return
             // to execute the new task
             return;
         }
     }

     TaskNext++;
     if(TaskNext > MAX_THREADS - 1)
```

```
    {
        TaskNext = 0;
    }

    if(Threads[TaskNext].Executed == 0)
    {
        // we are going to execute this task first time
        Threads[TaskNext].Executed = 1;
        fp = Threads[TaskNext].TaskPtr;
        fp();
     }else
     {
         if(Threads[TaskNext].FastCtxtSwitch == 0)
         {
             // Restore context from the external RAM
            longjmp(Threads[TaskNext].State);

         }else
         {
             // restore context from Internal Register files
             fast_longjmp(TaskNext);
         }
     }
 }

////////////////////////////////////////////////////////////
// Function     :   initOS()
//
// Parameters   :   -   void
//
// Return       :   -   void
//
// Desription   :  This funstion is called by the application to
//                 initialize the thread structure objects. By
//                 default all thread are initialized for context
//                 switching using external RAM. Application has
//                 to set the correct context switch requirement
//                 at the time if thread creation.
////////////////////////////////////////////////////////////////

 void initOS(void)
 {
     int i;

     // Inititailse all thread structures
     for(i = 0; i < MAX_THREADS; i++)
     {
         Threads[i].TaskID = 0;
         Threads[i].FastCtxtSwitch = 0;
     }
   return;
 }
```

APPENDIX B-2

```
#####################################################################
# FILENAME: boot.asm
# AUTHOR: Deepak Gauba
# DATE CREATED: 1/12/02
# PROJECT: Hardware Implementation of RTOS Context Switch
# DESCRIPTION:
#    Initializes the stack pointer and jumps to main().
#    Which intern calls context switch (Save and restore)
#    functions to switch the context using internal
#    Register files as well as using external RAM.
#####################################################################

   #Reserve 512 bytes for stack
   .comm InitStack, 512
   .text
   .align 2
   .global entry
   .ent     entry
entry:
   .set noreorder

   #These four instructions should be the first instructions
   #as are initializing the stack pointer. This is the basic
   #requirement for system to understand 'C'
   #convert.exe previously initialized $gp, .sbss_start, .bss_end, $sp

   la    $gp, _gp             #initialize global pointer
   la    $5, __bss_start      #$5 = .sbss_start
   la    $4, _end             #$2 = .bss_end
   la    $sp, InitStack+488   #initialize stack pointer

   jal   main
   nop

      .set reorder
   .end entry

#######################################################
   .global   setjmp
   .ent     setjmp
setjmp:
   .set noreorder
   sw    $16, 0($4)   #s0
   sw    $17, 4($4)   #s1
   sw    $18, 8($4)   #s2
   sw    $19, 12($4)  #s3
   sw    $20, 16($4)  #s4
```

```
   sw    $21, 20($4)  #s5
   sw    $22, 24($4)  #s6
   sw    $23, 28($4)  #s7
   sw    $30, 32($4)  #s8
   sw    $28, 36($4)  #gp
   sw    $29, 40($4)  #sp
   sw    $31, 44($4)  #lr
   jr    $31
   ori   $2,  $0, 0

   .set reorder
   .end setjmp


######################################################
   .global   longjmp
   .ent     longjmp
longjmp:
   .set noreorder
   lw    $16, 0($4)   #s0
   lw    $17, 4($4)   #s1
   lw    $18, 8($4)   #s2
   lw    $19, 12($4)  #s3
   lw    $20, 16($4)  #s4
   lw    $21, 20($4)  #s5
   lw    $22, 24($4)  #s6
   lw    $23, 28($4)  #s7
   lw    $30, 32($4)  #s8
   lw    $28, 36($4)  #gp
   lw    $29, 40($4)  #sp
   lw    $31, 44($4)  #lr
   jr    $31
   ori   $2,  $5, 0

   .set reorder
   .end longjmp


######################################################
   .global   fast_setjmp
   .ent     fast_setjmp
fast_setjmp:
   .set noreorder

   scxt  $4
   jr    $31
   ori   $2,  $0, 0

   .set reorder
   .end fast_setjmp

######################################################
   .global   fast_longjmp
```

```
   .ent     fast_longjmp
fast_longjmp:
   .set noreorder

   rcxt $4
   jr    $31
   ori   $2,  $5, 0

   .set reorder
   .end fast_longjmp
```

APPENDIX B-3

```
//////////////////////////////////////////////////////////////
// File Name     : DebugSerial.c
//
// Desription    :   This file implementsthe code to write
//                   debug messages on the debug serial port
//                   in ASCII format. The numbers are printed
//                   in hexadeciaml format before sending to
//                   the serial port.
//////////////////////////////////////////////////////////////


#include "plasma.h"

#define MemoryRead(A) (*(volatile unsigned int*)(A))
#define MemoryWrite(A,V) *(volatile unsigned int*)(A)=(V)

//////////////////////////////////////////////////////////////
// Function      :   xtoa()
//
// Parameters    :   int num - input integer
//
// Return        :   char * - pointer to the string containing
//                            ASCII characters of the given hex value
//
// Desription    :   This function converts the given integer
//                   to ASCII string of its hex value
//////////////////////////////////////////////////////////////
char *xtoa(unsigned long num)
{
   static char buf[12];
   int i, digit;
   buf[8] = 0;
   for (i = 7; i >= 0; --i)
   {
      digit = num & 0xf;
      buf[i] = digit + (digit < 10 ? '0' : 'A' - 10);
      num >>= 4;
   }
   return buf;
}
```

```
//////////////////////////////////////////////////////////
// Function     :   putchar()
//
// Parameters   :   int num - Value to send on UART
//
// Return       :   - void
//
// Desription   :   This function writes the given value
//                  on the UART write address.This function
//                  is mainly being used to send debug
//                  messages at serial port.
//////////////////////////////////////////////////////////
void putchar(int value)
{
   while((MemoryRead(IRQ_STATUS) & IRQ_UART_WRITE_AVAILABLE) == 0)
      ;
   MemoryWrite(UART_WRITE, value);
   return ;
}

//////////////////////////////////////////////////////
// Function     :   puts()
//
// Parameters   :   char *
//
// Return       :   - void
//
// Desription   :   This function is used to print debug
//                  messages on the terminal via serial port.
//////////////////////////////////////////////////////
void puts(const char *string)
{
   while(*string)
   {
      if(*string == '\n')
      {
          putchar('\r');
      }
      putchar(*string++);
   }
   return;
}
```

APPENDIX B-4

```
////////////////////////////////////////////////////////////
// File Name    : plasma.h
// Description  : This header file defines the Plasma processor
//                address map
////////////////////////////////////////////////////////////
#ifndef __PLASMA_H__
#define __PLASMA_H__

#define FAST    0
#define NORM    1

//*********** Hardware addesses ***********
#define RAM_INTERNAL_BASE 0x00000000 //8KB
#define RAM_EXTERNAL_BASE 0x10000000 //1MB
#define RAM_EXTERNAL_SIZE 0x00100000
#define UART_WRITE        0x20000000
#define UART_READ         0x20000000
#define COUNTER_REG       0x20000060
#define FLASH_BASE        0x30000000
```

APPENDIX B-5

```
//////////////////////////////////////////////////
// File Name   : co_op_rtos.h
// Description : This file provides the co-operative
//               RTOS interface to the application
//               software.
//////////////////////////////////////////////////

#ifndef CO_OP_RTOS_H
#define CO_OP_RTOS_H

void createTask(unsigned int TaskID, void* funcptr, unsigned char
cnxt_type);
char *xtoa(unsigned long num);
void schedule(void);
void initOS(void);
int puts(const char *string);

#endif
```

APPENDIX C-1

```
//////////////////////////////////////////////////////////////
// File Name     : Application_1.c
//
// Author        : Deepak Gauba
//
// Date          : 19th December, 2009
//
// Desription   :   This file implements a Basic application to
//                  test the newly implemented hardware and
//                  operating system. This file call operating
//                  system functions to create four threads and
//                  then start the operating system.
//////////////////////////////////////////////////////////////

#include "co_op_rtos.h"
#include "plasma.h"

 int a,b,c,d,e;
 int PrevCount;
 int PrevData;

//////////////////////////////////////////////////////////////
// Function     :   Task0()
//
// Parameters   :   -   void
//
// Return       :   -   void
//
// Desription   :   This is the first thread of the application
//                  This thread increments four global variables
//                  in while loop and after incrementing the varables
//                  call 'schedule' function to relesae the control
//                  to the next thread in the queue
//////////////////////////////////////////////////////////////

 void Task0()
 {
     while(1)
     {
         PrevCount = *(volatile int*)COUNTER_REG;
         puts("Task0 :  Incrementing Variables \n");

         a++;
         b++;
         c++;
         d++;
```

```
        schedule();
    }
}

//////////////////////////////////////////////////////////////////
// Function     :   Task1()
//
// Parameters   :   -   void
//
// Return       :   -   void
//
// Desription   :   This function is part of the thread 1.
//                  This thread add the global variables and
//                  store the results in another global variable.
//                  This thread also runs in never ending loop and
//                  after each addition calls the OS function
//                  'schedule' to release the control to the next
//                  thread.
//////////////////////////////////////////////////////////////////
void Task1(void)
{
    while(1)
    {
        puts("Task1 : Adding a, b, c, d \n");
        e = a + b + c + d;
        schedule();
    }
}
//////////////////////////////////////////////////////////////////
// Function     :   Task2()
//
// Parameters   :   -   void
//
// Return       :   -   void
//
// Desription   :   This function is part of the thread 2.
//                  This thread prints all the current values of the
//                  global variables on the debug serial port and
//                  releases the control to the next thread in
//                  the queue
//////////////////////////////////////////////////////////////////

void Task2(void)
{
    while(1)
    {
        puts("Task2 : a = 0x");
        puts(xtoa(a));
        puts(", b = 0x");
        puts(xtoa(b));
        puts(", c = 0x");
        puts(xtoa(c));
```

```
            puts(", d = 0x");
            puts(xtoa(d));
            puts(", Sum = 0x");
            puts(xtoa(e));
            puts("\n");

            schedule();
        }
    }

////////////////////////////////////////////////////////////////
// Function     :   Task3()
//
// Parameters   :   -  void
//
// Return       :   -  void
//
// Desription   :   This function is part of the thread 3.
//                  This thread prints the total number of cycles
//                  taken to execute all four threads.
////////////////////////////////////////////////////////////////

    void Task3(void)
    {
        int diff;
        int Ticks;
        while(1)
        {
               Ticks = *(volatile int*)COUNTER_REG;
               diff = Ticks - PrevCount;
               puts("Task3 : Ticks Taken for whole process = 0x");
               puts(xtoa(diff));
               puts("\n \n");
               PrevData = diff;
               schedule();
        }
    }
////////////////////////////////////////////////////////////////
// Function     :  main()
//
// Parameters   :   -  void
//
// Return       :   -  void
//
// Desription   :   This function is the main entry point of the
//                  application. This is called from the boot.asm
//                  file after initializing the stack pointer.
////////////////////////////////////////////////////////////////

int main(void)
{
      // initialize the global variables
      a = 1;
```

```
        b = 2;
        c = 3;
        d = 4;

        // Initialize OS structure objects
        // for all the threads
        initOS();

        // Create four threads with fast
        // context switch setting
        createTask(0, Task0, 0);
        createTask(1, Task1, 0);
        createTask(2, Task2, 0);
        createTask(3, Task3, 0);

        // Start the OS by scheduling the first Thread
        schedule();

    return 0;
}
```

APPENDIX C-2

```
////////////////////////////////////////////////////////////////
// File Name     : Application_2.c
//
// Author        : Deepak Gauba
//
// Date          : 29th December, 2009
//
// Desription   :   This file implements an application to
//                  calculate and print the number of clock cycles
//                  taken for the context switch using internal
//                  register files and cycles taken for the context
//                  switch using external RAM. The application
//                  also calculates the performance improvement
//                  in terms of clock cycles saved.
////////////////////////////////////////////////////////////////

#include "co_op_rtos.h"
#include "plasma.h"

 int PrevCount;
 int PrevData;

////////////////////////////////////////////////////////////////
// Function     :   Task0()
//
// Parameters   :   -   void
//
// Return       :   -   void
//
// Desription   :   This is the first thread of the application
//                  and stores the current clock cycles counter
//                  value in a global variable and releases the
//                  control to the next thread.
////////////////////////////////////////////////////////////////

 void Task0()
 {
     while(1)
     {
         // save the current clock cycle count
         PrevCount = *(volatile int*)COUNTER_REG;
         // schedule the next thread
         schedule();
     }
 }
```

```
//////////////////////////////////////////////////////////////
// Function     :   Task1()
//
// Parameters   :   -   void
//
// Return       :   -   void
//
// Desription   :   This function reads the current clock cycle
//                  counter value and calculate and prints the
//                  difference between the current and previous
//                  value and then store the difference in a
//                  global variable for further processing.
//////////////////////////////////////////////////////////////////
 void Task1(void)
 {
     int diff;
     int Ticks;
     int i;
     while(1)
     {
             // save the current clock cycle count
            Ticks = *(volatile int*)COUNTER_REG;
            // print the difference between current and previous
            diff = Ticks - PrevCount;
            puts(xtoa(diff));
            puts(",");
            // Store the clock cycles taken for the first
            // context switch
            PrevData = diff;
            // schedule the next thread
            schedule();
     }
 }
//////////////////////////////////////////////////////////////
// Function     :   Task2()
//
// Parameters   :   -   void
//
// Return       :   -   void
//
// Desription   :   This function has the same code as Task0 and
//                  and stores the new value of current cycle counter
//                  in the same global variable.
//////////////////////////////////////////////////////////////////

 void Task2(void)
 {
      // save the current clock cycle count
      PrevCount = *(volatile int*)COUNTER_REG;
      // schedule the next thread
      schedule();
 }
```

```
//////////////////////////////////////////////////////////////
// Function     :   Task3()
//
// Parameters   :   -  void
//
// Return       :   -  void
//
// Desription   :   This function is part of the last thread which
//                  calculates the performace improvement in terms of
//                  clock and cycles and print the results on debug
//                  serial port.
//////////////////////////////////////////////////////////////////

 void Task3(void)
 {
     int diff;
     int Ticks;
     int Gain;
     int i;
     while(1)
     {
            // save the current clock cycle count
            Ticks = *(volatile int*)COUNTER_REG;
            diff = Ticks - PrevCount;
            puts(xtoa(diff));
            puts(",");
            // calculate the performace improvement
            // in term of clock cycle
            Gain = diff - PrevData;
            puts(xtoa(Gain));
            puts("\n");
            // Schedule the first thread again
            schedule();
     }
 }
//////////////////////////////////////////////////////////////////
// Function     :  main()
//
// Parameters   :   -  void
//
// Return       :   -  void
//
// Desription   :   This function is the main entry point of the
//                  application. This is called from the boot.asm
//                  file after initializing the stack pointer.
//////////////////////////////////////////////////////////////////

int main(void)
{
      // initialize the global variables
      a = 1;
      b = 2;
      c = 3;
```

```
        d = 4;
        // Initialize OS structure objects
        // for all the threads
        initOS();

        // Create two threads with fast
        // context switch setting
        createTask(0, Task0, 0);
        createTask(1, Task1, 0);
        // Create two threads with setting
        // for context switch on external RAM
        createTask(2, Task2, 1);
        createTask(3, Task3, 1);
        // Start the OS by scheduling the first Thread
        schedule();

    return 0;
}
```