Project 1: Timer Interrupt Handler

Rees Klintworth & Derek Nordgren

October 5, 2015

Introduction

Project goals

Our initial goals for this project include generally getting familiar with NIOS-II development: the IDE, Altera-specific programming, compiling and debugging our own code onto the board, etc. We used this project as an opportunity to build experience that will benefit us later on in building our prototype OS kernel.

Another of our goals was to gain a thorough understanding of interrupt declaration and handling in the NIOS-II environment. We learned the various function calls related to initializing, triggering, and handling interrupts. In addition, we disassembled our project's source and were able to explore the assembly code underlying the project's operation.

Problem description

This project focuses on simple interrupts with the Altera DE-2 board. Interrupts will be achieved by initializing an alarm that will interrupt at scheduled intervals and an interrupt handler. The alarm will utilize the DE-2's on-board clock and will cause the interrupt handler to be called at a controlled interval as a callback of a very simple function execution.

The project is broken down into four different task. The first task is to implement a simple provided program on the DE-2. The function to be executed was:

```
void uOS()
{
	while(1)
	{
		alt_printf("Hello from uOS!\n");
		int j;
		for (j = 0; j < 10000; j++);
	}
}
```

Interrupt handler design

The alarm is initialized with a call to `alt_alarm_start`. The exact code for this function can be seen in the Solution Overview section. As a part of initializing the alarm, a callback (interrupt handling) function had to be declared. The design of the function handling the interrupt is relatively simple; it accepts a context from the calling function and returns an `alt_u32` indicating what value the alarm's timer should be reset to.

The third step was to insert an assembly NOP command (No Operation) directly into the code using `asm("nop")`, and observe the behavior of the program. The command could be placed anywhere in the program.

The final step was to analyze the impact of inserting `asm("nop")` by debugging the program, paying special attention to the inclusion of the nop command

Project management

We began our work on this project two days before the due date. As a team, we reviewed the assignment shortly after it was assigned and determined that we would be able to complete the project in a relatively short amount of time. Prior to implementing a solution, each team member reviewed applicable sections of the NIOS-II Software Developer's Handbook.

Our team spent approximately 30 minutes creating the project and writing code to address the problem. We then spent around an hour debugging and learning about the function of our solution and testing different values for variables in the program.

Our team worked together on this assignment for the board programming portion. Each team member then completed different sections of the report.

This project was fairly low risk, because the requirements were specific and relatively easy to understand. We utilized few resources, focusing on resources provided in class (such as PowerPoint slides and assignment descriptions), and the NIOS-II Software Developer's Handbook.

Solution

We accomplished the tasks of the project (implement alarm-based interrupts on the NIOS-II board) by supplementing code provided as a part of the assignment with code obtained from the NIOS-II Software Developer's Handbook. We verified our solution by validating that the alarm interrupted our program periodically based on the values for variables that we selected.

For example, setting `MAX` (the value governing the number of iterations that the `for` loop spins for) to a larger value while maintaining the value determining the length of time the alarm will be reset for (the return value of the callback function) causes fewer non-interrupt print statements to be printed between interrupt print statements. Decreasing `MAX` causes more non-interrupt print statements to be printed, as expected. We chose a value of 10000 for most of our testing. This value allowed enough print statements to execute that we could observe differences with different return values in our callback handler, while making the output easy to view and study.

Using the defined macro `ALARMTICKS(x)`, we were able to replace x with various numeric values. Whatever we chose for x would be divided by 10, and that number equaled the seconds between each alarm interrupt (excluding the first call, which was initialized with the alarm to be `alt_ticks_per_second()`). We were able to modify this value and observe the behavior, and as expected

doubling x would result in approximately double the print statements from `uOs` being executed between each interrupt.

Inserting a nop assembly command has very little effect on the overall program. It does take up one clock tick, but the impact is not enough to noticeably affect program execution. Similarly, the assembly code is not impacted in any major way.

We were able to implement simple, scheduled interrupts and interrupt handlers on our board during this project. This is a topic that we covered at length in class and that will be integral to the operation of our prototype OS later on in the class.

Evaluation of Solution

Project organization

For this project, our project is organized as a single file that contains our main function, a prototype OS function, and our callback function for when the interrupt alarm is triggered.

Solution overview

Our solution successfully meets the criteria for the project, although it is not very creative or unique. This project was well-defined and had a small scope, so there was not much room for variation within our solution. Much of the code was already defined for us, such that we only needed to add code specific to starting a timer, and connecting that to our interrupt handler.

Our design and implementation were very simple, as we only added a few lines of code in addition to what we were given to begin with. We defined a macro to determine how long until the alarm callback function would be executed again, and we defined and initialized an alarm. Our goal was to achieve the requirements of the project with as little extra code as possible.

The solution is broken into four steps. The first step is to implement a simple OS function that will print a line to the console and then a for-loop that executes for a specific amount of time (can be increased or decreased by changing how many times the code will loop) before repeating.

The next step is to initialize an alarm before the above-mentioned for-loop. The alarm can be declared as follows:

```
alt_alarm_start(&alarm, alt_ticks_per_second(), interrupt_handler, NULL);
```

Once program execution begins, the alarm will trigger its interrupt handler callback after `alt_ticks_per_second()` ticks (which is the number of ticks on the board corresponding to a second in real-time) have expired.

From here, the callback function `interrupt_handler(void* context)` is triggered. Any code can be run in this function. We chose to execute a simple print statement. The return value indicates the value the alarm should be reset to; 0 cancels the alarm and any positive real value resets the alarm to that value, scheduling another interrupt.

Finally, program execution returns to the loop and the cycle repeats.

Relationship to Course Material

Conceptually, this project is straightforward. Interrupts are a basic concept that are crucial to any operating system being able to manage multiple processes, inputs, etc. We were able to learn about the correspondence of interrupts to the hardware clock of the DE-2 board, and will be able to use this simple implementation as a building block for later, more complex projects.

Suggested Project Improvements

Our team enjoyed this project, and the simplicity and straightforwardness meant that we were able to tackle the problem and understand the concepts fairly quickly. It would be nice, however, if this project went one step further in terms of implementing and using an interrupt. The concept is not difficult to understand, but in practice we are still inexperienced with interrupts and their uses. It would be interesting to incorporate one or two more steps where we take advantage of our interrupt to accomplish a specific task.

References

NIOS II Software Developer's Handbook. (2011). Altera Corporation.