The University of Waikato
Department of Computer Science

COMPX203-22A Computer Systems
Exercise 4 – Exceptions and Interrupts

Due Date: 02 June 2022

Objectives

The objective of this exercise is to reinforce the ideas presented in lectures on exceptions and interrupts. To achieve this, you'll be creating a stopwatch program that utilises the interrupt and exception handling mechanisms of the WRAMP CPU.

*It is highly recommended that you read this entire specification **before** beginning to write any code.*

Assessment

This exercise contributes **5%** towards your **internal grade**, to be assessed in two parts:

- The correctness of your source code, to be submitted via Moodle **(weight 0.7)**
- Your completion of an online quiz about the exercise, also on Moodle **(weight 0.3)**

Both of these parts must be completed on or before the due date.

This is an individual exercise; while you may discuss the assignment with others in general terms, you must write your own code, and complete the quiz by yourself.

The name of the file must match the following format:

ex*[exercise number]*_q*[question number]*.srec

For example, the first question in this exercise would be called ***ex4_q1.srec***

Introduction

In this exercise, you'll be creating a stopwatch program for the Basys board, using the interrupt handling features of the WRAMP CPU. Each question builds on the previous question, so make sure that each step is working flawlessly before moving on – fixing bugs while the program is small is *much* easier than fixing them later! Remember to keep copies of the code for each question, because they all need to be handed in as part of your submission. Feel free to use the *mark200* tool to make sure each step is working correctly.

Before starting this exercise, you'll need to read **Chapter 4** of the WRAMP manual.

Notes

This is probably the most challenging exercise in the course, with the exception of the upcoming multitasking kernel assignment. Be kind to yourself and start early, so that you can make the most effective use of the supervised lab sessions if you get stuck. Work in small steps, and make sure you understand what you're doing before attempting to write any code. If in doubt, ask!

Questions

1. **Counter**

Write an assembly program (called **ex4_q1.s**) that counts the number of times that any button (**User Interrupt Button and/or Push Buttons 0, 1, or 2)** is pressed. This will consist of two parts:

a. A mainline loop that continuously reads from a location in memory (perhaps called *counter* in the .data segment), and writes its value to the SSDs as a **two-digit decimal number**.

b. Exception handlers for the **User Interrupt Button** and **Parallel Port Interrupt**. These handlers should simply increment *counter* by one.

Make sure that your exception handlers are *compliant*; that is, they must not alter the value stored in any register other than **$13**. Also, make sure that you correctly call the default exception handler for any exceptions that your code doesn't handle directly.

Note: pressing both push buttons should have the same effect as a single button being pressed, and make sure that other parallel port interrupts (i.e. from the switches changing state) are ignored.

2. **Timer**

Create a copy of your code from **question 1**, and call it **ex4_q2.s**. Now modify the code so that the SSDs show the number of seconds that the program has been running. To do this, set up the **Programmable Timer** so that it generates one interrupt per second, and update your exception handling code accordingly. ALL buttons should no longer have any effect.

3. **Stopwatch**

Add the ability to start, stop, and reset the stopwatch with the push buttons. To do this, create a copy of your code from **question 2** (call it **ex4_q3.s**) and add a second exception handler that responds to interrupts from the **Parallel Port**:

a. **Push Button 1** should start (or resume) the stopwatch if it is not currently running, or stop (pause) the stopwatch if it is currently running. *Hint: just toggle the Timer Enable bit*.

b. **Push Button 0** should reset the counter to zero if the timer is **not** currently running. If the timer **is** currently running, this button should do nothing.

c. **Push Button 2** should terminate the program (e.g. `jr $ra` from the context of `main`). *Hint: use a termination flag that is set when handling ~~both~~ buttons~~s~~ 2 is pressed. Remember not to use a `syscall` here!*

Note: make sure that other parallel port interrupts (i.e. from the switches changing state) are ignored.

Now that you have a start/stop button, modify your timer initialisation so that the timer is **not running** when the program first starts. In other words, the timer should only start running when the user presses Push Button 1.

4. **Amazing Stopwatch, v2.0**

Add the ability to record lap times, with a resolution down to one hundredth of a second.
To do this, create a copy of your code from **question 3** (call it **ex4_q4.s**) and do the following:

a. Modify the timer initialisation so that it increments the *counter* value 100 times per second while it is running.
b. Modify the code that prints *counter* to the SSDs, so that it still displays seconds elapsed, i.e. *counter* divided by 100.
c. Modify the exception handler for **Push Button 0** so that it prints the current value of the counter once to **Serial Port 2**, including the hundredths of a second. This printing should only happen if the button is pressed *while the timer is running* – otherwise, this button should reset the counter to zero as it did in the previous question.

The string printed to the serial port must be in the format "\r\nss.ss", where '\r' is the carriage return character, '\n' is the newline character, and "ss.ss" is the number of seconds elapsed, including hundredths.

*Note: steps **a** and **b** are reasonably straightforward, but **c** requires a bit more work. Recall that slow operations like polled I/O should never be done in an exception handler; instead, either use polled I/O in the mainline code (recommended approach), or use interrupt-driven I/O (better, but much more complicated!). You may find it useful to begin by transmitting a single character to the serial port when the lap button is pressed, before trying to transmit the properly-formatted string.*

Model Solutions

For reference, model solutions are (for each question):

1. 90 lines, 40 of which are assembly instructions
2. 77 lines, 32 of which are assembly instructions
3. 131 lines, 65 of which are assembly instructions
4. 194 lines, 94 of which are assembly instructions

If you're writing significantly more than this, chances are that you're doing something wrong, or are missing a potentially simpler approach. In this case, please ask for assistance at one of the supervised lab sessions. *Note that the line counts quoted are for model solutions that have fully-compliant exception handlers, but do not comply with the WRAMP ABI for subroutines in the mainline code. ABI compliance is not necessary for this exercise.*

Submission

You are required to submit all source files that **you** have written; you do not need to submit any files that we have provided, nor any files that have been generated by a tool, e.g. *wcc*, *wasm* or *wlink*. Each file must follow the naming convention indicated below.

For this exercise, the required files are:

- **ex4_q1.s** (Question 1)
- **ex4_q2.s** (Question 2)
- **ex4_q3.s** (Question 3)
- **ex4_q4.s** (Question 4)

These files must be compressed into a single **tar.gz** archive called **firstName_lastName.tar.gz** (replace *firstName_lastName* with your own name) before being submitted to Moodle. You can create this archive from the terminal, using the command:

tar -cvzf firstName_lastName.tar.gz ex4_q1.s ex4_q2.s ex4_q3.s ex4_q4.s

Please be aware that a grade penalty applies if you do not submit your code in the correct format, including using the correct *file names* and *file extensions*. If you are unsure if what you've done is correct, please ask and we'll help you out.