The University of Waikato
Department of Computer Science

COMPX203-21A Computer Systems
Exercise 3 – Parallel and Serial I/O

Due Date: Saturday 7 May 2022

Objectives

The objective of this exercise is to reinforce the ideas presented in lectures about input and output (I/O) devices. To achieve this, you'll be writing WRAMP assembly code that interacts with the I/O hardware directly, without the assistance of library subroutines.

*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 ***ex3_q1.srec***

Introduction

There are several I/O devices available on the Basys 3 board, and this exercise will deal with two in particular: serial ports (1 and 2), and the parallel port. In order to complete this exercise, you will need to read **Chapter 3** of the WRAMP manual.

Connecting to the second serial port is done in one of two ways:

- On lab machines where ***remote*** works, open up a new terminal window and run the command ***vt320***. To exit this terminal, press Ctrl+A followed by K, then hit Y to confirm. Our ***vt320*** program is actually just a version of ***screen*** configured specially for the boards.
- If you're using the simulator, open the "Serial Port 2" form by selecting it from the bottom panel.

Each of these options provide the same functionality: any characters typed into these terminals will be sent to serial port 2 of the Basys boards, and any character received from serial port 2 of the Basys boards will appear in the terminal as an ASCII character.

For the remainder of this exercise, **VT320** will refer to either the virtual terminal, or the "Serial Port 2" form of the simulator, depending on which option you are using.

Notes:

How you organise your code for this exercise is entirely up to you, however the usual recommendations do apply:

- Tidy, well-organised code is less likely to contain bugs.
- Don't overcomplicate the program. Using more lines of code to perform the same job is usually a bad idea.
- Your code does not need to comply with the WRAMP ABI conventions, however doing this anyway is good practice and reduces the likelihood that bugs will occur.

Questions

1. **Serial Output**

Write an assembly program (called **ex3_q1.s**) that prints the entire lowercase and uppercase alphabet to the VT320 terminal, in order, from 'a' to 'z' and then from 'A' to 'Z'. Use polling, rather than interrupts. The program should terminate once it has finished printing.

To do this, you must interact **directly** with the serial port hardware, using the addresses described in the WRAMP manual. In other words, **do not use any library subroutines**. You may find it helpful to run the simulator, open up the "Serial Port 2 Registers" form, and see how your code interacts with the device registers. Also try typing into the "Serial Port 2" form and seeing how the device registers change.

Because this program is entirely contained in one file, you only need to specify a single object file when linking your code:

wlink -o ex3_q1.srec ex3_q1.o

2. **Serial I/O**

Write an assembly program (called **ex3_q2.s**) that:

a. Reads a character from **serial port 1** (use polling)
b. Checks if that character is a *lowercase letter*
   - If it is a lowercase alphabet letter, leave it as is
   - If it is any other character , replace it with a '*' character
c. Transmits the (modified) character back to **serial port 1**
d. Jumps back to the start to wait for the next character to arrive

Once this program is working correctly, you should be able to see characters that you type into either *remote* (with the real hardware), or the *Serial Port 1* form (with the simulator). Lowercase letters should appear unchanged, while other characters should be replaced with '*'.

3. **Parallel I/O**

Write an assembly program (called **ex3_q3.s**) that:

a. Waits for any of the three horizontal **push buttons** to be pressed (use polling).
b. Reads the value of the switches.
c. Checks which button was pressed:
   - If **push button 0** was pressed, leave the switch value as is
   - If **push button 1** was pressed, invert the switch value (i.e. flip all of the bits)
   - If **push button 2** was pressed, exit the program (Remember to back up and restore $ra using the stack)

   Remember that the parallel port buttons are numbered right to left. If multiple buttons are pressed, you can choose what the behaviour should be.
d. Displays the (modified) value on the SSDs as a four-digit hexadecimal number.
e. If the (modified) value is a multiple of 4, turn on all the LEDs. Otherwise, turn them off. Zero is considered a multiple of four.
f. Jumps back to the start to wait for the next button press.

You may find it helpful to run the simulator, open up the "Parallel Port Registers" form, and play with the switches and/or push-buttons to get an idea of how these device registers work.

4. **Cooperative Multitasking**

Create a new assembly program (called **ex3_q4.s**) that combines the functionality of the programs that you wrote for questions **2** and **3**. That is, it should satisfy the requirements of both questions simultaneously (copy **ex3_q2.s** and **ex3_q3.s** codes into **ex3_q4.s**).

The easiest way to achieve this is to use a rudimentary form of multitasking called *cooperative multitasking*. With this approach, each "process" will run for a short amount of time before voluntarily allowing the next process to run. A cooperative multitasking loop might look something like this:

```
main:
        jal serial_job        #based on main from ex3_q2
    jal parallel_job      #based on main from ex3_q3
    j main
```

You will need some additional processing to ensure that `parallel_job` can properly exit the program back to the WRAMPmon prompt. A good way to do this is to return a value from `parallel_job` according to the WRAMP stack conventions, where any non-zero value means the program should exit - similar to how many UNIX programs provide return values.

Each job does a small amount of work, and returns to give other processes a chance to run. For example, `parallel_job` could work almost exactly like `main` in question 3, but it will return after writing a value to the SSDs rather than looping back to the start.

You might notice that one major disadvantage of cooperative multitasking is that if one process hangs, other processes will never get a chance to run. For example, if the *serial* process simply waited for a character to be received, the *parallel* process might not be able to run for a long time. For this question you can avoid this pitfall by simply returning from `serial_job` if a character has not yet been received – after all, it will be checked again when `serial_job` is next called.

Model Solutions

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

1. 30 lines, 15 of which are assembly instructions
2. 28 lines, 14 of which are assembly instructions
3. 46 lines, 24 of which are assembly instructions
4. 92 lines, 47 instructions, majority copied from questions 2 and 3 with minor changes

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: these examples do not make use of the stack, and are not compliant with the WRAMP ABI. If your programs do make use of the stack, you can expect to have more lines of code.

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:

- **ex3_q1.s** (Question 1)
- **ex3_q2.s** (Question 2)
- **ex3_q3.s** (Question 3)
- **ex3_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 -zcf firstName_lastName.tar.gz ex3_q1.s ex3_q2.s ex3_q3.s ex3_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.