Computer Science 230
Computer Architecture and Assembly Language
Summer 2020

Assignment 3

Due: Monday, July 20th, 11:55 pm by conneX submission
(Late submissions **not** accepted)

Programming environment

For this assignment you must ensure your work executes correctly on the MIPS Assembler and Runtime Simulator (MARS) as was installed during Assignment #0. Assignment submissions prepared with the use of other MIPS assemblers or simulators will not be accepted. ***Solutions which prompt the user for input will not be accepted.***

Individual work

This assignment is to be completed by each individual student (i.e., no group work). Naturally you will want to discuss aspects of the problem with fellow students, and such discussion is encouraged. **However, sharing of code fragments is strictly forbidden without the express written permission of the course instructor (Zastre).** If you are still unsure regarding what is permitted or have other questions about what constitutes appropriate collaboration, please contact me as soon as possible. (Code-similarity analysis tools will be used to examine submitted work.) The URLs of significant code fragments you have found and used in your solution must be cited in comments just before where such code has been used.

Objectives of this assignment

- Use system calls for rudimentary output to the MARS output console.
- Write an interrupt handler for the keyboard of the "Keyboard and Display MMIO Simulator" tool in MARS, and write a program that takes advantage of the behavior of the handler.
- Write a procedure to render a four-pixel by four-pixel box on the "Bitmap Display" tool.
- Combine both tools used earlier in this assignment in order to control the movement of a box on the bitmap display.

With this assignment on conneX is an HTML file containing a link to video that demonstrates the behavior of all four parts of this assignment.

Part (a): Write procedures **`dump_array`**

```
dump_array:
```

parameters:
 `$a0` holds the address of first integer in an array of integers
 `$a1` holds the number of integers to be output on the console in a single line

return value: *none*

file that must be used: `a3-part-A.asm`

A lab for the course will briefly cover MIPS32 system-call interface implemented in MARS. However, in a physical computer such as your laptop or desktop, any use you make of the computer's hardware such as display, keyboard, or input/output devices used privileged actions performed by the operating system. That is, in order to ensure safety, security, and robustness of a running computer, user programs are unable write code (i.e., machine instructions) that directly access physical hardware. Instead, users must request actions be taken with the hardware via a more secure system call.

The MARS system is an interesting combination in that it simulates real hardware – that is, you can write programs as if they were running on top of a physical MIPS32 computer – but also it provides some of the typical system-call actions offered by most operating systems.

Your task in this program is to use two different system calls in MARS:

- System call 1 is used to print an integer on the console; the integer to be printed is stored in `$a0`, while the value 1 is stored in `$v0`.
- System call 4 is used to print a null-terminated string on the console; the address of the string to be printed is stored in `$a0`, while the value 4 is stored in `$v0`.

In both cases, the `syscall` operation follows storage of values into required registers.

Your task in this part of the assignment is write `dump_array` which prints in one line on the "Mars Messages" panel the integer values stored in the array passed as the first parameter. The second parameter to the function is the number of integers in that array. For example, given the provided assembly-code for this file (`a3-part-A.asm`) following lines of code in the main program:

```
    la $a0, ARRAY_A
    addi $a1, $zero, 4
    jal dump_array
```

your function must produce the following on the "Mars Messages" pane:

```
21 210 49 4
```

Therefore your function will loop through the array, retrieve values from this array, call the correct system call for integer output, and also call the correct system call for string output(i.e., for a space and for a newline). Please sure that a newline is output at the end of each such line.

Part (b): Complete `a3-part-B.asm`

The concept and mechanism of machine exceptions and interrupts are to be covered in lectures and labs. This description, therefore, is not meant to be a substitute for that material. In summary, however, you must be able to answer the following questions:

- How are interrupts on a specific MARS device enabled or disabled?
- Where is code transferred into the kernel upon an interrupt occurrence?
- How can you determine in the kernel which MARS device generated the interrupt?
- How can you transfer code within the kernel to the handler for the specific interrupt that was generated?
- How can you transfer information from and to data memory and the device via the interrupt handler?
- How do you ensure the interrupt handler performs the absolute minimum of computation(i.e., what is the division of labour between code in the `.ktext` area and code in the `.text` area)?

Writing code that involves interrupts is perhaps some of the trickiest and most difficult kind of computer programming. This is not because algorithms are complicated or because much code is required, but rather because so little code is needed yet that code must be very precisely written. Think of it as judo programming: short matches with very, very quick action, and most of the time you find yourself in trouble on the judo mat!

However, you do have one major benefit when exploring such MIPS32 programming while using MARS as the you can slow down the run speed of the program. It is hard to overemphasize how big an advantage this is compared to such programming using physical hardware.

Your goal for this part of assignment is to produce a program with the following behavior:

- Write an interrupt handler for the "Keyboard and Display MMIO Simulator" tool. One of the labs will discuss the "keyboard" part of this MARS tool in more detail.

- The handler is to transfer the pressed-key's value into data memory, and indicate that a keyboard event is "pending" (i.e., code based on the key pressed must still be executed).
- Non-handler code (i.e., code outside of `.ktext`) keeps track how many times the `'a'`, `'b'`, `'c'` , and `'d'` keys have been pressed.
- If the space key is pressed, the counts are output on the "Mars messages" pane (again through code outside of `.ktext`).
- All other key presses are ignored.
- After finishing all processing for a keypress, the program must now indicate there is no longer a pending keyboard event.

You are forbidden to write an interrupt handler which either uses the `syscall` operation or calls functions that themselves ultimately call `syscall`. In fact, the interrupt handler must not call any functions at all.

Therefore your finished program will have code you have written in the `.text` area (made up of instructions within an infinite loop) and code you have written in the `.ktext` area (your code to dispatch to the correct handler).

Part (c): Write the procedure `draw_bitmap_box`

`draw_bitmap_box:`

parameters:
 `$a0` holds the row # of the upper-left corner of the box
 `$a1` holds the column # of the upper-left corner of the box
 `$a1` holds colour of the box

return value: *none*

files that must be used: `a3-part-C.asm, bitmap-routines.asm`

In one of the labs you will be shown the "Bitmap Display" tool available in MARS. It can be somewhat difficult tool to use without previous knowledge. Therefore you are provided with an assembly file which permits you to call a procedure that sets a pixel in the display to a colour.

For this part of the assignment the "Bitmap Display" tool behaves as a 16x16 bitmap display. The code in `bitmap-routines.asm` makes this possible **by assuming you have set up the tool as follows**:

- Unit Width in Pixels: 32
- Unit Height in Pixels: 32
- Display Width in Pixels: 512
- Display Height in Pixels: 512
- Base address for display: 0x10010000 (static data)

An exercise in one of the labs will give you some experience of working with the display and with the `set_pixel` procedure provided in `bitmap-routines.asm`. Also explained in the lab will be the way a 32-bit value for a pixel colour is interpreted as a 24-bit RGB value.

Your own work for the `draw_bitmap_box` procedure is to take the parameters and draw a four-by-four pixel box where row/column location of upper-left hand corner of that box is given as the first two parameters. You may assume: row 0, column 0 is the upper-left corner of the complete display; row 15, column 15 is the lower-right corner of the complete display.

Part (d): Complete `a3-part-D.asm`

The previous parts of this assignment have led up to the last part. You are to combine the keyboard interrupts and the bitmap display in order to control the movement of a four-by-four box on the display.

Initially the four-by-four box will in the top-left of the display (i.e., box-coordinate is row 0, column 0). The following keys will "move" the box:

- `'d'`: move the box right by one pixel
- `'a'`: move the box left by one pixel
- `'w'`: move the box up by one pixel
- `'x'`: move the box down by one pixel

Note that "moving" here means erasing the box in its current location (i.e., re-drawing the box with colour `0x00000000`), and the re-drawing the box in its new location. Therefore you must maintain the values `BOX_ROW` and `BOX_COLUMN` (and remember that the interrupt handler you complete in `.ktext` **must not modify these values** – only code in the `.text` is permitted do modify them).

To reduce some of the complexity of your solution, you do not need to worry about keeping the box in its boundaries (i.e., depending on what keys are pressed, the box might end up moving off the screen). Also don't worry about the way the box might flicker as it moves.

What you must submit

- Your completed work in the four assembly files: `a3-part-A.asm`, `a3-part-B.asm`, `a3-part-C.asm`, and `a3-part-D.asm`.

Evaluation

- 4 marks: Solution part A.
- 6 mark: Solution for part B.
- 4 marks: Solution for part C.
- 6 marks: Solution for part D.

Therefore the total mark for this assignment is 20.

Some of the evaluation above will also take into account whether or not submitted code is properly formatted (i.e., indenting and commenting are suitably used), and the file correctly named.