ECS 50: Programming Assignment #4

Instructor: Aaron Kaloti

Fall Quarter 2021

Contents

1 Changelog

You should always refer to the latest version of this document.

- v.1: Initial version.
- v.2: Added grading breakdown and submission details. Added the following clarifications/updates (all marked in red), most notably:
  - Clarified that the entry point of the cartridge need not be the instruction that ends up at address `0x20000000`.
  - To the end of the part #1 directions, added a note about use of the `.end` directive.
  - Added note about how to run the simulator in normal mode. (All you've to do is omit the `-d` flag.)
  - Stated explicitly that you are not allowed to modify the signature of `getInput()`.
  - Added tip about debugging to end of directions for part #2.

2 General Submission Details

Partnering on this assignment is prohibited. If you have not already, you should read the section on academic misconduct in the syllabus.

This assignment is due the night of Monday, 11/29. Gradescope will say 12:30 AM on Tuesday, 11/30, due to the "grace period" (as described in the syllabus). *Be careful about relying on the grace period for extra time; this could be risky.*

3 Programming Problems

You will be using the same RV32EM simulator as before.

Update: To run the simulator in normal mode (instead of debug mode), use the same command that you were previously using to run the simulator, but without the `-d` flag.

You will ONLY submit the files `loader.s` and `get_input.c`. You are effectively not allowed to modify any other file, since I will not see the effects of such modifications when I grade your submission.

*This content is protected and may not be shared, uploaded, or distributed.

I will (or already did, depending on when you read this) talk about this assignment (and take any questions on it) during the 11/19 lecture.

3.1 File Organization

I recommend that you organize the files for this assignment into four folders:

- `cartridge_pokemon/`
- `cartridge_super_mario/`
- `get_input/`
- `loader/`

`cartridge_pokemon/`, `cartridge_super_mario/`, and `loader/` are for part #1; `get_input/` is for part #2. You may notice that this is the organization used with the provided code on Canvas, so in other words, you should keep the file organization that I already set up for you. Part of the reason for this file organization is that there are some files – e.g. `crt0.s`, `Makefile`, `riscv32-console.ld` – that appear in each folder. Each folder is meant to correspond to one executable, although in the case of `get_input/` (the folder for part #2), you could set things up so that there are multiple executables, if you're willing to tweak the makefile and add more `main()` implementations.

Needless to say, it is highly recommended that you start with all of the files provided to you on Canvas. Note that files that have the same name are not necessarily the same. For instance, `get_input/crt0.s` sets up interrupts, whereas `loader/crt0.s` does not. Moreover, `riscv32-console.ld` slightly differs between the two cartridges, the loader program for part #1, and the program for part #2.

3.2 Restrictions

For part #1 of this assignment (`loader.s`), you will write RV32EM assembly code. All code that you write will be RISC-V code for a RV32EM processor. ***You must write this assembly code yourself; you cannot use a tool, such as a compiler[1], to generate assembly code. For instance, you cannot write C code and then use a compiler to convert it to RISC-V assembly code. We will not accept that, and we will easily be able to tell if you submitted tool-generated assembly code.***

For part #2 of this assignment (`get_input.c`), you will *only* be writing C code, so the above does not apply.

3.3 Part #1: Responding to a Cartridge Load

The only file that you should modify for this part (and the only file that you will submit for this part) is `loader.s`.

During lecture (most notably when going through slides #11 and #12 of slide deck #5), we talked a bit about cartridges in game consoles. A cartridge contains an executable/program, and running the executable that is in the cartridge will start whatever game is supposed to be in the cartridge.

In part #1, you will write a program (i.e. add code to `loader.s`) that responds to the loading of a cartridge. (More specifics on what this response entails are provided as you go through these directions.) This program, thanks to the provided setup/makefile, will be compiled into an executable called `loader`. You are provided two example cartridges (in the folders `cartridge_pokemon/` and `cartridge_super_mario/`); the code for each – thanks again to the provided setup/makefile – will be compiled into the executables `cartridge_pokemon` and `cartridge_super_mario`, respectively. Keep in mind:

- `loader`, `cartridge_pokemon`, and `cartridge_super_mario` are all executables. Each has `_start` and `main`.
- `loader` should be loaded the same way you've loaded any other program into the simulator so far, i.e. with the Firmware button.
- A cartridge should be loaded with the Cartridge button. This should ideally be done while another program (i.e. `loader`, previously loaded with the Firmware button) is already running. Note that only one cartridge can be inserted at a time. Clicking the Cartridge button while a cartridge is already inserted removes the current cartridge.

You've perhaps noticed that whenever we've loaded a program into the simulator, it starts at address `0x00000000`. That is always the case for firmware. That begs the question: where is a cartridge loaded? You can get the answer (and find a general view of the memory layout of the simulator) in Professor Nitta's documentation on the simulator here. The short version is that it the instructions of the cartridge are loaded to address `0x20000000`[2].

Short version of what you need to do for this part: When a cartridge is inserted (and its instructions are loaded to address `0x20000000` onwards), your loader program should immediately jump to the entry point of the cartridge. Although the

[1] Here, by "compiler", we mean the program that does the step of converting source code to assembly code.

[2] The data (globals) of the cartridge get loaded to address `0x70100000`, but we do not care about that in this assignment.

cartridge's instructions will be loaded at `0x20000000` onwards, you should not assume that the entry point (i.e. the instruction meant to be executed first in the cartridge) is at that address.

As mentioned during lecture, the cartridge is functionally an I/O device. As stated in slide #18 of slide deck #5, you could detect cartridge behavior through either a wait-loop or an interrupt, but for this part, **I require you to use a wait-loop.**

You will have to interact with the appropriate memory-mapped register(s). Which register(s) would this be? Read the documentation on the simulator in order to find out, specifically this page.

You may assume that the cartridge always ends with an infinite loop. When testing your code, I will never eject the cartridge, only insert it.

`cartridge_super_mario` is meant to be the first example. When this cartridge is inserted and the loader goes to it, the message, "Super mario" should be printed.

`cartridge_pokemon` is meant to be the second example. When this cartridge is inserted and the loader goes to it, the message, "Pokemon!" should be printed. Notice what I do in the `crt0.s` file of this cartridge in order to cause the first instruction of `_start` to not be at address `0x20000000`.

Miscellaneous advice / possibly helpful tips:

- In debug mode, you can see the values of the memory-mapped registers if you click the CS button ("CS" for "chipset") in the memory window at the bottom-right. Note that it will not update while the simulator is running; you will need to pause execution in order to see updates.
- Update: If you use the `.end` directive, only do so at the end of your file. In previous RV32EM programming assignments, a few students incorrectly used the `.end` directive. Keep in mind that `.end` effectively deletes / cuts off all instructions that come after it, so putting it anywhere other than at the end of your file is typically a mistake. (I don't think there's any real benefit to using the `.end` directive, so you could just not use it.)

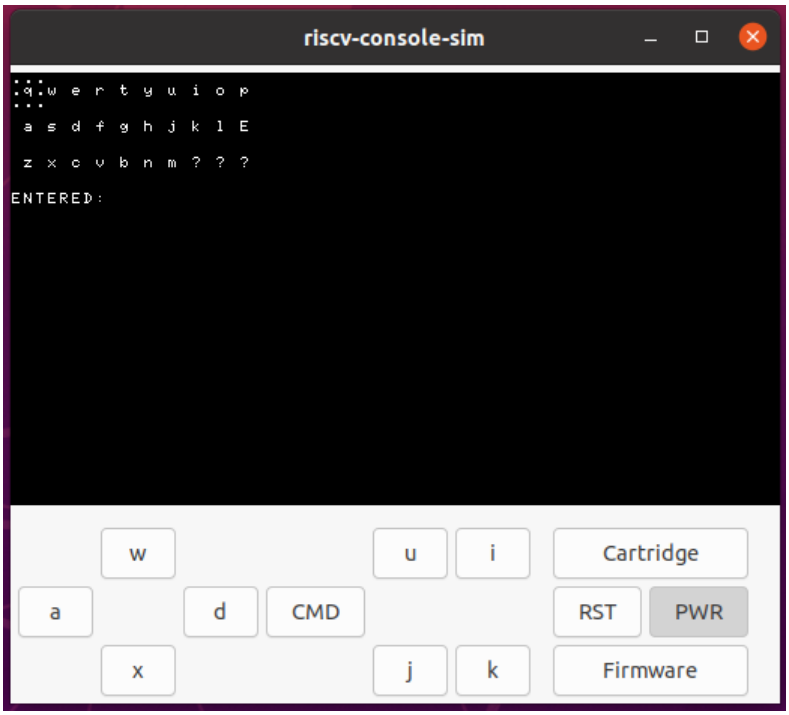
3.4 Part #2: Get Input

The only file that you should modify for this part (and the only file that you will submit for this part) is `get_input.c`.

For this part, you will implement a C function called `getInput()`. The function returns no value and takes as its only argument the starting address of an array of characters (or string)[3]. You are not allowed to modify the signature of `getInput()`. The function should present the user with a "keyboard" in the video screen and allow the user to "press" keys to enter characters. The characters entered by the user should go into the array of characters referenced by the function's argument. In short, you are implementing a function that allows the user to enter a string.

Below is how the keyboard should look. When your `getInput()` function is called, it should display this keyboard. Notice that there is a square (made of periods) around the 'q' key in the top-left; this is the *cursor*. The user can "press" a key by pressing the CMD button; this will press the key that the cursor is on. In the case of the below, if the user were to press the CMD button, then the 'q' key would be pressed.

[3]Throughout these directions, we will not worry about null-termination, i.e. ending a string with the null byte/terminator. The null byte convention would matter if we used common library functions like `printf()`, but since we lack access to such library functions in the simulator, we might as well not bother with the null byte convention.

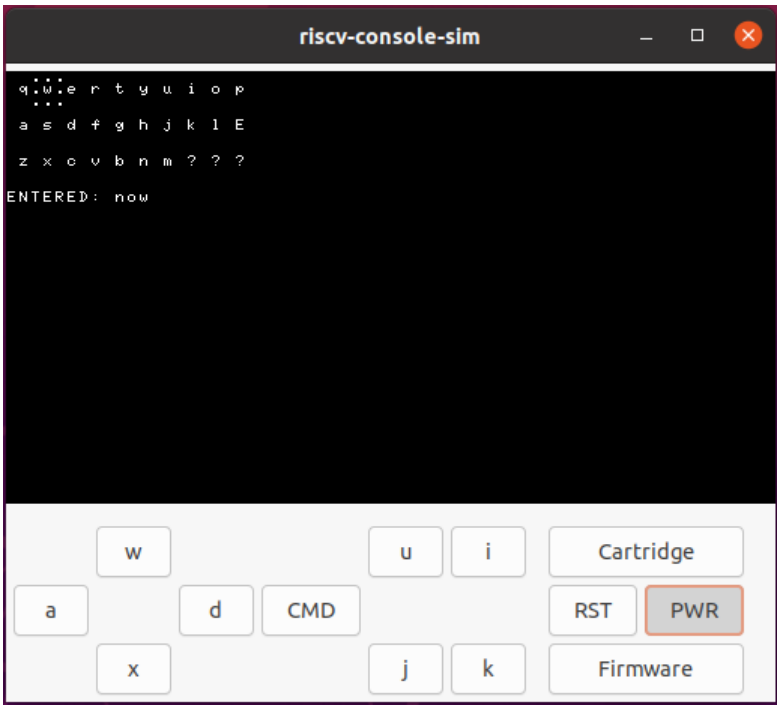


The first row of keys should be on the second row of the screen. The second row of keys should be on the fifth row. The third row of keys should be on the eighth row. The "ENTERED:" message (talked about below) should be on the eleventh row. The below image, which shows the row and column indices along the margins, should give you a good idea as to how things ought to be positioned.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	.	.	.																					
1	.	q	.	w		e		r		t		y		u		i		o		p				
2	.	.	.																					
3																								
4		a		s		d		f		g		h		j		k		l		E				
5																								
6																								
7		z		x		c		v		b		n		m		?		?		?				
8																								
9																								
10	E	N	T	E	R	E	D	:																
11																								
12																								

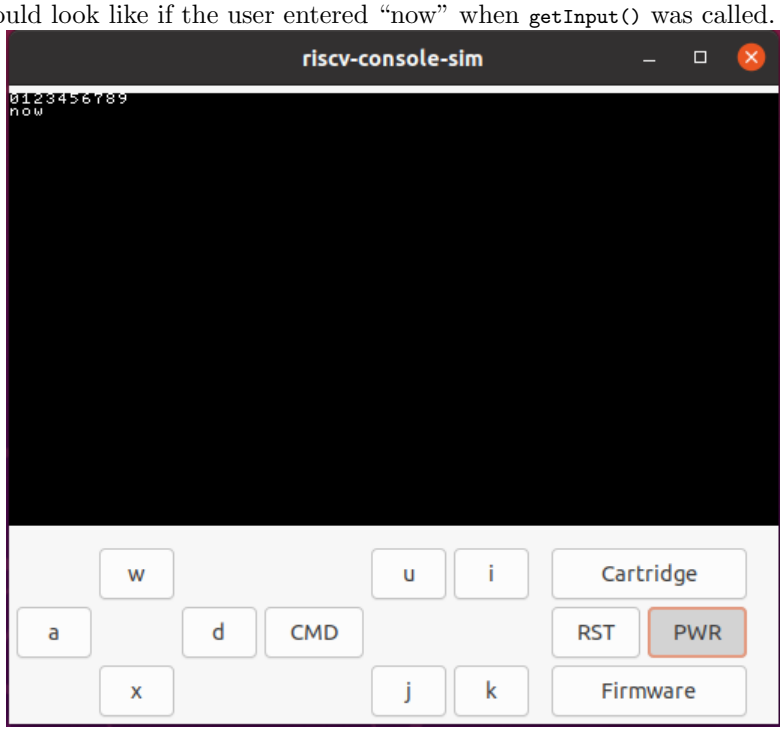
To make things easier, the three rows of keys all have the same length. The third row ends with three question mark keys. Pressing any of these should indeed result in entering a '?'. The second row of keys ends with an 'E' key. Pressing this key does not enter a character. Instead, pressing this key is analogous to pressing Enter on a real keyboard and is used to indicate that the user is done typing his/her input. That is, after the user presses the 'E' key, your `getInput()` function – after doing whatever necessary final actions – should end.

There is no Backspace key. Once the user enters a character, there is no taking it back. Each character that the user has entered so far should be displayed after the "ENTERED:" message. As an example, here is how things should look after the user presses the 'n', 'o', and 'w' keys.



Cursor movement is done with the 'w', 'a', 'd', and 'x' buttons and is intuitive, i.e. the 'w' button moves the cursor up, the 'a' button moves the cursor left, etc. The cursor cannot go out-of-bounds; any attempt to move the cursor out-of-bounds should result in the cursor staying still. The cursor must always be made of a square of periods around the current key. The cursor should not flicker. (Flickering can happen if you keep erasing and re-drawing the cursor every single loop iteration, even when unnecessary.)

In `main.c`, there is an implementation of `main()`. In it, your `getInput()` function[4] is called. After that, the screen is erased (in `clearScreen()`), and the result obtained from `getInput()` is displayed to the screen (in `printInput()`). Below is what this part should look like if the user entered "now" when `getInput()` was called.



Since you must detect when the CMD button is pressed, you must deal with interrupts. The interrupt setup is mostly provided to you and is mostly the same as in the example we did involving the CMD button in slide deck #5. Steps #1-#3 of the interrupt setup is already done in `crt0.s`, which you are not allowed to modify, and step #4 is already done for you in the provided skeleton version of `getInput()` in `get_input.c`. The provided version is set up such that CMD button interrupts

[4]You may have noticed that `getInput()` is marked `extern` at the top. Marking this declaration `extern` is a way of saying, "This function is defined somewhere else." I suppose we could have used a header file instead.

are only enabled for the duration of `getInput()`. The interrupt handler is `_interrupt_handler()` (which is defined in the provided file `interrupt.s` and already loaded into `mtvec` in `crt0.s`). This handler calls `c_interrupt_handler()`. (We talked about this split during lecture when talking about step #1 of the required setup.) `c_interrupt_handler()` is defined in `get_input.c`, since you will need to modify it to respond to the pressing of the CMD button.

You will have to put in some code to make sure that when 'w', 'a', 'd', or 'x' is pressed (meaning held down and then released), the cursor only moves once. If all you do is check if a button is held down, then whenever a button is pressed, your code will probably act as if the button was pressed many times, and the cursor will move a lot of spots. To fix this behavior, I would suggest you have a variable to keep track of if any button was pressed during the previous iteration of your loop.

Some assumptions your `getInput()` function can make:

- The user will only ever press the 'w', 'a', 'd', 'x', or CMD buttons.
- The user will never hold down two buttons simultaneously. (See the note about button jamming below.)
- The user will never enter more than 10 characters.

Miscellaneous advice / possibly helpful tips:

- In the provided `get_input.c` file, I used macros for the memory-mapped registers instead of variables. While writing my own solution, there were a few situations in which I got compiler errors due to performing arithmetic with the variable versions. Depending on your approach, you could encounter such errors. I don't want to go into too much detail on the errors here; the important part is that interacting with the macros is in general no different from interacting with the variable versions.
- Do not be afraid to use global variables in `get_input.c`[5]. In fact, you will probably have to use at least one global variable, since there is no way that you can pass arguments to `c_interrupt_handler()`.
- There seems to be a bug in the simulator that causes a button to sometimes jam when clicked. This means that the button will be stuck in a clicked state. (You'll be able to tell based on its color.) You can un-jam the button by pressing the key that corresponds to the button. (Don't forget that you do not have to click the 'w', 'a', 'd', and 'x' buttons; you can press the 'w', 'a', 'd', or 'x' keys on your actual keyboard instead.)
- I highly encourage you to use helper functions. If you do implement any helper functions, make sure to do so above the implementation of `getInput()`, or else the compiler will complain that `getInput()` is using functions that are not defined.
- Any global variable or helper function that you add in `get_input.c` should be marked `static`. Applying `static` like this makes the global variable or helper function "invisible" outside of `get_input.c`. This can prevent a naming collision (which would lead to a linker error), e.g. if `main.c` happened to have a global variable or helper function with the same name as one in your `get_input.c` file. Note, by the way, that `static` in this context has nothing to do with `static` in the context of C++ classes; I am not sure why the C/C++ creators decided to use the `static` keyword for different purposes.
- When running the simulator in debug mode, in the memory window at the bottom-right, you are shown in the rightmost portion the character representations of the bytes. You probably haven't needed to care about this yet, but it may help if you are trying to verify that certain characters end up in memory.
- Update: Since the simulator is a bare metal embedded system with no operating system to provide common services (such as for printing to a terminal), there is no `printf()` function or anything similar. There is also no debugger you can use at the C code level. You can still use the debug mode of your simulator to view through and step through the assembly code, but this means you will need to be aware of which assembly code corresponds to which C code. You can use the `riscv32-unknown-elf-objdump` program (with the `-D` flag) to get some idea of how the global variables are laid out too. This program is in the same directory as the RISC-V compiler in my CSIF files, so as long as you have done the below, you should be able to use the command. I will talk more about how to use it during the 11/19 lecture. It is easier to find where global variables are in assembly code than local variables, so that is perhaps another reason you may wish to make much use of global variables in `get_input.c`.

```
export PATH=$PATH:/home/aaron123/opt/riscv32/bin
```

4 Grading Breakdown

As stated in the updated syllabus, this assignment is worth 7% of your final grade. Below is the approximate worth of each individual part of this assignment.

- Part #1: 2.5%
- Part #2: 4.5%

[5] If you have been taught that global variables are a bad practice, you may find this brief defense of global variables here interesting; it was written by Professor Norm Matloff, who is in our CS department.

Even if you only get some of part #2 working, you should still submit it; you might get partial credit for what you do have working.

5 Submission Details

As with the previous programming assignment, there is no autograder for this assignment.

Submitting on Gradescope: There is a place on Gradescope that you can submit to. Only submit the files `loader.s` and `get_input.c`. You may be penalized for submitting other files.

Note that only your active submission (i.e. your latest submission, unless you manually activate a different submission) will count. I'll only see the file that you submitted in your active submission.

UCDAVIS
COMPUTER SCIENCE