Computer Science 230
Computer Architecture and Assembly Language
Spring 2021

Assignment 4

Due: Monday, April 19th, 11:55 pm by Brightspace submission
(Late submissions **not** accepted)

NOTE:

> **This assignment is OPTIONAL. Your best 3 of 4 assignments will be used to calculate your assignment grade.**

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.** 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

- Implement operations for 16x16 byte arrays (i.e., two dimensions, or 2D) using row-major order.
- Implement a specific operation on such a 2D byte array.
- Convert an array bitmap patterns (as explored in Lab 9 of this course) into a 2D array of bytes (where “bit set” corresponds to a 1 stored in a specific array location, and “bit unset” corresponds to a 0 stored in a specific array location).
- Render the contents of a 2D byte array (as described above) as on-off pixels on the Bitmap Display tool.
- Write a procedure that will implement the “Game of Life” by using work fulfilling the previous objectives for this assignment.

Provided with this assignment on Brightspace is an HTML file containing a link to a video demonstrating the behavior of all some (but not all!) parts of this assignment.

This assignment is designed in such a way that each part builds on those completed earlier. Please read all of the assignment before starting work. You do not necessarily need to understand all of the description before starting work. In fact, some of the later parts will make more sense to you as you complete earlier parts.

In order to ensure that your work on later parts does not break successful work on earlier parts, each of the parts for this assignment is submitted in a separate `.asm` *file.*

Part 1: 2D byte arrays (16x16)

`set_16x16`

parameters:
- `$a0` holds the address of first byte in the array
- `$a1` holds the row index
- `$a2` holds the column index
- `$a3` holds a word whose right-most byte is to be stored in the array at the row index, column index

return value: *none*

comment: If the provided row index or column index (or both) are out of range, there is no effect on the computer's memory.

`get_16x16`

parameters:
- `$a0` holds the address of first byte in the array
- `$a1` holds the row index
- `$a2` holds the column index

return value: `$v0` holds the value of the byte currently stored in the array at the row and column

comment: If the provided row index or column index (or both) are out of range, a value of 0 is returned in `$v0`.

`copy_16x16`

parameters:
- `$a0` holds the address of first byte in the *destination* array
- `$a1` holds the address of first byte in the *source* array

return value: *none*

File that must be used: `a4-part-1.asm`

Two-dimensional arrays are an abstraction provided to programmers. In reality all memory is one-dimensional – that is, a 32-bit computer's memory consists of an array of bytes starting from index/address `0x00000000` and going up to index/address `0xffffffff`. However, many kinds of problems are much easier to solve as a computer program if we can not only think in terms of two-dimensional

(2D) arrays, but also express our solution in a way that directly uses the row and column indexes of a 2D array.

For example, below is a representation of a 16-byte one-dimensional array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	14	15	9	2	6	**53**	5	89	79	3	23	**84**	62	643	38

The element with value 53 is indexed at 6; the element with value 84 is indexed at 12. Notice also that we use 0 to index the first element in this array.

However, we could also represent the given data as a four-by-four 2D array:

		columns			
		0	1	2	3
rows	0	3	14	15	9
	1	2	6	**53**	5
	2	89	79	3	23
	3	**84**	62	643	38

Notice now the element with value 53 is indexed as row 1, column 2; the element with value 84 is indexed as row 3, column 0.

The curious thing is that both figures represent exactly the same memory!

2D arrays are often implemented in what is known as *row-major order*. That is, the contents of each row are laid down, one after the other, in a 1D array. Calculating the location of a 2D element in a 1D array where row-major order is used is therefore straightforward.

Assume `rowlength` is 4:

- Element with value 53: row 1, column 2: 1 * `rowlength` + 2 = 6
- Element with value 84: row 3, column 0: 3 * `rowlength` + 0 = 12

You are to complete the three procedures for this Part 1 such that row and columns can be used to **read** (i.e., `get_16x16`) and **write** (i.e., `set_16x16`) values in a 16x16 2D array of bytes (i.e., where `rowlength` *is 16 bytes*). For completeness you will also write a procedure `copy_16x16` to make a copy of an 2D byte array (of size 16x16) in a different area in memory.

Part 2: A typical operation using 2D array

`sum_neighbours`

parameters:
`$a0` holds the address of first byte in a 16x16 array of bytes
`$a1` holds the row index
`$a2` holds the column index

return value: `$v0` holds the sum of array elements directly neighbouring the given row & column element.

File that must be used: `a4-part-2.asm`

In part 1, a motivation for 2D array operations was suggested. That is, sometimes it is easier to think in terms of 2D arrays and to write solutions using 2D operations.

Below is a representation of some possible 16x16 byte array. (The data was randomly generated, so there is significance to the values.)

columns

rows	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	9	3	8	5	7	0	8	3	9	3	4	7	3	5	7	**1**
1	4	3	2	1	7	8	5	3	7	5	3	6	6	3	1	6
2	7	4	3	1	9	5	4	6	6	3	6	1	6	6	0	7
3	3	8	1	5	0	5	5	0	4	9	2	0	6	2	4	1
4	2	6	5	9	7	2	7	8	4	2	8	0	1	1	0	9
5	5	8	7	1	9	9	7	2	2	3	8	7	2	1	2	4
6	5	6	1	0	8	8	5	7	0	3	4	5	1	4	2	4
7	7	3	6	1	8	5	**3**	1	4	2	0	0	6	9	7	9
8	0	3	5	4	7	3	8	9	8	5	5	0	2	4	5	5
9	6	6	0	3	8	1	3	2	1	2	5	1	5	0	7	3
10	5	8	8	3	2	7	8	8	5	4	4	4	3	6	3	7
11	4	0	3	0	9	5	7	7	0	4	8	3	0	7	9	0
12	0	6	7	4	9	2	7	0	0	4	9	1	1	9	7	5
13	8	1	2	7	6	1	4	0	3	5	3	8	1	3	3	2
14	2	9	3	7	2	0	3	8	8	3	1	9	8	0	5	8
15	2	9	7	2	1	1	0	**7**	9	9	9	9	1	4	6	2

Consider the element at row 7, column 6. This element has eight neighbours with values (starting at 12 o'clock and going clockwise) of 5, 7, 1, 9, 8, 3, 5, and 8, which all sum to 46.

Now consider the element at row `15`, column `7`. This element is at one "edge" of the array (so to speak), such that it has fewer neighbours. However, if we were to treat the "missing neighbours" as having values of `0`, then (starting at 12 o'clock and going clockwise) we would have the values `8`, `8`, `9`, `0`, `0`, `0`, `0`, and `3`, which all sum to `28`.

Finally consider the element at row `0`, column `15`. This element is at a "corner" of the array (so to speak). If we were again to treat the "missing neighbours" as having values of `0`, then (starting at 12 o'clock and going clockwise) we would have the values `0`, `0`, `0`, `0`, `6`, `1`, `7`, and `0`, which all sum to `14`.

You are to complete the single procedure `sum_neighbours` for this Part 2 such that the values of the neighbours around a given element in some 16x16 byte array are added together and this value returned. The element's location is given by its row and column. You are permitted to copy-and-paste work from your complete Part 1 as part of your solution to Part 2.

Part 3: Bitmaps, 2D byte arrays, and the Bitmap Display tool

`bitmap_to_16x16`

parameters:
`$a0` holds the address of first byte in the array
`$a1` holds the address of the first word holding a row's bitmap pattern

return value: *none*

`draw_16x16`

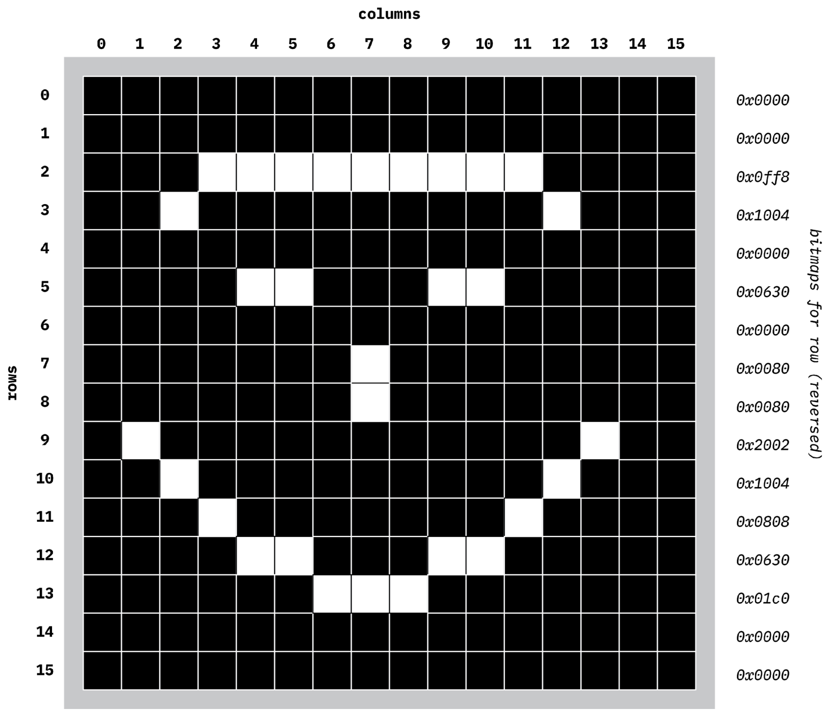
parameters:
`$a0` holds the address of first byte in the array

return value: *none*

File that must be used: `a4-part-3.asm`

In one of the later labs this semester, you learned how the individual bits in a 32-bit word can be used to represent whether or not a pixel is on or off in the Bitmap Display tool. (More precisely, the right-most 16 bits of a word is used to represent the pattern of on-off pixels for one row in the 16x16 bitmap display.)

Consider the following diagram:



Each element in a row is **dark if the pixel is unset**, and is **light if the pixel is set**. We may represent this figure by a 16x16 byte array. For example, the byte at row 5, column 10, will have the value 1; the next element to the right at row 5, column 11 will have the value 0.

Your task in this part is twofold. You are to write `bitmap_to_16x16` which will take 16 row-bitmap patterns (i.e., stored in 16 consecutive 32-bit words) and convert these into values of 0 and 1 for some given 2D 16x16 byte array. You will also write `draw_16x16` which will take a given 2D 16x16 byte array, and where an element (*row*, *column*) in the array has the value of 1, the corresponding pixel in the Bitmap Vector (at that *row*, *column*) will be set to white; if the value in the array is 0, the corresponding pixel will be set to black. You are permitted to copy-and-paste work from previously completed parts of this assignment into your solution to Part 3.

Part 4: The “Game of Life”

```
life_next_generation
```

parameters: *none*

return value: *none*

File that must be used: `a4-part-4.asm`

The mathematician John Conway became famous in 1970 when his “Game of Life” was published in an issue of the *Scientific American*. Your task in this part is to combine your work for the previous parts of this assignment into an implementation of this game. (Sadly, Dr. Conway died in April 2020 from complications of COVID-19.)

Strictly speaking this is not a game where the user interacts with the computer; “game” here is closer to the mathematical meaning of the term (as in “game theory”). Some starting value of on-off values in a 2D array (such as would result from calling `bitmap_to_16x16`) is used to compute the “next generation” of 16x16 values. Each successive generation is displayed. An element is “alive” if the value of the element is `1`; the element is “dead” if the value of the element is `0`.

In order to compute the next generation from the current generation, there are three rules:

1. If an element is alive in the current generation, and if the element has exactly two or three neighbours who are alive, then in the element is alive in the next generation (i.e., it goes from “alive” to “alive”).

2. If an element is alive in the current generation, and if the element has exactly one neighbour, or has four or more neighbours, then that element is dead in the next generation (i.e., it goes from “alive” to “dead”).

3. If an element is dead in the current generation, and if the element has exactly three neighbours who are alive, then in the next generation that element is alive (i.e., it goes from “dead” to “alive”). Otherwise the element remains dead (i.e., it goes from “dead” to “dead”).

If you are curious for more information about Conway’s game, then have a look at the Wikipedia article (which is very complete) at: `https://bit.ly/2ODGlpi`

In code provided for you in this part, the main code performs the following steps:

- The 16x16 byte array `GEN_A` is initialized to values of 0 or 1 based on an existing bitmap pattern (e.g., `PATTERN_GLIDER`).
- The `GEN_A` byte array is drawn on the Bitmap Display tool.

- The procedure `life_next_generation` is called, and it is assumed that when the procedure returns, the value of the next generation of `0` or `1` values is stored into `GEN_A`.
- The `GEN_A` byte array is drawn on the Bitmap Display tool. Then the code loops back to the previous step (i.e., infinite loop, moving from generation to generation).

You are to complete the code for `life_next_generation`. It is very important that this procedure does not itself call the `set_pixel` procedure. That is, the architecture of this solution depends upon separating the computation of the next generation from its rendering on the Bitmap Display tool. You are permitted to copy-and-paste work from previous completed parts of the assignment in your solution to Part 4.

(Thanks to David Clark for creating a "proof-of-concept" for this assignment idea.)

What you must submit

- Your completed work in the four assembly files: `a4-part-1.asm`, `a4-part-2.asm`, `a4-part-3.asm`, and `a4-part-4.asm`.

Evaluation

- 4 marks: Solution part 1.
- 5 marks: Solution part 2.
- 3 marks: Solution part 3.
- 5 marks: Solution part 4.

Therefore the total mark for this assignment is 17.

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.