2048 (10 + 1 Points)

This project is about implementing the well-known puzzle game 2048 [1, 2] in MIPS.

1 About the Game

2048 is a popular sliding tile puzzle game where tiles have to be moved around and combined on an $m \times n$ board, with $m, n \geq 2$. Each tile has an associated value, which is a positive natural number. The game starts with two tiles at random positions which either have the value 2 or 4. Every turn, the player selects a direction (left, right, up or down) in which all tiles of the board are shifted. All tiles are shifted as far as possible, leaving no gaps, as if the board was tilted and the tiles were sliding in the given direction. If two tiles with the same value collide in the process, they are merged together to form a single tile that has the combined value of the merged tiles. After every turn, a new tile with the value 2 or 4 is placed on the board at a random unoccupied position. The goal of the game is to merge tiles until a 2048-tile is created. The game is lost, if there is no direction left which would move or merge tiles. This is the case if the board is full and there are no horizontally or vertically adjacent tiles with matching values. In all other cases, tiles can be moved to the free spaces or adjacent tiles can be merged together, thus freeing up a space.

Figure 1: Graphical user interface

2 Preparation

Git

In case you did not participate in the pre-course, you have to configure Git. To do so, please execute the following commands:

- `git config --global user.name "<Givenname Lastname>"`
- `git config --global user.email "<id>@stud.uni-saarland.de"`

Remember to replace the given and last name as well as the id marked in `<..>` with your personal information and student ID.
Next, clone the project repository into a folder with the name `project1`:

- `git clone https://prog2scm.cdl.uni-saarland.de/git/project1/<username> project1`

Remember to replace `<username>` with your personal CMS user name. It should be noted that the project server is only available from the university's network.

MARS

In the *Settings* menu of MARS, activate the *Assemble all files in directory* and *Initialize Program Counter to global 'main' if defined* options. This is required to translate all additionally required files while building `src/main.asm` and to have the program's entry point set to the label `main`.

3 Implementation

The implementation is partially given. This includes reading of keyboard inputs and coordination of the game's flow. You will have to complete the game by implementing the missing subroutines. In the following sections, we describe how the playing field is represented in memory and how it is handed to the subroutines.

3.1 Tile Representation

The content of a single space in the playing field is represented by an *unsigned halfword*. If the halfword is zero, this means that the space is free, i.e. there is no tile on this space. If the halfword is non-zero, a tile with the numeric value of this halfword occupies the space. Note that tile values are greater than zero. For, example a 2048-tile is represented by the unsigned halfword $\langle 0000100000000000 \rangle$.

3.2 Playing Field Representation

The playing field is represented by an *array of unsigned halfwords*. The address of the first element of the array is called the *base address* b of the array. Each entry in this array corresponds to a single field of the board. To map the two-dimensional board to a one-dimensional array, the fields are stored in *row-major order*. This means that whole rows of the board are stored consecutively in the array, one after the other, from top to bottom.
Figure 2 depicts how the positions of a 4×4 board are mapped to the addresses visually. Here, each space is identified by its coordinate (x, y) in the grid. Note that the y-axis faces downwards in this depiction. For example, the field with the coordinates $(2, 2)$ is located at position 10 in the array, corresponding to the memory address $b + 2 \cdot 10 = b + 20$. The state of the game in Figure 1 is represented in memory as seen in Figure 3.

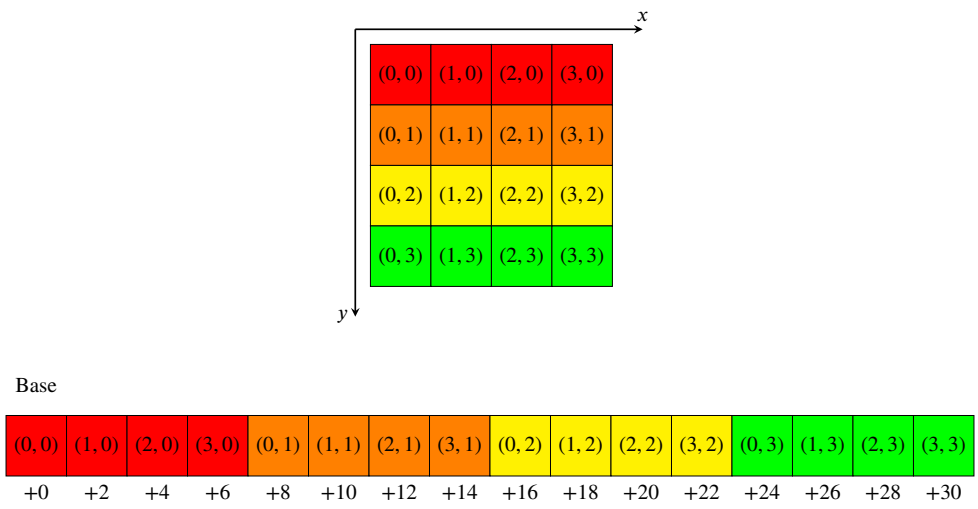


Figure 2: Representation of the playing field in memory

Base

+0	+2	+4	+6
0000000000000010	0000000000000100	0000000000000000	0000000000000000

+8	+10	+12	+14
0000000000000100	0000000000001000	0000000000000000	0000000000000000

+16	+18	+20	+22
0000000000010000	0000000000100000	0000000000000100	0000000000000010

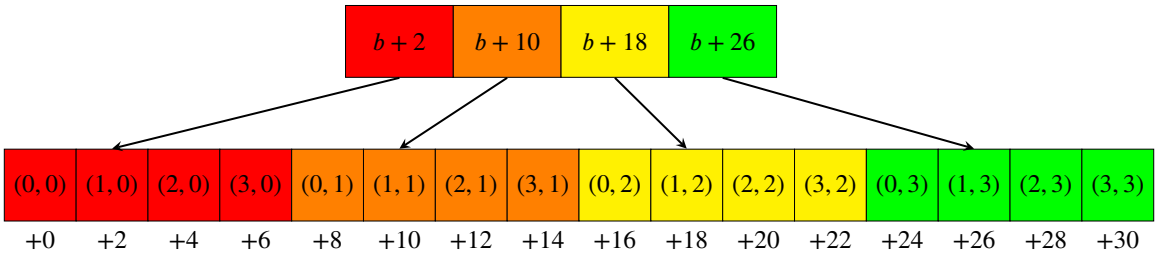
+24	+26	+28	+30
0000000000000010	0000100000000000	0000001000000000	0000000000001000

Figure 3: Visualization of the playing field's state as seen in figure 1 mapped to memory

3.3 Row and Column Representation

The subroutines you will implement that are concerned with the shifting of tiles operate on rows and columns of the board individually instead of the whole board. Conceptually, a row or column is merely a sequence of fields on the board. Each field can be identified by its address in memory, so a row or column is represented by an *array of memory addresses*. Therefore, when a row or column of the playing field must be given to a subroutine as a parameter, an array containing the addresses of the fields of the row/column is passed. The subroutines assume that the array enumerates the fields of the row/column from left to right/top to bottom.

Example For example, the second column of a 4×4 board corresponds to the following array of addresses:



3.4 Move Directions

The shifting and merging of tiles is dependent on the directional input of the player. For simplicity, we will assume that the input direction is fixed for the subroutines you will need to implement. Given that we have a correct implementation for a move to the left, we can simulate a move into the other directions by:

1. Rotating the board so that the intended direction becomes left
2. Shifting to the left
3. Undoing the rotation

This behaviour is already pre-implemented, so you need not worry about this. You only have to concentrate on the case where *left* is the direction in which tiles are shifted/merged.

4 Assignments

You only have to implement the subroutines that are described in the following. For each of these subroutines, you do not have to check if the parameters passed to the subroutine are valid. You may assume so throughout each of the exercises. Every subroutine is implemented in its own file. These files also contain additional comments regarding the passed parameters and the return value. All other files (`main.asm, buffer.asm, utils.asm`) must **not** be changed.

Remark: Unless otherwise specified, all subroutines must work on an arbitrary $m \times n$ playing field. It is therefore invalid to assume that the length of passed arrays is always a specific number. Tile values may not necessarily be a power of two in our tests, it is only guaranteed that the value of a tile is positive.

4.1 Checking for Victory (1 Point)

The game is won, as soon as a 2048-tile is created. For this assignment you have to implement a subroutine in the file `check_victory.asm` that checks whether any tile on the playing field has the value 2048.

Parameters

1. `$a0` – Base address of the playing field array
2. `$a1` – Number of elements in the array

The subroutine shall return 1, in case the playing field contains a tile with the value 2048, and otherwise 0. The board may not be changed by this subroutine.

4.2 Placing Tiles (1 Point)

After every turn, a new tile with either the value 2 or 4 is placed onto a randomly chosen unoccupied field. In this assignment you have to implement a subroutine in the file `place.asm` which places a tile with a given value at a given position, if that field is still empty. If the space is occupied by a tile, nothing shall happen. The position is given as an integer index of the playing field array.

Parameters

1. `$a0` – Base address of the playing field array
2. `$a1` – Number of elements in the array
3. `$a2` – Index of the field in the array
4. `$a3` – Value of the tile that shall be placed

If the tile was placed, the subroutine shall return 0, otherwise 1 shall be returned.

4.3 Checking for possible Moves (1 Point)

The game is lost if it is no longer possible to perform a move in any direction. For this assignment you have to implement a subroutine in the file `move_check.asm`, which checks whether moving *left* would have an effect on a given row. Moving left has an effect on the row if there is an adjacent empty space left of a tile, or there are two adjacent tiles with the same value. You do not have to consider other move directions.

Parameters

1. `$a0` – Base address of an array of addresses representing a row
2. `$a1` – Number of elements in the array

If a move is possible, the subroutine returns 1, otherwise 0. The board may not be changed by this subroutine.

4.4 Executing the Move (5 Points)

A turn is split into two components, the moving and the merging of tiles. In the following you will have to implement multiple subroutines that are used to perform the turn.

4.4.1 Shifting Tiles by one Space

In this assignment you have to implement a subroutine in the file `move_one.asm` that moves all tiles of the given row one field to the *left*, if possible. Tiles are shifted to the left simultaneously. The tile in the leftmost space of the row remains in place, if it exists. You do not have to consider other move directions.

Parameters

1. `$a0` – Base address of an array of addresses representing a row
2. `$a1` – Number of elements in the array

The subroutine returns 1, if at least one tile was moved, otherwise 0.

Example The following example shows the values of the tiles instead of their addresses for simpler visualization.

4.4.2 Shifting Tiles as far as possible

For this assignment you have to implement a subroutine in the file `move_left.asm`. It shall move all tiles in a given row as far as possible to the *left*. You do not have to consider other move directions.

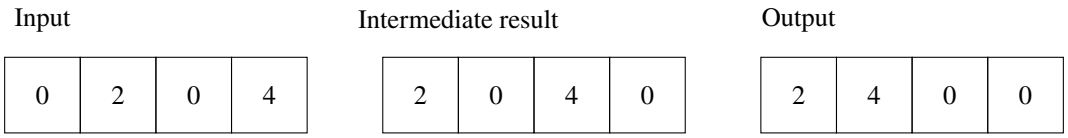
Remark: To do so, make use of the subroutine from Task 4.4.1.

Parameters

1. `$a0` – Base address of an array of addresses representing a row
2. `$a1` – Number of elements in the array

The subroutine has no return value.

Example The following example shows the values of the tiles instead of their addresses for simpler visualization.



4.4.3 Merging Tiles

The second component of a turn is merging the tiles. In this assignment you have to implement a subroutine in the file `merge.asm`. The subroutine shall merge two adjacent tiles with equal values in a row towards the *left*. Two tiles are merged by replacing the leftmost tile with a tile that has the sum of the two tiles as its value and removing the rightmost tile. Tiles must be merged sequentially from left to right. You do not have to consider other move directions.

Parameters

1. `$a0` – Base address of an array of addresses representing a row
2. `$a1` – Number of elements in the array

The subroutine has no return value.

Example The following example shows the values of the tiles instead of their addresses for simpler visualization.

Input

2	2	4	4

Output

4	0	8	0

4.4.4 Executing the full Turn

In this assignment you have to combine the two subroutines for moving (task 4.4.2) and merging (task 4.4.3) of the tiles so that all tiles in a given row are moved as far as possible to the *left* and merged. You do not have to consider other move directions.
To do so, move all tiles as far as possible to the left. Afterwards, merge the tiles once. Finish by moving all tiles as far as possible to the left again. Implement the subroutine in the file `complete_move.asm`.

Parameters

1. `$a0` – Base address of an array of addresses representing a row
2. `$a1` – Number of elements in the array

The subroutine has no return value.

Example The following example shows the values of the tiles instead of their addresses for simpler visualization.

Input

2	0	2	4

Moved

2	2	4	0

Merged

4	0	4	0

Moved

4	4	0	0

Note that the tiles may only be merged once per turn. This leads to the two tiles with value 4 not being merged in the shown example.

4.5 Visualizing the Playing Field (2 Points)

To make the game 2048 playable, a graphical representation of the playing field is still missing. In this assignment you have to implement a subroutine in the file `printboard.asm` that shows a playing 4×4 field with tiles that can contain up to four-digit numbers.

Parameters

1. `$a0` – Base address of the playing field array

The subroutine has no return value.

Example The following example visualizes the expected output for the board state depicted in Figure 1.

```
-----------------------------
|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|
|␣␣␣␣2␣␣|␣␣␣␣4␣␣|␣␣␣␣0␣␣|␣␣␣␣0␣␣|
|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|
-----------------------------
|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|
|␣␣␣␣4␣␣|␣␣␣␣8␣␣|␣␣␣␣0␣␣|␣␣␣␣0␣␣|
|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|
-----------------------------
|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|
|␣␣␣16␣␣|␣␣␣32␣␣|␣␣␣␣4␣␣|␣␣␣␣2␣␣|
|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|
-----------------------------
|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|
|␣␣␣␣2␣␣|␣2048␣␣|␣␣512␣␣|␣␣␣␣8␣␣|
|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|␣␣␣␣␣␣␣|
-----------------------------
```

The symbol ␣ represents a space character and is only used to visualize the number of used space characters. Do not add additional whitespaces or new lines. For every value there is space for 4 digits. In case the number has fewer digits, the empty space *in front of* the number is filled with additional space characters. The last line must end with a single newline (`'\n'`).

4.6 Bonus (1 Point)

To get the extra point, you have to extend the subroutine for executing a move such that it returns the score of a move as well: Every merge of two tiles is worth x points, where x is the value of the new tile. The basic score of a move is the total worth of all merges. On top of this, the basic score is multiplied by a combo factor of 2^{v-1}, where v is the total number of merges that happened during executuion of this one move, and the resulting final move score is accredited to the player. For example, when four tiles with the value 2 are merged to two tiles with the value 4, $2^1 \cdot (4 + 4) = 16$ points are awarded. If additionally two more tiles with value 2 are merged to a tile with value 4, i.e. in total six tiles with value 2 are merged to three tiles with value 4, the given points would be: $2^2 \cdot (4 + 4 + 4) = 48$.
Your task is to first extend the subroutine for merging by also returning the total number of merges that happened in register `$v0`, as well as the summed up worth of the merges in `$v1`. Afterwards, extend the subroutine used to execute a single move on a row analogously by also returning the total number of merges in register `$v0` and the summed up worth of the merges in `$v1`.

5 Evaluation

For every task there are *public*, *daily* as well as *eval* tests. The *public* tests are available in your local repository and can be executed at any time. The next section will detail how to run those tests. You have to pass all *public* tests for a task to get any points for that task. Note that a task can contain multiple subtasks, as seen with task 4.4. Every public test of every subtask must be passed.

The *daily* tests are not available locally. These are scheduled to run after you upload your project to our server (`git push`). After running the tests you will be notified via e-mail and can lookup the results in the CMS. Based on the limited server capacity, it might take a while before the tests are run. We only guarantee that you get the tests results once per day.
Finally, the *eval* tests will be used to rate the projects upon completion. These tests are unavailable to you.

6 Executing Tests and Debugging

Execute the command `./run_tests.py` in the root directory of the project to test your implementation. By default all public tests in the directory `tests/pub` are run. To run only specific tests, use the option `-t test_name_1 ...test_name_n` to run only the tests with names `test_name_1` through `test_name_n`. For example, execute `./run_tests.py -t test_check_win1` to run the public test `test_check_win1` only. To list all test names, use the `-l` option: `./run_tests.py -l`.
The output of the tests is colored to make the results more clear. This should display correctly in the majority of terminals, including the integrated terminal of Visual Studio Code. If the colored output poses a problem for the terminal you use, you can disable it with the option `-nc`.
If a test fails, you can debug it in MARS. The command `./build_testbox tests/pub/test_X.asm` copies the test as well as your implementation into the folder `testbox/`. From there you can start the test using MARS. Note, that only the files given in the task descriptions are included in the evaluation and the automated tests. Therefore do not commit additional files, especially not in the `testbox/` folder.

Self-written Tests

We recommend writing your own tests to supplement the given *public* tests. You can put your own test files into the folder `tests/student`.
A test comprises two files, `test_name.asm` contains the test file while `test_name.ref` contains the expected console output of the test. The printed output of the test is checked against the content of the reference file, *including newlines and spaces*. If the content matches, the test is passed. Please use the public tests as a reference.
The utility subroutines `assert_eq_board` and `print_board_test` provided in the file `test/test_utils.asm` can be used to write your own tests. The first subroutine checks if two boards are equal and can be used to assert that your subroutine does not change the board state. The second subroutine can be used to print the board state. Please conduct `tests/test_utils.asm` for further details.
With the option `-d dir_1 ... dir_n`, you can run all tests contained in the subdirectories `dir_1` through `dir_n` of `tests/`. For example, with `./run_tests.py -d pub student`, you can run all public tests and all of your own tests in sequence. The option can be combined with the `-t` option. Use `./run_test.py -d student -t test_X` to selectively run the self-written test `test_X`.

Remark: Remember that when consecutively printing two integers with a `syscall`, no space is produced between them automatically, which for example leads to the ambiguous output "11" if the value "1" is printed twice. Make sure you format the printed output appropriately when writing your tests.

7 GUI

Supplementary to the command line interface, for which you have to implement the `printboard` subroutine, we provide you with a graphical user interface (GUI). The GUI can be started using the command `./run_gui.py`. The game is then controlled using the arrow keys. The GUI communicates with your MIPs implementation, so the game is not functional at the beginning, as no subroutines have been implemented yet. Once you have correctly implemented all the required subroutines, the game will start working.

Remark: The GUI is provided only as an additional visual aid to measure your progress and to have fun with your implementation. It is not required that your submission is playable in the GUI.

8 Submission

The last commit on the master-branch before **10.05.2022, 23:59** is considered your submission.

References

[1] `http://gabrielecirulli.github.io/2048/`

[2] `https://en.wikipedia.org/wiki/2048_(video_game)`