G51SYS

ASSESSED EXERCISE #2B

Steven R. Bagley

Submission Deadline:
14/12/2017

This is the third and final 'weekly' G51SYS assessed exercise. In this exercise, we'll finish off creating the simplified 'Minesweeper' game we started in exercise two. In this set of exercises, we will continue to build some of the basic support functions we will use in the game before combining them to form a complete game. Specifically, you'll write a subroutine to clear the screen, write a subroutine to accept input from the user, convert your code to generate a board into a subroutine (which will need to use a stack), and finally put all the subroutines together and write the main game loop to allow people to play the game.

To allow the subroutines to be tested separately, there are skeleton files to implement most of the subroutines separately, and a final skeleton file, `minesweeper.s`, for the complete game. You will need to copy all your relevant subroutines into `minesweeper.s` for the final game to work.

As ever, you might want to start by thinking about how to implement these routines in C, or similar language before jumping straight into the assembler versions.

Assessment Notes

This exercise makes use of subroutines to implement various routines that will be part of our Minesweeper game and we'll put these together in the next exercise to produce the full games. For both this and the next exercise, half of the marks for each part of the exercise are allocated for your use of the subroutines.

Specifically, the test scripts will be checking that you are actually using subroutines, calling them correctly (using `BL`), and returning from the subroutines correctly (using `MOV PC, R14`). In addition, for these exercises, the scripts will be making sure your subroutines conform to the ARM Procedure Call Standard.

The other half of the marks will be allocated for the 'features' of the program — i.e. that the subroutines perform the correct task, prints things correctly, use the most appropriate instructions to complete the task (i.e. don't multiply numbers by doing repeated additions!)

Minesweeper Game Description

Our Minesweeper variant is played on a eight by eight grid, of which eight squares are filled with mines (represented by the letter 'M'). The eight mines are distributed randomly on the board, and then each square is covered up by a '`*`'. As the game is played, the user selects a square to be uncovered (by typing in the co-ordinates for the squares). If the uncovered squares contains a mine, then they lose. Otherwise, the square is revealed to show how many mines are in the squares surrounding it (as a number between '1' and '8') or a empty (represented as a space) if there are no

mines around it. The game is won if the player can turn over the 56 squares that do not contain mines, without accidentally uncovering a mine.

The following diagram shows the initial board display, and the grid reference system can be seen. The rows are identified by letters (case-insensitive) and the columns by numbers, allowing the user to enter a co-ordinate as 'b1' or 'A4'. The system should not accept incorrect co-ordinates, and should ask the user to re-enter valid coordinates.

```
      1   2   3   4   5   6   7   8

A     *   *   *   *   *   *   *   *

B     *   *   *   *   *   *   *   *

C     *   *   *   *   *   *   *   *

D     *   *   *   *   *   *   *   *

E     *   *   *   *   *   *   *   *

F     *   *   *   *   *   *   *   *

G     *   *   *   *   *   *   *   *

H     *   *   *   *   *   *   *   *

Enter square to reveal:
```

Revealed squares are shown thus (note the blank squares):

```
     1   2   3   4   5   6   7   8

A    1   M   1               1   M   M

B    1   1   1               1   3   M

C                                1   1

D    1   1   1

E    1   M   1   1   1   1

F    1   1   1   1   M   1

G    1   1   2   2   2   1

H    1   M   2   M   1


Enter square to reveal:
```

With mines being represented as an 'M':

```
     1    2    3    4    5    6    7    8

A    1    *    *    *    *    1    *    M

B    *    *    *    *    *    *    3    *

C    *    *    *    *              *    *    *

D    1    1    *    *    *    *    *    *

E    *    *    *    1    1    *    *    *

F    *    1    *    1    *    *    *    *

G    *    *    2    *    *    *    *    *

H    1    *    *    *    1         *    *

You lose...
```

There are three parts to this week's exercise, and so you'll find **four separate program files** in your forked git repository.

Subroutines

Your main routine and your subroutines all share the same set of registers. This means that if you modify a register in your subroutine it will change the value in your main routine as well (there's no scope of variables as in C). This means you will need to think carefully about which registers you use where. As a general rule of thumb, I'd suggest only using `R0-R3` in your subroutines, which means you can then assume that the values in `R4` onwards won't be affected.

Board Representation

We'll represent our board in memory as an array of 64 integers, stored linearly with one for each cell. The top left cell (i.e. the one with the co-ordinates A1) is cell zero, with the cells indexed sequentially across each row and down to the bottom. Therefore, cell 63 would be the bottom right square (i.e. H8), cell 58 would be the bottom-left (H1) and cell 3 would be the middle of the top line (A4). etc. Each entry in the array can contain one of the follow values:

Value	Menaing
`0`	The cell is empty, and no squares around it have any mines in it. Should be printed as a space.
`1`,`2`,`3`,`4`,`5`,`6`,`7`, or `8`	The cell is empty and *n* squares around it have mines in them. Should be printed as the number.
`66`	Cell contains a mine, and should be printed as a letter '`M`'.

Since each cell on the board is represented by an integer, it will take four bytes in memory. Therefore, if we want to advance from one cell to the next we need to add four to the address to get there.

We can also convert from a row and column index for a cell (e.g, row *r*, column *c*) and find the array index for that cell by using the formula: 8*c*+*r. Assuming that both* r *and* c *start counting from zero.*

In the same way, we can convert that array index into an offset from the beginning of the array by taking the index and multiplying it by four (which you can do quickly with a left shift).

Hint: accessing the values of integers in an array, is the same as accessing characters in a string, as seen in lecture 18. You just need to use `LDR` to read an integer, rather than `LDRB` that we used to read a single character (as a byte) in the lecture, and add four to get to the next value in the array.

Clear Screen routine

We'll need away to remove and redraw the board after each turn. Unfortunately Komodo doesn't provide a direct way to clear the 'Features' window but we can simulate the effect in a couple of different ways.

One way, is to just print out a series of blank lines — by printing out several newline characters (ASCII code 10), until the current output in the features window scrolls of the top. Printing 100 should do the trick.

Another way, which has a slightly more interesting effect is to 'delete' what is currently in the 'Features' window. This can be done by printing the 'backspace' character (ASCII code 8) and effectively has the same effect as if you were continually pressing the backspace key in the features window. If you send enough of these (one for every character printed) then it will clear the previous input from the features window. It also has the effect of clearing the screen in an aesthetically pleasing manner.

Pick one of the two methods above and implement the `clearScreen` subroutine to clear the 'Features' window. The skeleton file provided initially writes some output to the screen so you can see that it works.

Board Square Input

This `boardSquareInput` subroutine should work like a simpler version of the hexadecimal input routine you created a few weeks ago, and should first print a prompt to the user (specified in `R0` when calling the subroutine) and then accept the input in the form `C1` , `D8` or `H4` etc., followed by pressing ENTER. If the user doesn't enter a valid square number (e.g. `F9`, or `J1`) then the program should print the prompt the again until the user gives valid input.

On return, the subroutine should return in `R0` an integer between 0 and 63 that refers to the relevant array entry for the square — the top left cell (i.e. the one with the co-ordinates A1) is cell zero, with the cells numbered sequentially across each row and down to the bottom. Therefore, cell 63 would be the bottom right square (i.e. `H8`), cell 56 would be the bottom-left (`H1`) and cell 3 would be the top-right (A4). etc. You can use the formula above to convert the row and column specified into the array entry.

Generate Board subroutine

You will need to convert your routine for this to be a subroutine, `generateBoard`, but since this will need to call the `randu` subroutine — you'll need to ensure that the value of `R14` is preserved on a stack. We'll cover how to do this in lecture 24. Before then, you can experiment by simply using `STR` and `LDR` to store and load R14 to a memory location.

This subroutine should take the address of the array to fill in with the board in `R0`, and then execute as described in the previous exercise.

Note The assessment for this exercise will assume that R14 is saved on the stack so please make sure you modify it to use a stack before submission.

Minesweeper

Finally, using the subroutines we have created in this and the previous exercise (although it is probably worth checking they correctly preserve registers according to the ARM Procedure Call Standard, first), you are to write the main game loop that will implement minesweeper. For this your program, should:

- Generate a new board using your `generateBoard` subroutine, filling in memory defined to store the `board` array.
- Then in a loop until all the non-bomb squares have been revealed
    - Clear the screen (using the `cls` subroutine above)
    - Print out the board using your `printMaskedBoard` subroutine
    - Print out a message saying how many squares still need to be revealed.

    - Ask the user to select a square to reveal (using the `boardSquareInput` subroutine above)
    - If the square has already been revealed (i.e. if that index in `boardMask` is 0), print a message to this effect and ask the user for another location.
    - Now that the user has selected a valid square, we need to mark it as revealed by setting that index in the `boardMask` array to 0
    - Now check in the board itself to see what is located at that square.
        - If the board contains a mine, then you should print out a message telling the use they have stepped on a mine and died. The game should then quit.
        - Otherwise continue in the loop
- Once all the non-mine squares have been revelled (hint, count them as they are revealed , and remember that there are 56 non-mine squares, and 8 mines on the baord), print a message telling the user that they have successfully navigated the boat and exit the program with `SWI 2`

You'll find that the skeleton fit contains various strings predefined for you to use.

Good luck…