C SC 367 – Assignment 2

PART 2: PROGRAMMING IN ARM

In order to get your first experience of ARM programming you will be implementing a subset of the same code you did for assignment 1. This way you are familiar with the problem itself, its design, its flow of execution and you can use your own C code as pseudo-code. Make sure you check the answers posted for assignment 1 and you consider carefully whatever feedback you received from your own answer. Start from a correct basis!

You are not expected to implement the whole program from assignment 1. The main outline is shown in the ARM file given to you in the file "`A2TemplateImage.s`". The pseudo code for the overall flow is shown in Figure 1.

Figure 1: Revised Pseudo Code for the ARM program

```
Initialization:
Open the input file
        if problems, print message and exit program
Print a header message to screen
Obtain the input data:
Read row size (Rsize1) and column size (Csize1) of image from the input file
Read exactly (Rsize1)x(Csize1) integers as elements for an image,
        convert to characters, and store as 2D char array
Print image as characters with headings to screen
Processing the image:
Construct the Horizontal Mirror of the image into a second image
        and print the resulting image with headings to screen
Construct the Vertical Mirror of the image into a second image
        and print the resulting image with headings to screen
BONUS: Construct the Diagonal Right of the image into a second image
        and print the resulting image with headings (note that this
        is the equivalent of constructing the transpose of a matrix)
Closure:
Print a final message to screen
Close the input file
Exit the program
```

The most salient differences from the previous assignment.

1. All your ARM code is to reside in the "`main`" routine. You are not expected to code yourself any functions or procedures at this point, since their introduction in the lectures is too recent. It is all right to write one giant "`main`" at this point (but do not assume this is a good habit!). If you have studied ahead and feel confident, however, feel free to modularize your program. I suggest you come and check so that you implemented a correct interface.
2. The exception to the above is a procedure "`PrImage/PrImage2`", whose code, however, is given to you in the startup file "`A2_TemplateImage.s`". You are given careful guidance there on how to call

this procedure so that you can use it correctly. Make sure that you try and understand what is going on in this code given to you - it is examinable and questions based on it *may* appear in your upcoming midterm test. However we are helping you in that the code is provided and you do not have to figure it out on your own. The logic in it should be helpful when coding some of the other tasks.

3. The amount of work done by this program is a subset of the previous one. You will have to implement only a Horizontal Mirror transformation and a Vertical Mirror transformation. You can get bonus marks for implementing the Diagonal Right transformation (equivalent to computing the transpose of a matrix).
4. The amount of I/O processing is also smaller in that you need to implement file I/O only for input. You need to be able to open a file for input, read the sizes of a 2D image and then read the integers values describing the image. Most of the code for such a task is given to you in the examples in the lab, namely the file "`IO_Example2a.s`". The printing of images with appropriate headers is only to standard output, the screen. There will be no output to files and no need to check for end of files on input.
5. Please check again on the web pages for further instructions, should they be needed. Updates will be communicated in the lectures as well if questions arise, just like in a professional environment consultations with a manager regarding specifications and expectations take place regularly. Of course we try to have everything planned through in advance, but some details may still escape.
6. Take advantage of consulting with your instructor(s) and start working and planning early. This is a new language and a new paradigm for most of you!
7. The maximum sizes for the 2-dimensional images are 10 x 10.
8. Follow the outline given in the templates.

The supporting files and examples available to you

File Name or Example	What is it?	Where is it?
`A2Template2_ImageARM.c`	A C program which implements the solution expected for this assignment.	Linked from the Assignments web page
`A2TemplateImage.s`	The starting ARM file for your assignment. Rename it appropriately.	Linked from the Assignments web page
`A1inF2.txt`	Possible test file for current assignment 2, both for C and for ARM	Linked from the Assignments web page
`IO_Example_2a.s` with test file `LabIO2.txt`	File with appropriate I/O sample code as presented in the Lab	Linked from the Labs web page
`A1Final.c` with test files `A1inF.txt` and `A1inR.txt`	Full answer to assignment 1 in C	Linked from the Assignments web page
`CopyRow` example	Segment of ARM code to copy row i of a 2D array of integers to row j of another 2D array of integers	Lecture notes
`PrImage` and `PrImage2`	Segment of ARM code to print a 2D array which shows also the use of the addressing mode [Rn,Rm]	`A2TemplateImage.s`

Learning about data structures in depth: the 2-dimensional array

It is time to think about the allocation in memory of the 2-dimensional array you are using. It is helpful to view memory as a set of contiguous locations of words/bytes with increasing addresses. Figure 2 shows a snapshot of a section of memory where an array of characters has been allocated as "`char Afull[25]`". It has been initialized with the letters A through Y as shown. Note that `Afull[0]` has address `0000B100` and contains "`A`", while, for example, `Afull[13]` has address `B10D` and contains "`N`". In fact the element `Afull[13]` is exactly 13 bytes away in distance from the base address of `Afull` itself. As a side note, if

this had been an array of integers, the element `Afull[13]` would be exactly 13 x 4 bytes away in distance from the base address of `Afull` itself, since each elements contains 4 bytes.

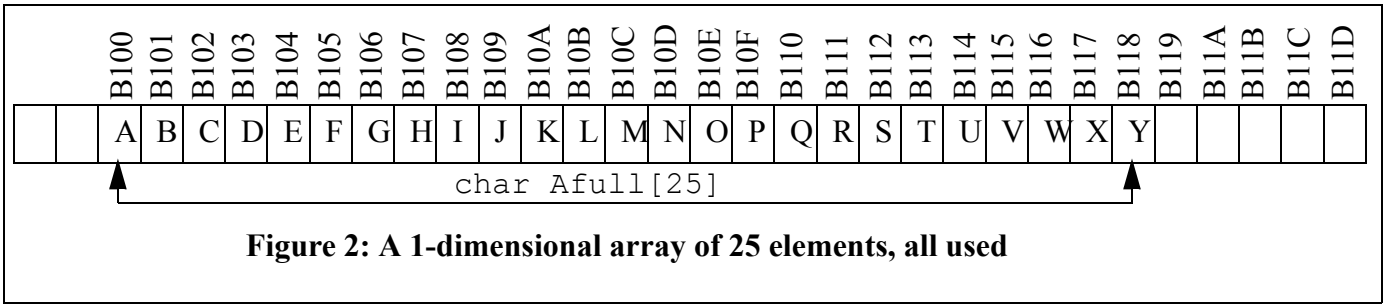

Figure 2: A 1-dimensional array of 25 elements, all used

Suppose now that one declares an array of the same size, as in "`char Bpart[25]`", as a possible maximum size ever to be filled, but only a portion is used depending on the application. One needs to keep track elsewhere of the actual size of the array used. Figure 3 shows a possible example Note the memory allocation of 25 bytes is the same, but only the locations at addresses `B100` to `B106` are currently used.. You should not have any problems understanding these two straighforward situations for a 1-dimensional data structure.

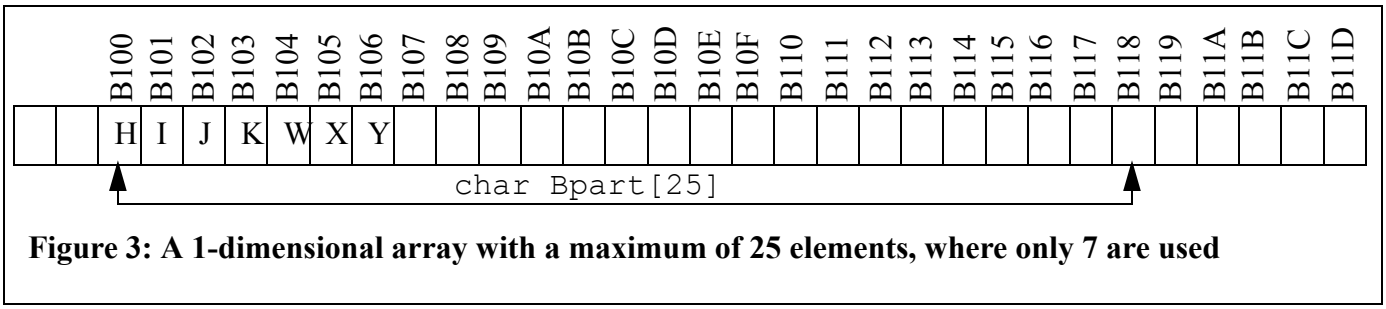

Figure 3: A 1-dimensional array with a maximum of 25 elements, where only 7 are used

What happens with 2-dimensional data structures? Consider Figure 4 where a 2-dimensional array is shown, of size 5 x 5, declared in C as `char C2full[5][5]`, and initialized with exactly 25 elements. The internal representation of memory allocation is not different from the way a 1-dimensional array of 25 elements looks in Figure 2, yet the C programmer's view of the data structure is represented by the matrix of characters shown at the bottom of Figure 4. The semantics of the C language interprets this view such that the element `C2full[2][3]` is known to be in row 2 and column 3 and its address is correctly calculated as an offset from the base address of the array. The example shows that the offset (that is, the distance from the beginning address of the array) for element `C2full[2][3]` containing the character "N" is of 13 bytes. If this were

an array of integers, the offset for element `C2full[2][3]` would be of 13 x 4, since each element occupies 4 bytes.

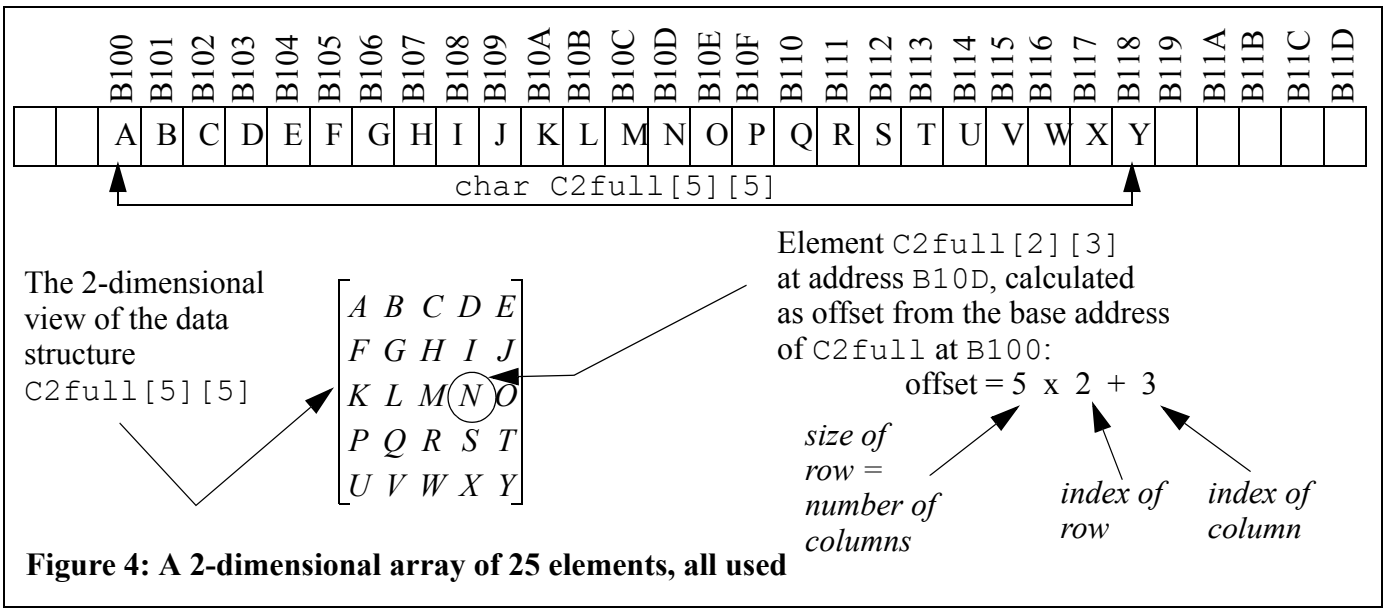

Figure 4: A 2-dimensional array of 25 elements, all used

Going one step further, what if one declares such a 2-dimensional array, where 5 x 5 is the possible maximum size ever to be filled, but not all elements are being used at a particular time, as is the case in your assignment? At this point there are two possible paths a programmer can take when dealing at this level. Starting with the view from a high level language, as in C, Figure 5 shows the interpretation. The `D2part[5][5]` is still allocated 25 consecutive locations in memory, which are the maximum size ever needed. However the program needs to use at some execution time only a 4 x 3 subset of the 2-dimensional array, as in the leftmost view in Figure 5. The high level language view together with the initial allocation are enforced in the storage discipline such that each 5 consecutive elements are always seen to compose one row and empty locations are left for smaller instances. Thus one can see the empty spaces in the locations where the corresponding elements are not used in the restricted application which needs only a 4 x 3. This is shown as a 2-dimensional matrix with empty elements in Figure 5, and by the empty locations in the linear memory view above. A sample element, "`H`", denoted in C by `D2part[2][1]`, resides at address `B10B`, accurately calculated as the distance from the base address of the array, keeping in mind that each row is defined to have 5 elements, even if not all used at this time. This is the way the translation happens from a high level language to a low level

execution. The corresponding assembly language/machine code generated by a compiler would implement it as such, leaving the empty spaces in each row and column.

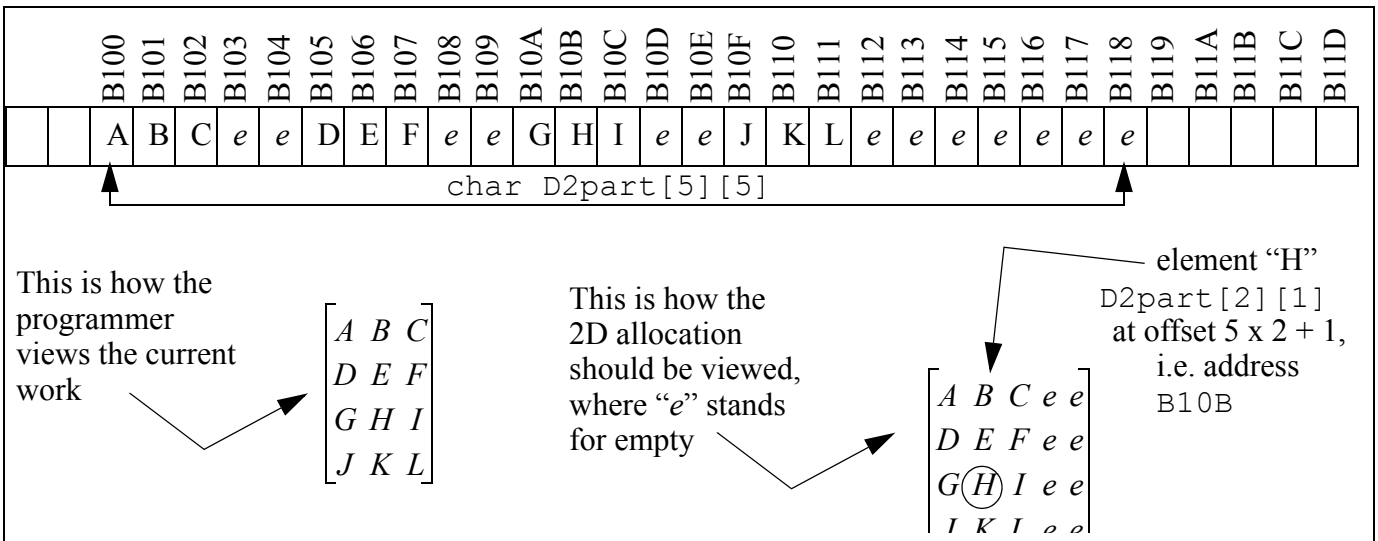

Figure 5: A 2-dimensional with a maximum of 25 elements, where only 4 x 3 = 12 are used

The alternate view starts from the assembly language level. When declaring in a data section a 2-dimensional array with a possible maximum size of 5 rows and 5 columns, one needs to allocate 25 locations at a maximum. In fact it is completely irrelevant at this level whether these 25 locations represent:

- a 1-dimensional array of 25 elements all used (as in Figure 2),
- or represent a 1-dimensional array of 25 elements of which only 7 are used (as in Figure 3),
- or represent a 2-dimensional array of 25 elements arranged in 5 rows and 5 columns all used (as in Figure 4),
- or represent a 2-dimensional array of a maximum of 25 elements "logically" arranged in 5 rows and 5 columns of which only 12 are used in the same "logical" view,
- or, finally, represent a 2-dimensional array of a maximum of 25 elements of which only 12 are used in a particular instance and they occupy the first 12 locations.

Figure 6 shows this final situation with the array called `E2part`. The programmer knows that the maximum allocation ever needed for this application would be for a 2-dimensional array of 25 elements, arranged in any shape regarding the number of rows and number of columns. These sizes are dynamically chosen at run time by reading input data. However no dynamic allocation of memory is being done here, given that the maximum sizes are statically predefined. Thus, as an example, the 4 x 3 matrix needed by this particular execution of the application can be stored directly as a "vector", that is a consecutive set of elements, and the row size and column size used for processing it correctly.

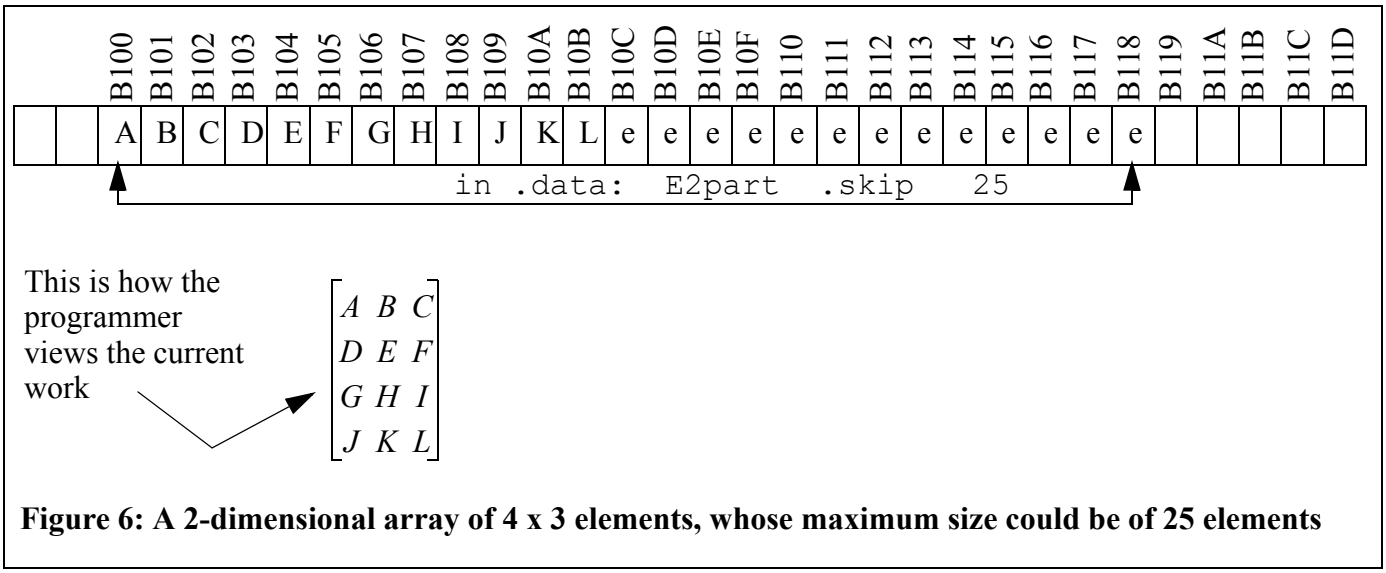

Figure 6: A 2-dimensional array of 4 x 3 elements, whose maximum size could be of 25 elements

If programming directly in assembly language level, one does not need to translate exactly the 2-dimensional semantic constraints of the high level language. The C language solution given to you should be used as pseudo code for the logic and it does not imply that you are expected to implement a direct translation as a compiler would do. You would need to be consistent with the high level language view only when writing with a mixture of the two languages.

Which data structure should you implement?

You have a choice to implement the 2-dimensional data structure following exactly the C language paradigm, the way a compiler would translate to assembly language code (see Figure 5), or to implement it more directly from the assembly language view as a linear array where you use the indices to enforce the 2-dimensional view (see Figure 6).

Does it make a difference? It impacts the decision on how you store each element when you read them in. Obviously it affects also the code you need to develop later for the image transformations and it affects the code to print the 2-dimensional image correctly. In the code given to you, you will find two versions of the routine for printing the image. The first version "`PrImage`" assumes the allocation as in Figure 6, where all elements have been stored consecutively. The second version "`PrImage2`" assumes the allocation as in Figure 5, where elements are stored as a subset of rows and columns of predefined fixed sizes, namely a maximum of 10 rows and 10 columns.

The important goal is that you understand both views and you understand what you are programming. A hint might be that the solution of Figure 6 is probably easier to handle at this point.

Where to start so that you can finish successfully

1. Using the files `A2TemplateImage.s` and `IO_Example_2a.s,` make the necessary customization with the goal to have a correct well tested program which can read an image and print it out, together with headings and your name. Choose your memory allocation strategy.
2. Design carefully, using diagrams on paper and pointers, the steps you will need to do the Horizontal Mirror image processing, the easier of the two. Start by using the segment of code to copy a row from a 2D array to another and test it. Then work your way to writing a loop which copies rows (0,1,...,Nrows-1) of Image to rows (Nrows-1,Nrows-2,...,1,0) respectively of Image2. Look at the solution in C for assignment 1 in `A1v2.c`.

3. The Vertical Mirror transformation is a little trickier and you should design it and work out all the pointers and position using figures and diagrams bfore you spend too much time coding and debugging. Make sure you understand how you can use the useful indexed addressing modes:
   - [Rn],
   - [Rn],#c ,
   - [Rn, Rm], etc. Come and ask questions if you do not understand.

Evaluation

Correct execution of your program will gain you at most 50% of the total marks. This implies as well that there are no assembler warnings! The other 50% of the evaluation is based on the quality of your code which will be examined for design, structure, documentation, clarity, organization, modularity. Good documentation is absolutely crucial at this language level!

What to hand in

1. Your working implementation in a file named "`A2csc230.s`" by electronic submission.

2. Both the source code and the output from execution must include your name and student number else it might be refused marking.