C SC 230 – Summer 2012 – Assignment 2 part 2

Due Wednesday June 27, by 16:00

Total Marks = 35 (20 for execution, 15 for design)+4 (bonus)

1. The problem to be solved and implemented with an ARM assembly language program

Given a 2-dimensional matrix of integers, print its diagonal, compute its transpose and check whether it is symmetric or skew-symmetric.

This is a subset of the C program in assignment 1. It should focus you to learn ARM programming, since you are completely familiar with the problem and indeed you can use the previous C code as your guideline pseudo-code when designing your ARM program. Also feel free to use the C code provided as a general answer to all and posted on the course web pages.

You are asked to do computation on 2-dimensional matrices.The input comes from a file containing integer data which represent the elements of matrices. Each matrix is printed and then you must print its diagonal, compute and print its transpose and check whether the matrix is symmetric or skew-symmetric or none of them. A set of test matrices are given to be read from an input file until end-of-file is reached.

2. Important note on implementation

At this point in your learning experience with ARM assembly language it is perfectly acceptable to write a program which consists only of a giant "`main`" routine, with the exception of the function already given to you (see below, they are `ReadInt, Rdmat` and `Prmat`). You will receive full marks for this type of implementation. This is because the material covering function calls with parameters may be encountered too late in the lecture to be of immediate value to you. However you are strongly encouraged to make use of a modular design with function and procedure calls. You will be rewarded with and extra 10% (4 marks actually). Make sure though that you implement this modularity correctly and well, following the rather strict guidelines and expectations outlined in the lectures. You will not get the bonus marks if your parameter passing or your side effects are not constrained appropriately - it is not enough to call a function and leave all sorts of global data floating around. feel welcome to come and ask for feedback and help on this item.

3. Definitions (subset repeated from assignment 1)

3.1. Transpose of a matrix

Given a 2-dimensional matrix *A* of integers, of size $m \times n$, the matrix A^T is called the ***transpose*** of *A*, where the rows of *A* are the columns of A^T. An example is given in Figure 1. More formally, the transpose of an $m \times n$ matrix *A* is the $n \times m$ matrix A^T where:

$$A_{ij}{}^T = A_{ji} \text{ for } (1 \le i \le n),(1 \le j \le m)$$

The matrix A (*3 x 4*):

$$M1_{3\times 4} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ -5 & -6 & 7 & 8 \\ 9 & 10 & -11 & 12 \end{bmatrix}$$

and its transpose A^T (*4 x 3*):

$$M1^T{}_{4\times 3} = \begin{bmatrix} 1 & -5 & 9 \\ 2 & -6 & 10 \\ 3 & 7 & -11 \\ 4 & 8 & 12 \end{bmatrix}$$

Figure 1: A matrix "M1" of size (3x4) and its transpose $M1^T$ of size (4x3)

3.2. Symmetric matrix

A *square* matrix *A* is defined to be ***symmetric*** if it is equal to its transpose, that is: $A = A^T$, as shown in the example of Figure 2.

$$M2_{3\times 3} = \begin{bmatrix} 3 & 2 & -1 \\ 2 & -5 & 17 \\ -1 & 17 & 4 \end{bmatrix} \qquad M2^T{}_{3\times 3} = \begin{bmatrix} 3 & 2 & -1 \\ 2 & -5 & 17 \\ -1 & 17 & 4 \end{bmatrix}$$

Figure 2: The square matrix M2 (*3 x 3*) is symmetric, since M2 = M2^T

3.3. Skew-symmetric matrix

A *square* matrix *A* is defined to be ***skew-symmetric*** if its transpose is equal to its negative matrix, that is: $-A = A^T$, as shown in the example of Figure 3.

$$M3_{3\times 3} = \begin{bmatrix} 0 & 6 & 4 \\ -6 & 0 & 5 \\ -4 & -5 & 0 \end{bmatrix} \qquad M3^T{}_{3\times 3} = \begin{bmatrix} 0 & -6 & -4 \\ 6 & 0 & -5 \\ 4 & 5 & 0 \end{bmatrix}$$

Figure 3: The square matrix M3 (*3 x 3*) is skew-symmetric, since M3 = -M3^T

3.4. Diagonal

The *diagonal* of a square matrix *A* is the line of entries from the top left corner (element at position [0,0]) to the bottom right corner (element at position [n-1,n-1]). It includes all the entries a_{ij} where $i=j$. In a non-square matrix this can be still seen as the line of elements a_{ij} where $i=j$, even if the diagonal, always starting from the top left corner (element at position [0,0]) does not finish at the opposite corner. Three examples are shown in Figure 4.

Figure 4: Diagonals of matrices

Square matrix $A_{3\times 3} = \begin{bmatrix} \textcircled{1} & 6 & 4 \\ -6 & \textcircled{2} & 5 \\ -4 & -5 & \textcircled{3} \end{bmatrix}$ Diagonal $= \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$

Non square matrix $B_{4\times 5} = \begin{bmatrix} \textcircled{1} & 1 & 1 & 1 & 1 \\ 2 & \textcircled{2} & 2 & 2 & 2 \\ 3 & 3 & \textcircled{3} & 3 & 3 \\ 4 & 4 & 4 & \textcircled{4} & 4 \end{bmatrix}$ Diagonal $= \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$

Non square matrix $C_{3\times 4} = \begin{bmatrix} \textcircled{1} & 6 & 4 & 5 \\ -6 & \textcircled{2} & 5 & 7 \\ -4 & -5 & \textcircled{3} & 9 \end{bmatrix}$ Diagonal $= \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$

4. The expected processing

The easiest way to formalize the flow of your application is to outline the overall functionality with pseudo-code or with a flowchart. The former is given here and you will have to draw a new flowchart yourself. More detailed specifications are given below.

Note that if the matrix is not square there is no need to check explicitly whether it is symmetric or skew-symmetric (look at the definition above). Many of you did not think about this in the last assignment and your program was unnecessarily inefficient.

```
Print initial messages including identification
Open the input file
    if problems, print message and exit program
Call "ReadInt" to read row size of matrix
While it is not end of file {
    Call "ReadInt" to read column size of matrix
    Call "Rdmat" to read integer elements of matrix and store them
    Print the matrix headings
    Call "PrMat" to print the matrix
    Print the diagonal of the matrix with a heading (calling subroutine Diag is optional)
    Compute the transpose of the matrix (calling subroutine Transpose is optional)
    Print the transpose headings
    Call "PrMat" to print the transpose
    If the matrix is square
        Test if symmetric (calling subroutine "Symm" is optional)
        Print message about symmetric
        Test if skew-symmetric (calling subroutine "SkewSymm" is optional)
        Print message about skew-symmetric
    Call "ReadInt" to read row size of the next matrix
}
Exit program: close the input file, print a closing message, exit.
```

Figure 5: Pseudo Code for the overall program

5. The code for the I/O processing

Most of the code for I/O processing is given to you in a template file posted on the web pages and called "`A2Symm-frame.s`". There an input file is opened, the data is read in, stored and printed. You are absolutely responsible to *understand every detail* of the code given to you and you are free to modify it if you think it appropriate. However before doing so, come and ask questions. Moreover you should learn from the code style what the expectations are for your own code (e.g. documentation and structure).

5.1. The input file (similar to assignment 1)

The data in the input text file is in the form of *K* groups of integers, where each group represents a matrix. In each group the first 2 integers represent the row size *m* and the column size *n*, while the following $m \times n$ integers represent the entries of the matrix in row-major order. The input file *__must__* be called: "`INA2.txt`".

5.2. Assumptions on the input data

- The maximum sizes of a matrix are m = 10 for the rows and n = 10 for the columns.
- Given the sizes of a matrix, it is guaranteed that exactly $m \times n$ integers follow, thus there is no need to do any validation on the data.
- All entries are guaranteed to be integers.

5.3. Sample input and output

A sample input file to be used for testing purposes will be posted on the web pages. The template for the output is similar to assignment 1. If you lost marks in assignment 1 in this regard, make sure to improve. Note the details of the expectations for the format, including name, student number, comments, etc.

6. The specifications: storage and I/O

6.1. Allocating the storage.

The declaration for the required storage of a matrix and its transpose must take into consideration that the maximum dimensions are *10* × *10*. Remember that you must allocate the correct number of bytes! You must allocate separate storage for a matrix and its transpose and label them `MatMain` and `MatTransp` respectively. The sizes of the matrices must be called: `RsizeM`, `CsizeM` for `MatMain`, and `RsizeTr`, `CsizeTr` for `MatTransp`.

6.2. All I/O processing.

The instructions for I/O processing are local to the simulator and are explained in the ARMSim# manual. You are expected to use them correctly and thus understand their interfaces. Additional sample source files, for example reading a list of integers from a file and printing them out both to the Output View screen (which in ARMSim# corresponds to "stdout") and to a file, are available to you. Help with these processing steps are also given in the lab sessions. The initial file given to you for this assignment includes most of the I/O processing and you can use some of the code there as your guide.

6.3. Reading the dimensions and the integer elements from a file.

The matrix is given as a set of integers and you are guaranteed that the correct number is present, that is, there are exactly `RsizeM` × `CsizeM` of them. Read `RsizeM` and `CsizeM`, and then each integer in a loop. End of file should be checked only when attempting to read the next `Rsize` for the next matrix. (This is done for you in the initial file).

6.4. Printing the matrix.

The matrix should be printed row by row as integers, thus you must print a newline character at the end of each row. This is already done for you in the `PrMat` procedure, but you should feel free to add custom touches.

6.5. Printing messages.

The local ARMSim# instructions for printing a string both to the Output View and to a file are explained in the ARMSim# manual. You are expected to format appropriate messages. Examples are shown in the code already provided to you plus in the lab examples and in the documentation.

6.6. Names for the input file, and for the program file

In the final submission:

- The input file *__must__* be called: "`INA2.txt`"
- The program file containing the source code *__must__* be called: "`A2csc230.s`"

7. The specifications: implementing the tasks

Each task should simply be implemented as part of the main program, as the proper calling of functions and procedures may not have been part of your learning experience by the time the assignment is due (see more details below). The tasks have been explained in assignment 1, except for the printing of the diagonal (which should be straightforward to understand, even if not necessarily so to implement).

8. The specifications: everything else

8.1. Printing characters and strings.

A *character* is a 1-byte entity. A *string* is defined as a sequence of characters. Make sure to note the difference between an "`.ascii`" and a "`.asciz`" declaration, especially as to how they relate to C.

8.2. Input and Output Examples.

Look at the web pages for the file posted for examples of input to aid you in testing your program, but do not hesitate to build up your own. There are also examples of output as produced by an implemen-

tation to give you an idea of the expectations. Pay attention to the expectations, since many of you lost marks in assignment 1 by submitting non-conformant output.

8.3. Documentation.

Do not forget about good documentation throughout, that is, meaningful and logical comments, not simply stating the sometimes obvious semantics of each line of code. This is particularly important at the assembly language level where an instruction may look incredibly simple and yet it badly needs to be commented in order to communicate the meaning of the action. For example the instruction `MOV R1,#1` obviously can be read directly as an assignment of `1` to register `R1` and in that it needs no further explanation. Yet its real meaning may be the initialization of a counter for a loop, something which would be obvious in a high level language where the same counter would be given a self documenting name (e.g. "`counter`"), but here everything is only a register number. It is extremely easy to get lost in your own code!

9. A systematic approach: labs will help as well.

1. `A2csc230V1.s:` Start with the code given to you for reading and printing matrices. Analyze it in depth, assemble it and make sure you understand what is going on.
2. `A2csc230V2.s:` Design and insert the code for printing the diagonal [5 marks].
3. `A2csc230V3.s:` Design and insert the code for the transpose and print the results [5 marks].
4. `A2csc230V4.s:` Design and insert the code for symmetry checking and print the results [5 marks].
5. `A2csc230V5.s:` Design and insert the code for skew-symmetry checking and print the results [5 marks].

Make sure that for every step above you make a separate version of your program file without overwriting your previous results. test each step. Do not proceed to the next task until you have fully finished the previous ones and you have come to ask questions if there are any doubts. It is then easy to backtrack to a tested working version when debugging.

10. What about the proper use of functions and procedures?

The program described above with all the processing done in "`main`" is not exactly an elegant solution. In fact in the initial template given to you there are three subroutines already: the function `ReadInt`, the procedure `RdMat` and the procedure `PrMat.` Each of them uses parameter passing through registers, saves and restores the state of computation (the list of registers) on the stack so that there are no side effects and follows the interface expectation of a regular C compiler (that is, which registers are used for input and output). All this is explained in the lectures, but at the point when this assignment is being designed and implemented by you it might be too early for the information to have been fully absorbed and ready to be used correctly. Thus you *are not required* to use any subroutines in your submission.

However, you may have a perfectly working program plus time and willingness to expand your experience. Do go ahead and use functions and procedures and you will receive and extra 4 marks (more than an extra 10% bonus), 1 marks for each extra function. My sample solution uses the following subroutines with these interfaces:

- `void Transpose(R0:addr matrix; R1:addr transpose; R2:row size; R3:col size)`
- `R0=CheckSymm(R0:addr matrix; R1:addr transpose; R2:total number of elements)`
- `R0=CheckSkewSymm(R0:addr matrix; R1:addr transpose; R2:total number of elements)`
- `void Diag(R0:addr matrix; R2:row size; R3:col size)`

Just make sure that you implement your functions correctly, where *all* needed information is passed *only* through the parameters, there is *no* usage of global data, *no* loading of data from memory, *all* locally used registers are saved and restored so that there are *no side effects* (that is, proper modularity and encapsulation).

11. To be handed in by Wednesday June 27 at 16:00.

1. Your working implementation in a file named "`A2csc230.s`" by electronic submission through the Assignment menu item on the course Connex web page.
2. A *printed* copy of your source code in the CSC 230 box on the second floor of the ECS building.
3. Your source code file must start with: name and student number, file name, assignment number, course and the purpose of the program overall.

12. Appendix: ARM Parameter Passing Conventions[1]

The Gnu C compiler gcc can translate a function into code which conforms to the ARM procedure call standard (or APCS for short), when given the appropriate command-line options. When cross compiling and linking C code and ARM code it is essential to follow the APCS precisely of course. This is not true when simply writing standalone ARM code where one may choose any register to be a parameter for input or output. However it is useful to build up good habits from the beginning, so you are encouraged to follow the APCS rules in your code (even if they are a little cumbersome at times).

The subset of the APCS rules directly relevant at the moment are as follows:

- The first four arguments are passed in R0, R1, R2 and R3 respectively. If there are fewer arguments then only the first few of these registers are used. Thus: parameter 1 always goes in R0, parameter 2 always goes in R1, parameter 3 always goes in R2, parameter 4 always goes in R3.
- Any additional arguments are pushed onto the stack (this has certainly not been covered yet so ignore it for now, even if you happen to have more than 4 parameters).
- The return value always goes in R0.
- The function must preserve the contents of locally used registers (excluding PC of course).

1.Only in case you decide to implement functions and procedures.