

## Manipulating Integer Matrices in *MIPS* Assembly

- **Electronic copy is due at 11:59 p.m., Friday, November 6, 2009.**
- Late projects will not be accepted.
- Please write your last name, first name, your ID number and the project number as a comment right at the beginning of your program.
- Write comments in your code.
- You are not required to hand in a hard copy of this project.

### Outline:

This is a *MIPS* program that builds on the skills you developed doing Project 1 by getting you to write longer and functionally more complex *MIPS* assembly code.

Your program will read in an integer matrix and implement matrix addition, scalar multiplication, transposition, and 'table' lookup. Specifically,

1. Your program should prompt for a matrix (call this *M1*) and read the matrix in from the console.
  - First prompt the user with the message: "*Please enter the number of rows in the matrix\n*".
  - When you have read in the number of rows, now prompt the user with: "*Please enter the number of columns\n*".
  - After reading the number of columns, prompt the user to start entering the elements of the matrix: "*Please enter the elements of the matrix\n*".
  - When you have finished reading the matrix in, print it out on the console, each row on a separate line, with the elements of a row separated by tab characters (*ASCII* '\t') or blank spaces.
2. Your program should then prompt for the operation that is desired. A single character will be used for each option
  - **'A' for addition.** For this option:
    - Prompt for another matrix ("*Please enter the elements of the matrix\n*") and read it in, as you did for *M1* (call this new matrix *M2*).

- When you have finished reading the matrix in, print it out on the console, each row on a separate line, with the elements of a row separated by tab characters (`'\t'`) or blank spaces.
  - Print out the result of the addition.
  - Your program should return to prompting the user for another operation after completing this command (*i.e.*, repeat Step 2.).
- **'S' for scalar multiplication.**
    - Prompt for the value to multiply by.
    - Print the result of the scalar multiplication.
    - Your program should return to prompting the user for another operation after completing this command (*i.e.*, repeat Step 2.).
  - **'T' for transposition.**
    - Print out the transposed matrix.
    - Your program should return to prompting the user for another operation after completing this command (*i.e.*, repeat Step 2.).
  - **'L' for table lookup.**
    - This option should prompt for a row and then prompt for a column.
    - It should then print out the value at that location in  $MI$ . The upper left corner element of  $MI$  is row one, column one.
    - If an invalid row or column index is entered (*i.e.*, negative value, or a value that is too big), print an error message and prompt for new input.
    - **This option should loop**, asking for row and column indices and printing the answer until a zero is entered as the row index. At that point your program should return to prompting the user for another operation (*i.e.*, repeat Step 2.).
  - **'E' for exiting the program.**
3. If a nonexistent option is chosen, print an error message and re-prompt for the option (*i.e.*, repeat Step 2.).

### Technical Specifications:

- Normally, a program this complex should implement each of its operations (reading in a matrix, printing out a matrix, 'A', 'S', 'T', 'L', *etc.*) as a separate function. Since you have not yet seen how to handle functions, you should write the program as a single piece of code in `main` without function calls.
- Despite this, try not to duplicate code. For example, you should have just one instance of the pieces of code that read in a matrix and then print it out, with a 'flag' value in some register to tell you whether you are reading the original matrix  $MI$ , or a matrix  $M2$

during the ‘A’ operation, or printing out the transposed  $M1$ , and so on. By testing this flag when you have finished printing you can jump/branch to the appropriate place in your code to continue.

- Try implementing your code one operation at a time. First write code that prompts for  $M1$ , reads it in and prints it out. When you have that working correctly, now start adding code that implements one of the simpler operations, ‘S’ say. When that is correctly working, start adding code for another operation, and so on. Do not try to do everything at once right from the start; it will be more difficult and confusing to debug.
- Your program may limit itself to matrices sized 8 by 8 and smaller.
  - Use the `.space` assembler directive to reserve enough space in `.data` for  $M1$ :

```
.align 2
M1: .space 256
```

This will reserve 256 bytes (sufficient for 64 integers of 4 bytes each) starting at label `M1` and aligned on a word boundary. You will need to reserve space for a second matrix, into which you would read  $M2$  and/or do the transposition of  $M1$  before printing it out, *etc.*

- The contents of a matrix will be signed integers. A matrix is entered one element at a time. Each element is typed into the console on a line by itself. Press the *Enter* key after the last digit of the element.
  - To read an integer from the console, you need to issue a `syscall` with value 5 in `$v0`:

```
li $v0, 5
syscall
```

The binary equivalent of the integer read in will be returned to you in `$v0`.

- To read a string (which can, of course, be just a single character) from the console, you need to issue a `syscall` with value 8 in `$v0` and the address in `$a0` of a label, `str` say, in `.data` where the string will be stored. `$a1` should contain the maximum length of the ‘buffer’ at label `str`. Assuming we used `.space` to allocate, say, 4 bytes at label `str`:

```
li $v0, 8
la $a0, str
li $a1, 4
syscall
```

Note that the newline character ‘\n’ will be part what is stored at label `str` if the user pressed *Enter* after typing in the character or characters of the string. It will also be null terminated.

### **Help:**

- The *PCSpim* version of *SPIM* you install on your PC from the textbook CD has a *Help* feature. There is also a *SPIM* tutorial on the CD. You can also, of course, use the [MARS](#) simulator instead.
- Refer to the *HELP WITH SPIM* section of course web site (<http://www.cs.sunysb.edu/~cse220/helpwithSpim.html>).
- There are several code examples in the `~cse220/examples/spim` directory on *sparky*. You can directly copy and run these on the *PCSpim* and *MARS* simulators you install on your PC.
- Section B.10 (*pp.* B-44 to B-80) of Appendix B of the textbook has a list of the *MIPS* instruction set. Two complementary versions of this are also available to you at the course web site:
  - [http://www.cs.sunysb.edu/~cse220/Spim\\_inst\\_set.html](http://www.cs.sunysb.edu/~cse220/Spim_inst_set.html) (linked to from the *HELP WITH SPIM* section of the course web site mentioned above).
  - <http://www.cs.sunysb.edu/~cse220/Handouts/Instr.html> (which is the handout to Lecture 11).

### **Electronic Submission:**

- From your account on *sparky*, submit an electronic copy of your program to the *sparky cse220* account.
  - You have to be logged in to your account on *sparky* which should already have a copy of the `*.asm` file you want to submit in the directory from which you will execute the submit command.
  - Execute: `~cse220/submit`
  - The submit command will ask you for a file name. Give the full file name, including the “`.asm`” extension.
  - Your project will then be copied under your name to a `handin` directory in the *cse220* class account.
  - If your submission is successful, you will get an acknowledgement right away.
  - If you are having problems, try typing `./<file-name>` (including the `.asm` file name extension) when asked for the file name.
  - If you submit more than once, each new submission overwrites the previous one, so only the last submission will be retained.

- You must submit electronically on *sparky*. Please do not email your program from *sparky* or anywhere else (*yahoo*, *gmail*, *etc.*) – not to me, not to the TA and certainly not to the *cse220* course account on *sparky*. (In fact, do not send email of *any* kind to the *cse220* account on *sparky* – we never check it!)
- If you are having problems with the submit command and decide not to continue with electronic submission of your project, then at least do not touch the *\*.asm* file you have on *sparky* after the due date and time. You can always show the timestamp of the file to the TA and the TA can then copy it to the *cse220* course account.