Arrays and Floating Point In MIPS

Multiplication (and Division, which is programming wise the same as integers), Procedure Calls and Recursive Procedure Calls.

# Floating Point

1. Using what you have learned in previous tutorials, you should be able to write a program that multiplies two **floating point numbers** together (See slides 36-40 in the notes on arithmetic)
   1. Literally this is just

l.s $f14, 2.5

l.s $f18, 3.5

mul.s $f16, $f14, $f18

* 1. Except… it isn’t. Unfortunately this is one place MARS differs from the book and SPIM

|  |
| --- |
| .data  a**:** .float 15.50  **.text**  main**:**  l.s $f0 a  mov.s $f12 $f0  li $v0 2  **syscall** |

Will let you load a floating point number and then print it off

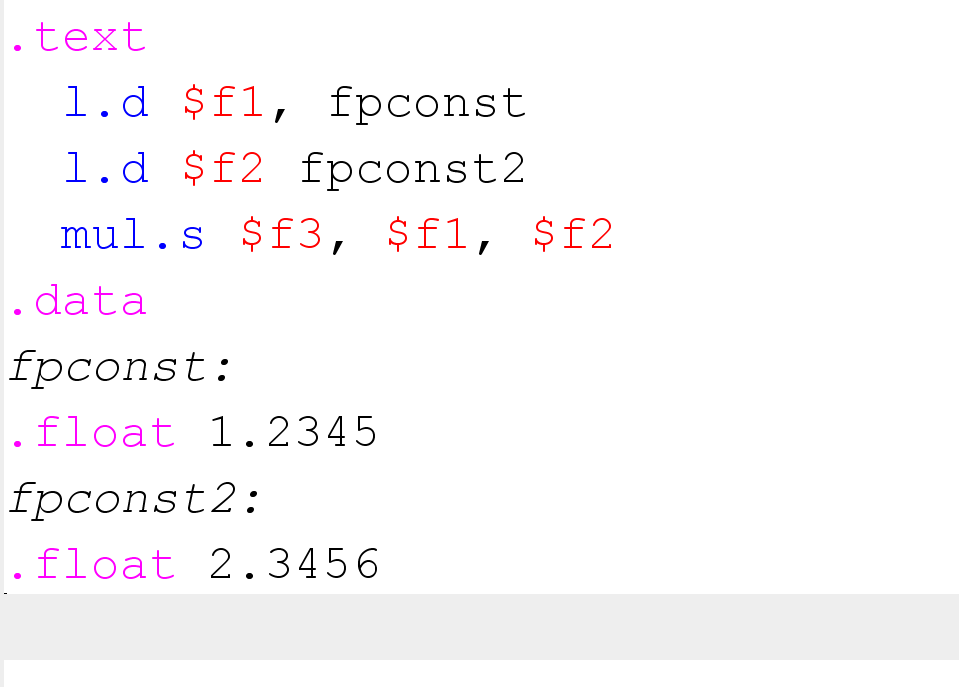
* 1. If you wanted to read in from the console something like

|  |
| --- |
| li $v0 6  **syscall**  mov.s $f2 $f0  mov.s $f12 $f0  li $v0 2  **syscall** |

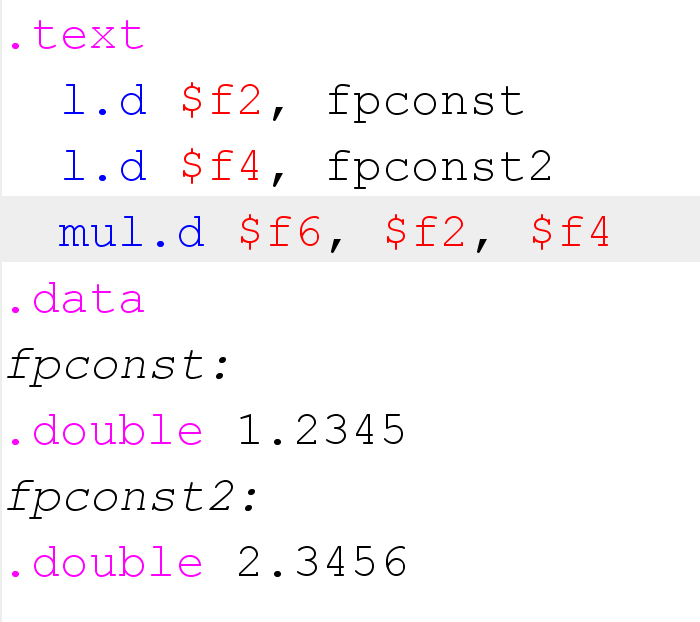
Will read in a value and print it off.   
  
Modify those to hard code one value (say A), read in another from the console, put them in $f1, and $f3, multiply them together and print the result out.

There are a couple of things you should play with. To see the state of floating point variables you inspect **COPROC 1** . Notice in (a) I have you separate your floating point values by 2 memory spaces. That isn’t always necessary (so if you did everything in $f1, $f2, $f3 it would work fine)

But the following doesn’t work



Because those are now load doubles (poke around with this to make it work, but there are a lot of things happening, you need to use mul.d, the variables are now doubles not floats, and you can use sequential odd numbered memory addresses)

is a working version with double precision

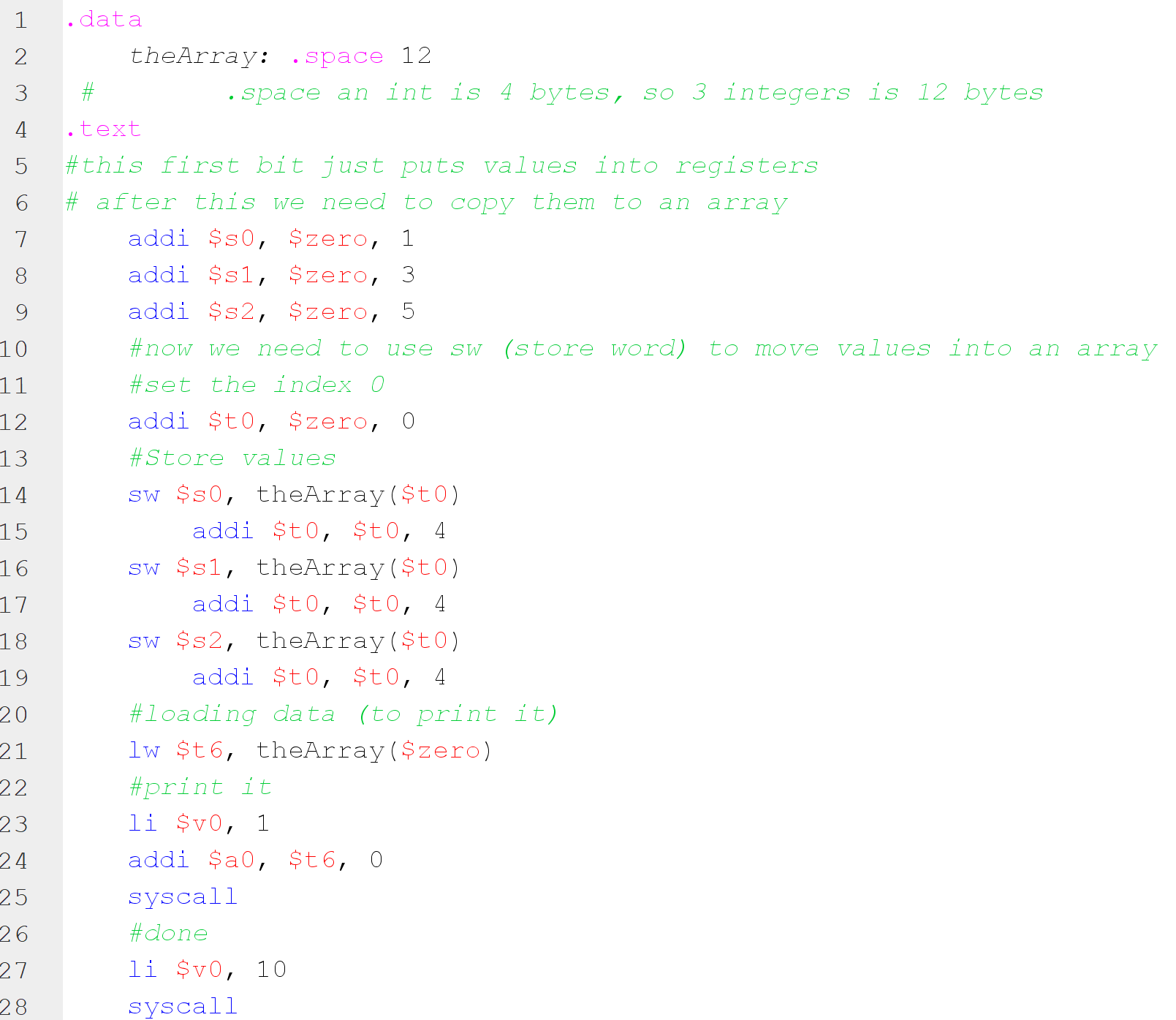
1. Convert your program to use integers, experiment with signed and unsigned (See slide 11 of lecture 4)
   1. The opcode for this is mul rs, rt with two operands (you obviously need to set rs and rt to something such as $v0 $v1)
   2. Once you execute the instruction you get a result that is now in two registers (hi and low), you can move them one by one into rd to test for overflow and underflow, again, see slide 11.

# Arrays: Goals for Today:

Make an array, do stuff with it. E.g. Arrays: int n[10]; // array of 10 integers … only in mips

# Arrays

Arrays in MIPS could be stored in registers, that severely constraints what the array could do, no more than 32 elements, to the point that using a struct might make more sense in that case. The better option for an array is to declare it in the ‘.data’ subsection and then it’s stored out in main memory. (Because MIPS is a simulator not a native language we are still constrained on array sizes compared to C, but that constraint is on the order of a billion elements).

Here is a simple program to create and print one element of an array (of integers)  
  
Same code in the box here, but with MS word formatting

|  |
| --- |
| .data  theArray: .space 12  # .space an int is 4 bytes, so 3 integers is 12 bytes  .text  #this first bit just puts values into registers  # after this we need to copy them to an array  addi $s0, $zero, 1  addi $s1, $zero, 3  addi $s2, $zero, 5  #now we need to use sw (store word) to move values into an array  #set the index 0  addi $t0, $zero, 0  #copy pasteable  #Store values  sw $s0, theArray($t0)  addi $t0, $t0, 4  sw $s1, theArray($t0)  addi $t0, $t0, 4  sw $s2, theArray($t0)  addi $t0, $t0, 4  #loading data (to print it)  lw $t6, theArray($zero)  #print it  li $v0, 1  addi $a0, $t6, 0  syscall  #done  li $v0, 10  syscall |

Array Continued:

Goal for today:

Change the array to 5 elements (you need to change the .space parameter to something > 12)

You then need to load up those elements into the array

And then print off all of the elements in the array.

This guy has some very good tutorials for the simple stuff we’re doing

<https://www.youtube.com/watch?v=BfHcogmKM20>

<https://www.youtube.com/watch?v=Vb8kuvxc4NE>

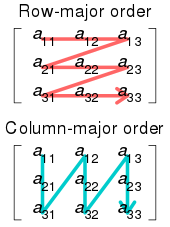
(I feel like me making tutorials that do the same thing would be redundant, and he’s got thousands of views per video)

2D Arrays in MIPS

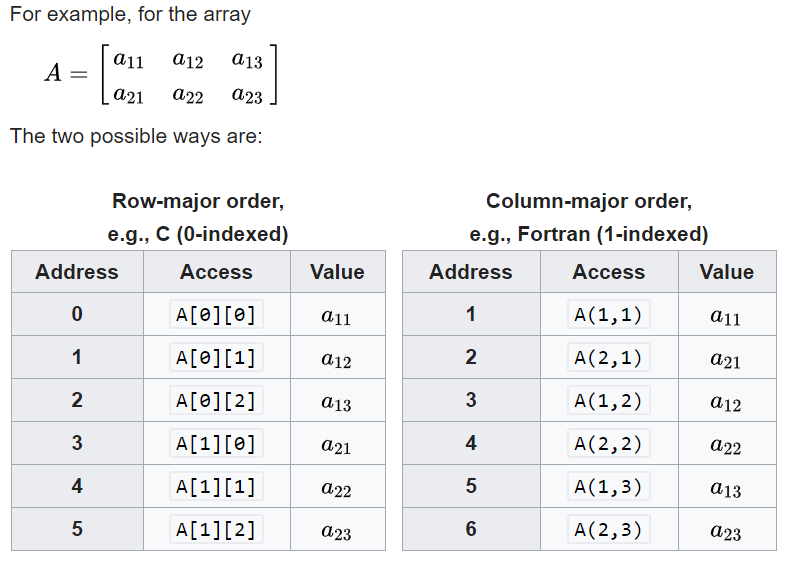
If you don’t get to this on the same day as the array lab, that’s probably ok. We should have time.

# Representation of Arrays

Real memory is physically linear in how it is accessed. This leads to two ways of ordering arrays in 2D. Row major and column major. They are more or less what they sound like.



From Wikipedia



Also from Wikipedia

So for any array element we need to be able to both put it in the correct place, and read it in, from it’s address

We will stick to Row Major though we could do this in column major

For today: **First up, you get to derive the equation that finds the correct array index.**

Second. To find the correct *memory address* from the equation you just derived, you need to multiply by the DATA\_SIZE (that’s how big each ‘word’ is, 1 for chars, 4 for ints/floats, 8 for doubles)

You’re going to make a simple program work to sum up the diagonals of a matrix.

You should implement matrix addition and multiplication (where you define 2, 4x4 matrices and add/multiply them)

Refresher if you need it:

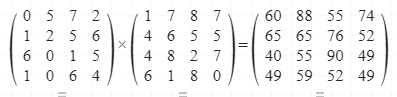
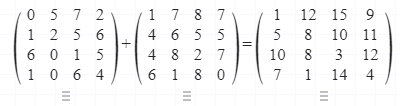
<https://www.mathsisfun.com/algebra/matrix-introduction.html>  
  
<https://www.mathsisfun.com/algebra/matrix-multiplying.html>

|  |
| --- |
| .data  # the next two lines define an array (mdArray as a 2x2 multidimensional array)  mdArray: .word 2,5  .word 3,7    size: .word 2 #dimension of the array (2x2 in this case, note this is only for square matrices)  .eqv DATA\_SIZE 4 # number of bytes per element, 4 for ints, 1 for chars, 8 for doubles  .text  main:  la $a0, mdArray # base address  lw $a1, size # size  jal sumDiagonal #sum of diagonals, in our starting example, this is 9.  move $a0, $v0 # this is because sumDiagonal will return it's last value in $v0  li $v0, 1  syscall  #and done  li $v0, 10  syscall  sumDiagonal:  li $v0,0 #sum =0  li $t0,0 #t0 as the index  sumLoop:  #The next 4 lines are the bit you need to derive and fill in yourself  **mul**  **add**  **mul**  **add**  lw $t2, ($t1) #getting element  add $v0, $v0, $t2 # sum = sum + mdArray[i][i]    addi $t0, $t0, 1 # i = i+1  blt $t0, $a1, sumLoop #if i < size, loop again  jr $ra #ends sum diagonal |

Note that the loop works on a variable called “size” which is toward the top.

**Modify your program to work on a 4x4 array and show that it works.**

**Show Matrix addition, and matrix multiplication on simple 4x4 matrices (define these in code), e.g.**



<https://matrixcalc.org/en/#%7B%7B0,5,7,2%7D,%7B1,2,5,6%7D,%7B6,0,1,5%7D,%7B1,0,6,4%7D%7D%2A%7B%7B1,7,8,7%7D,%7B4,6,5,5%7D,%7B4,8,2,7%7D,%7B6,1,8,0%7D%7D>

array base address + 4 \* (array height \* column index + row index)