We have seen how primitive data like integers, floating point numbers, and characters are stored in memory (as contiguous numerical values, accessed by location/address in memory). How do we represent more complicated data/data structures in memory at the machine level? Different high-level languages may use different representations. You will learn more about this in lecture, and will experiment with some of these representations in lab today.

One-dimensional arrays of numerical values

Exercise 1: Let’s begin with perhaps the simplest data structure, a one-dimensional array (similar to the one you worked with on the assignment). When all the elements of an array are the same size (such as an array of byte values), the elements of the array can be stored in their indexed order in memory, and accessed using:

\[
\text{base address of array} + (\text{size of element in bytes} \times \text{index})
\]

Here is the MIPS code to define an array, and a main program to invoke a procedure getElement, which takes as parameters the base address and the index of an array element, and returns the value of the specified array element:

```mips
.data
elements: .word 7  # number of elements in the array
          .word 1  # size of element in the array
          .byte 1,5,19,22,4,7,3
prompt:  .asciiz ‘Enter an array index: ‘
result:  .asciiz ‘The value of the array element is: ‘

.text
.globl main
main:
    li $v0,4  # prompt for an index
    la $a0,prompt
    syscall
    li $v0,5  # read in the index and store in $a0
    syscall
    move $a0,$v0
    la $a1,elements # put the base address of array in $a1
    jal getElement # read in the value
    move $t0,$v0 # move returned value to $t0
    li $v0,4  # output the result string
    la $a0,result
    syscall
    move $a0,$t0 # output the result
    li $v0,1
    syscall

endmain:  li $v0,10
syscall
```
1. What registers are used for the parameters to the procedure?
$a0 = index of array element, $a1 = base address of array

2. In what register is the element value returned?
$v0

3. Add the procedure `getElement` to the above code (you can copy and paste the starting code into MARS, but you may have to re-type the quotes used in the strings).

Your code should assume that the size of the elements in the array is 1 byte. Test by using various values in the range 0 – 6 for the index, and verify that you get the correct values.

Paste the code for your procedure here:
```
# procedure getElement takes as parameters the address of the array in $a0 abd the index of the
# element in $a0  
# and returns in $v0 the element of the array at the specified index
getElement:     lw $t2,4($a0)  # get the size of the array element
            addi $a0,$a0,8     # adjust $a0 so that it points past the length and size to elements)
            li $t1,1          # compare size of array with 1
            beq $t1,$t2,getByte # if the size is 1, get a byte value from the array
            sll $t1,$t1,1     # compare size of array with 2
            beq $t1,$t2,getHalf # if the size is 2, get a half word value from the array
getWord:        sll $t0,$a1,2    # size of array is 4, so multiply the index by 4
            add $t0,$a0,$t0   # add the index to the starting address of the array elements
            lw $v0,($t0)      # read the value at that address in memory, return value in $v0
            j done
getHalf:        sll $t0,$a1,1    # size of array element is 2, so multiply the index by 4
            add $t0,$a0,$t0   
            lh $v0,($t0)      # read the value at that address in memory, return value in $v0
            j done
getByte:        add $t0,$a0,$a1 # size of array element is 1, so simply add index to starting address
            lb $v0,($t0)      
done:           jr $ra
```

4. Add some new array declarations to your data segment, including an array of halfwords and one of words. Modify your main program so that it calls the procedure with the correct parameters to access your new arrays, and test to be sure the procedure works for different size elements. Demonstrate to the instructor

Paste the new array declarations here:
```
array2: .word 6
    .word 2
    .half 2,7,14,13,8,6
array4: .word 6
    .word 4
    .word 4,11,19,12,24,30
```
5. Write a procedure `printArray`, which will print out all the elements of an array, by using the `getElement` procedure. For the byte array `elements` defined above, it should print (include the square brackets and commas):

```
[1,5,19,22,4,7,3]
```

Paste the code for `printArray` here:

```assembly
# procedure printArray takes as a parameter in $a0 the address of the array to be printed, # and prints all the elements with punctuation, i.e. [1,2,3,4]
printArray:   addi $sp,$sp,-4       #save the $ra on the stack, since there is a nested procedure call
             move $s0,$a0
             sw $ra,0($sp)
             li $v0,11      # print starting bracket
             li $a0,','
             syscall
             lw $t3,0($s0)  # get the length of the array
             li $a1,0       # initialize the loop counter to 0
             printloop:   move $a0,$s0       # begin the loop by getting an element from the array
                           jal getElement
                           move $a0,$v0  # print the element
                           li $v0,1      
                           syscall
                           addi $a1,$a1,1 # increment the loop counter
                           beq $t3,$a1,printend # if the loop counter = array length, done printing elements
                           li $v0,11  # not done printing elements, print a separating comma and continue
                           syscall
                           move $a0,''
                           syscall
                           j printloop
             printend:   li $a0,']'        # print ending bracket
                           syscall
                           lw $ra,0($sp)  # restore the $ra from the stack
                           addi $sp,$sp,4
                           jr $ra
```

One-dimensional arrays of strings

Exercise 2: On the lab assignment for today, the strings in the array were all of equal length. This made it easy to access the elements of the array in the same way you did for the first exercise (base address + size * index). What if the strings are of variable length? There are a variety of techniques which might be used to represent the data in memory.

Assume you have declared the following array of strings in Java:

```java
String[] words = {'I','do','not','like','green','eggs','and','ham'}
```
Variable-length strings prefixed by bytes describing length, stored contiguously in memory:

```
stringarray1 .word 8    #length of the array
    .byte 1
    .ascii "I"
    .byte 2
    .ascii "do"
    .byte 3
    .ascii "not"
    .byte 4
    .ascii "like"
    .byte 5
    .ascii "green"
    .byte 4
    .ascii "eggs"
    .byte 3
    .ascii "and"
    .byte 2
    .ascii "ham"
```

Variable-length null-terminated strings stored (not necessarily contiguously) in memory, the array contains the addresses of the strings:

```
# using labels for each address for easier assignment
# note: strings are not defined in order
addresseggs: .asciiz "eggs"
addressI:   .asciiz "I"
addressham: .asciiz "ham"
addressnot: .asciiz "not"
addressgreen: .asciiz "green"
addressand:  .asciiz "and"
addressdo:  .asciiz "do"
addresslike: .asciiz "like"

# array of 8 addresses
stringarray2: .word 8
    .word addressI,addressdo,addressnot,addresslike,addressgreen,addresseggs,addressand,addressham
```

3. For the first definition above, implement a procedure `getAddressOf`, which takes as parameters the base address of the array and the index, and returns the address of the string at the given index. To test your procedure, examine the returned value using MARS, and verify that it is the correct address of the string element you are accessing.
4. Implement a new version of the procedure `getAddressOf`, assuming the array elements are addresses of strings. Use the address returned by the procedure to print the null-terminated string stored at the address which is returned from the procedure. Demonstrate to the instructor.

Notice the extra steps it takes to index elements of variable size! This strategy is not actually used as a construct in higher-level languages. It can be coded in a language like C, but it does not really fit C’s array support.

The second strategy (array of pointers to strings) is used in higher-level languages, and would be used to define memory for the Java declaration given above.

**Two-dimensional arrays**

Exercise 3: While Java allows two-dimensional arrays only as arrays of addresses of arrays (much like the arrays of addresses of strings from the last exercise), in C, nested array of arrays are used, where each row is stored contiguously in memory (row-major format), and the address of an element can be calculated by the following formula:

\[
\text{address of element}[x][y] = \text{base address of array} + (x \times \text{number of columns} \times \text{size of element}) + (y \times \text{size of element})
\]

-or-

\[
\text{base address of array} + (x \times \text{columns} + y) \times \text{size of element}
\]
Assume that the size of the elements in bytes will be 1, 2, or 4. This will not always be true in real-life data structures, but it makes the calculation more efficient here. Why?

Because you can shift to multiply by the size (instead of using the multiplication instruction)

1. Add the procedure *getElement* to the following code, which contains a declaration for a 4x4 array of integers, and some test code to allow the user to enter the indices to access an element of the array.

```assembly
.data

twodi: .word 4  # size in bytes of each element
    .word 4  # number of rows
    .word 4  # number of columns
    .word 1,3,5,7
    .word 2,4,6,8
    .word 9,11,13,15
    .word 10,12,14,16

prompt1: .asciiz "\nTo access an element A[x][y], enter x: "
prompt2: .asciiz " also enter y: "

.text
.globl main
main:
    li $v0,4  # prompt for an index
    la $a0,prompt1
    syscall

    li $v0,5  # read in the row index and store in $a1
    syscall
    move $a1,$v0

    li $v0,4  # prompt for an index
    la $a0,prompt2
    syscall

    li $v0,5  # read in the column index and store in $a1
    syscall
    move $a1,$v0

    la $a0,twodi  # put the base address of the array in $a0
    jal getElement
    move $a0,$v0  # move the value of the element from $v0 to $a0 for printing

    li $v0,1  # print the result
    syscall

    li $v0,10  # exit
    syscall
```
Paste your code for getElement here:

```assembly
# procedure getElement has parameters of base address of array in $a0, x coordinate of element in $a1, and y coordinate in $a2, and return value of specified array element in $v0

getElement: lw $t0,0($a0)  
               srl $t0,$t0,1  # $t0 = 0 for byte, 1 for half, 2 for word
               lw $t1,4($a0) # $t1 = number of rows
               lw $t2,8($a0) # $t2 = number of columns
               addi $s0,$a0,12 # $s0 points to the start of the array elements

               # calculate the address of the array element: base address + x * numcols * size + y * size
               mul $t3, $a1,$t2 # $t3 = x * #cols
               slv $t4,$t3,$t0 # $t4 = x * #cols * size
               sllv $t4,$a2,$t0 # $t4 = y * size
               add $s0,$s0,$t4 # $s0 = base address + x * numcols * size + y * size

               # get the value (byte, half, or word, depending on the size)
               li $t1,0
               beq $t0,$t1,getByte2d
               sll $t1,$t1,1
               beq $t0,$t1,getHalf2d
               getWord2d: lw $v0,0($s0)
                           j end2d
               getHalf2d: lh $v0,0($s0)
                          j end2d
               getByte2d: lb $v0,0($s0)
                           j end2d

end2d:  jr $ra # return the value of the array element in $v0
```

2. Write a procedure sumAll which uses a nested loop to iterate and sum all the elements. Demonstrate to the instructor.

Paste your code for sumAll here:

```assembly
# procedure sumAll has parameter = base address of array in $a0, returns sum of all elements in $v0

sumAll: addi $sp,$sp,-4 # allocate stack for $ra (nested procedure)
        sw $ra,0($sp)

        lw $t5,4($a0) # $t5 = number of rows
        lw $t6,8($a0) # $t6 = number of columns
        li $s4,0 # accumulate sum in $s4, initialize to 0
        li $t7,0 # $t7 is x for outer loop of nested loop

outerloop:  beq $t7,$t5,doneouter
            li $t8,0 # $t8 is y for inner loop of nested loop

innerloop:  beq $t8,$t6,doneinner
            move $a1,$t7
            move $a2,$t8
            jal getElement  # get the element A[x][y]
            add $s4,$s4,$v0 # add to the running sum
            addi $t8,$t8,1
            j innerloop

doneinner:  addi $t7,$t7,1
             j outerloop

doneouter:  lw $ra,0($sp) # restore the $ra and return with the sum in $v0
            add $sp,$s0,4
            move $v0,$s4
            jr $ra
```
3. Write a second version of sumAll, using a single loop (exploit contiguous row layout and calculate total array size to determine loop bound).

Paste your code for the second version of sumAll here:

```assembly
# procedure sumAll2 has a parameter = base address of array in $a0, returns sum of all elements in $v0
sumAll2: addi $sp,$sp,-4 # allocate stack for $ra (nested procedure)
    sw $ra,0($sp)

    lw $t5,4($a0) # $t5 = number of rows
    lw $t6,8($a0) # $t6 = number of columns

    mul $t5,$t5,$t6 # $t5 = x * y, total number of elements in array
    li $s4,0 # accumulate sum in $s4, initialize to 0

    li $a1,0 # row index will always be 0
    li $a2,0 # $a2 will act as column index/loop counter

singleloop: beq $a2,$t5,doneloop
    jal getElement # get the element A[x][y]
    add $s4,$s4,$v0 # add to the running sum
    add $a2,$a2,1 # increment index
    j singleloop

doneloop: lw $ra,0($sp) # restore the $ra and return with the sum in $v0
    add $sp,$s0,4
    move $v0,$s4
    jr $ra
```