

## Loops in SPIM

- We have already discussed loops, programs which can perform repetitive calculations rapidly for a large amount of data.
- Iterative loops are easy to structure in a higher-level language, in which simple sets of operations may be repeated innumerable times via “do,” “if,” “for,” and other compiler instructions.
- In assembly language, however, there are no high-level commands; loop details must be designed by the programmer.

## Loops (2)

- We saw a brief example of a simple loop in the last lecture, and also did a brief exercise. Today, we examine loops more closely.
- In assembly language, “the devil is in the details,” an old saying that means the programmer must do the detail work to create a loop.
- This includes writing the loop software, keeping track of the number of loop passes, and deciding via a test when the looping software has completed its job.
- Today’s examples will give a good overview of loop construction.

## Loops (3)

- **Requirements for a loop in assembly language:**
  - **Entry function to get into the loop.**
  - **Counter function (if a counter is used):**
    - **Counter software to keep track of passes through the loop.**
    - **Initialization function to set the counter to zero or preset it to some other appropriate number before starting the loop.**
    - **Increment (decrement) function for the counter.**
  - **Assembly language instruction set that performs the desired function within the loop.**
  - **Control statement (branch or set) that compares the number of passes in the counter (tests) or otherwise determines when to exit the loop at the correct time.**

## **Loops (4): Loop Process**

- **Initialization before loop starts:**
  - Set the counter to control the number of times the loop is executed, since we are using a counter.
  - Enter loop (jump instruction or similar).
- **Inside the loop:**
  - Execute loop instructions to complete each pass through loop.
  - Decrement or increment the counter after loop pass.
  - Test count and branch (leave loop if done):
    - If “no” (count not complete), go back to start of the loop.
    - If “yes” (count complete), exit the loop.

## Definitions of Arrays

- **An array is a set of data blocks (bytes, words, arrays[!], etc.) of equal size, stored contiguously in memory.**
- **Other examples of arrays include employee data files of a large company (or a university), or similar collections of information.**
- **A two-dimensional  $n$ -by- $n$  determinant which is used in the solution of algebraic equations is also an array, in this case of the equation coefficients of the  $n$  equations to be solved to find  $n$  unknowns.**

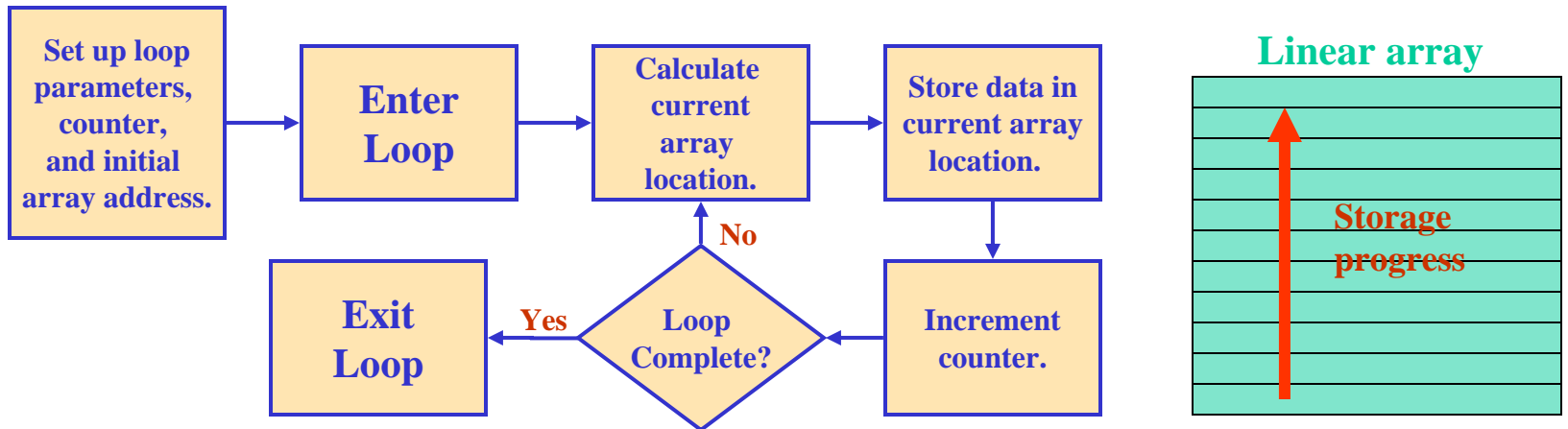
## Why a Fill-Array Demo?

- Often, in array calculations (such as the case of an  $n$ -by- $n$  determinant), many of the elements of the array have a value of 0 or some default value.
- It is relatively easy to populate an array by clearing it or storing a common value in all array locations and then changing the few array elements that have other values.
- The two programs today do this array initialization:
  - Lecture 15 Demo 1, Linear Array Loop demonstrates the initialization of a linear or one-dimensional array.
  - Lecture 15 Demo Program 2, 2-D Array Loop for a 4-by-3 two-dimensional array.

# Loop Programming Example #1

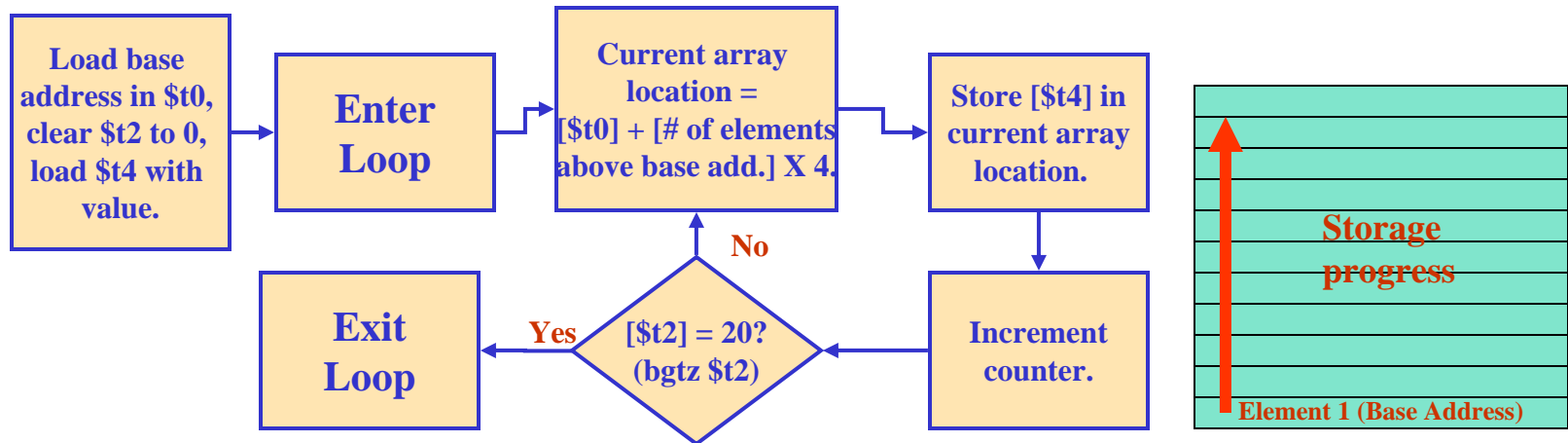
- **Lecture 15 Demo Program 1, Linear Array Loop:**
  - For this 1-dimensional problem, note that the index keeps track of the “position in line” of the current element of the array to be populated.
  - A single counter keeps track of the position in the array.
  - When the counter counts up to 20, all elements of the array have been loaded with the common value.
  - The beq (branch if equal) is used to determine when the loop is completed (the loop counter is up to 20).
  - Four (4) is added each time to the former address to get the new address, since  $\text{word address}_{n+1} = \text{word address}_n + 4$ .

# Filling a Linear Array



- The loop is set up (**counter is initialized**) and entered.
- Each time through the loop, the new data location in the array is calculated, data is stored there, and the loop counter is decremented (in this case).
- A branch instruction tests the counter, and the program goes through the loop once more or exits, depending on the counter value.
- The diagram on the next page shows the same loop with more detail.

# Expanded Detail of 1-D Array Loop



- Note array loop maintenance sequence:
  - Put base address in register to do register indirect addressing
  - Storing data in array using register indirect addressing
  - Counter keeps track of number of loop passes
  - The next storage address is calculated as:
 
$$[(\# \text{ of data words up the list}) \times (4 \text{ bytes/word})] + \text{base address}$$

## Lecture 15 Demo Program 1, Linear Array Loop

```
#  
# SPIM code to initialize the elements of a 1-d array. This program will populate a linear array  
# ("list") with a constant number (0x fedcba98). The linear array has 20 data locations which  
# must be filled. Note that each element is a word so that its size (in terms of memory space) is 4  
# (for 4 bytes).  
# Register assignments: $t0 -- base address (address of ar1[0]); $t1 -- current address to store the  
# data word; $t2 -- ctr and index; counts number of times through the loop and also acts as an  
# index to the current element # (i.e., indicator of the number of elements already processed)  
# $t3 -- offset = index times element size; $t4 -- val = value to be stored in each array element
```

**.data**

```
ar1: .word 0:20      # array of 20 integers (4 bytes each)
```

**.text**

```
main: la $t0,ar1     # load base address of array  
      move $t2,$0    # initialize array index to 0  
      li $t4,0xfedcba98 # put value to be stored in array in $t4  
  
loop: mul $t3,$t2,4   # loop begins here; compute offset in bytes  
      add $t1,$t3,$t0 # compute address of current array element  
      sw $t4,0($t1)   # store value in current array element  
      addi $t2,$t2,1  # increment array index  
      blt $t2,20,loop # branch back to loop if ctr < 20  
  
      li $v0,10  
      syscall        # program over  
#  
# end of file Lecture 15 Demo Program 1
```

# Exercise 1

- Construct a program to load the byte in memory labeled “str” into \$a0 and output it if it is a lower-case ASCII character.\*
- Ignore it if it is NOT a lower-case ASCII character.
- Continue inputting and processing bytes until you encounter a null (0), then stop.
- The data and text declarations are given, as is the “main” label.

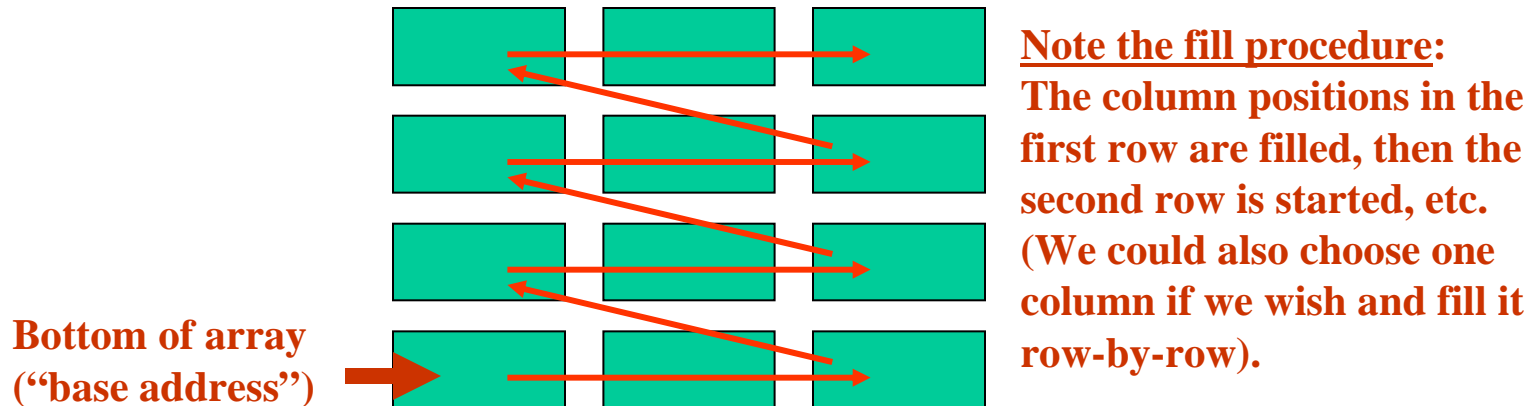
```
.text  
main:  
  
  
  
  
  
  
  
  
  
done:    li $v0,10  
          syscall  
          .data  
  
str:     .asciiz “ab12Cd”
```

\* Lower-case ASCII characters have numeric values between 0x61 and 0x7a.

## **Filling a Two-Dimensional Array**

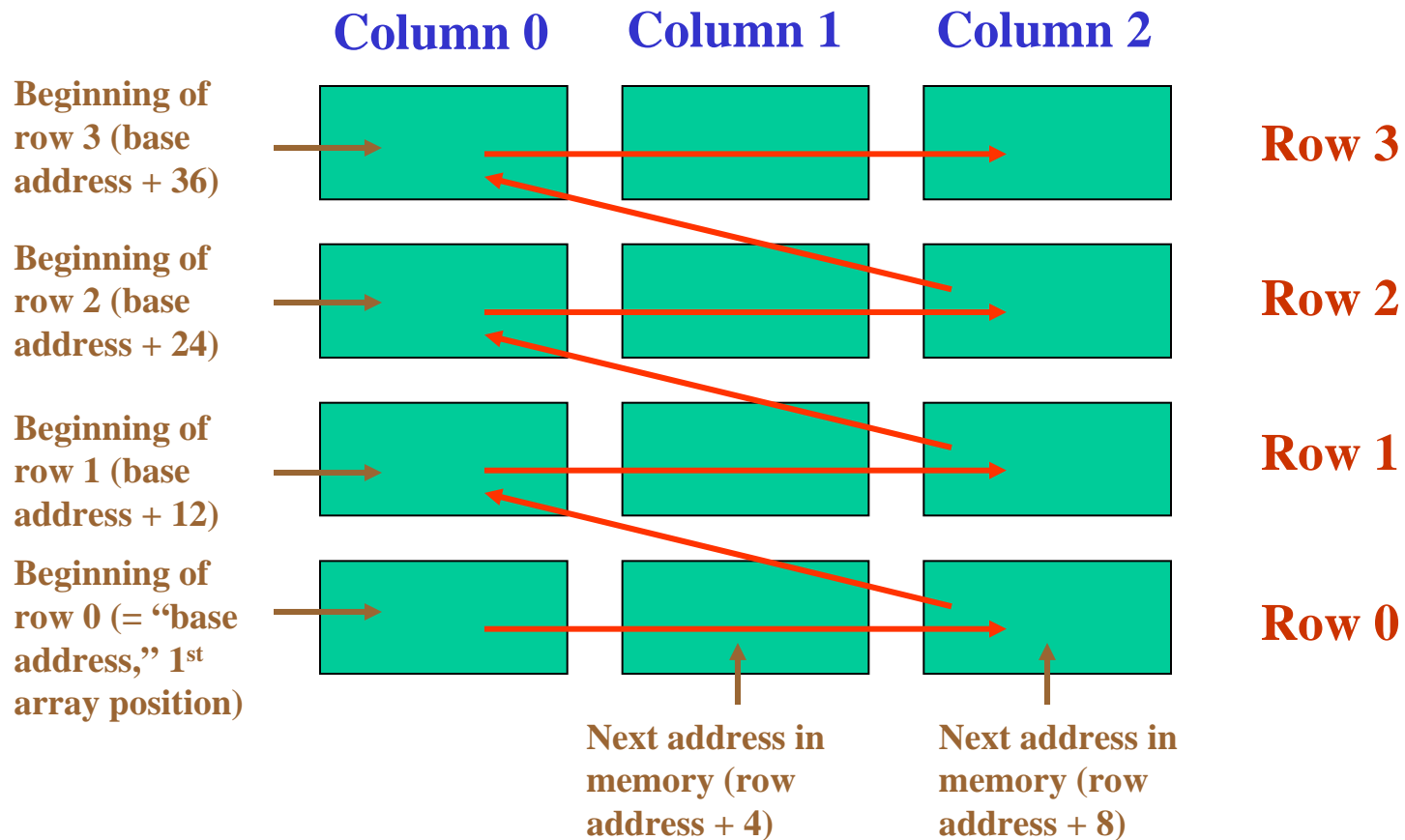
- **Filling a two-dimensional array is more challenging, since SPIM has no concept of 2 dimensions.**
- **A two-dimension problem is not difficult for a small array (we could treat a small 2-D array as a linear array).**
- **However, for this demonstration, we populate a small 2-D array with a common value in order to see how the programmer might navigate through two dimensions.**
- **In a 2-D array, to have an orderly method of storing values, we choose to treat the array as a series of rows, each of which has several column positions (next slide).**

## Filling a Two-Dimensional Array (2)

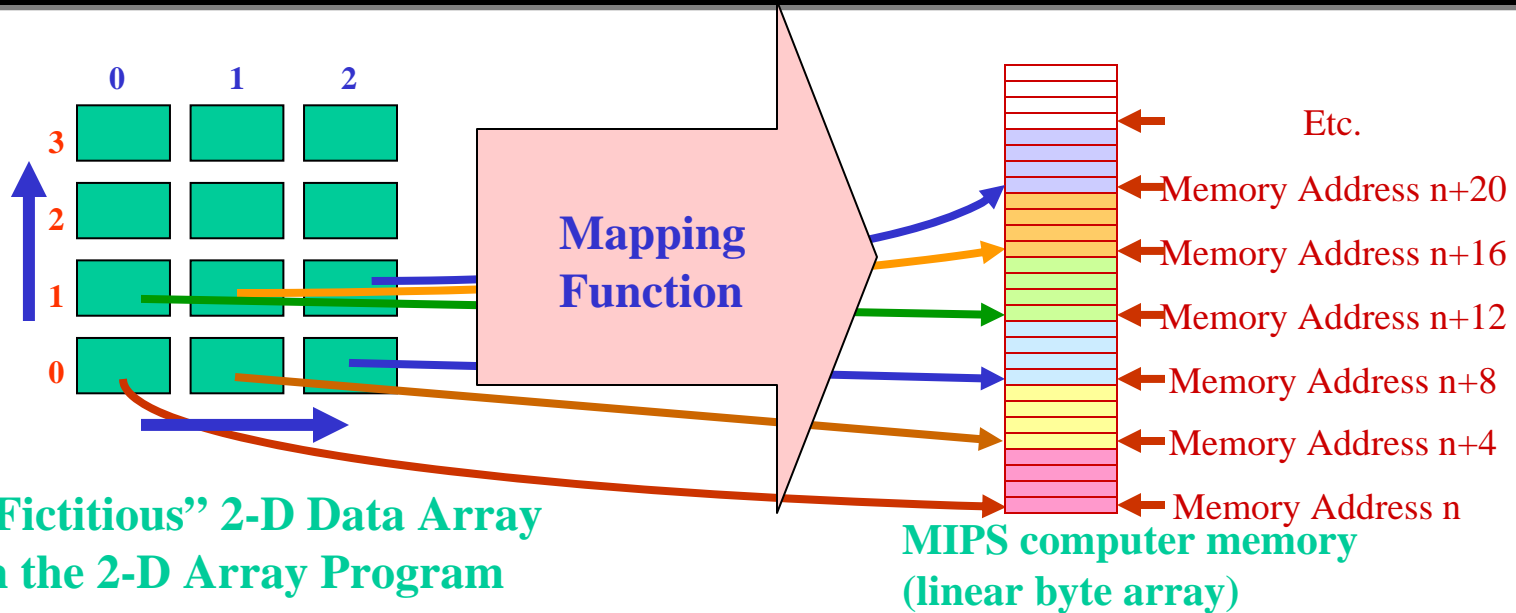


- In the 3 X 4, 2-D array above, the storage method is shown.
- Note that we will need two loops to provide this array fill – an “outer” row loop and an “inner” column loop (“row major order”).
- The two loops will be “nested” -- that is, the column loop is inside the row loop, so that a row is chosen, and then the column locations on that row are filled. After a row is filled, the column loop exits to the row loop, which starts a new row.

# 2-D Array Terminology

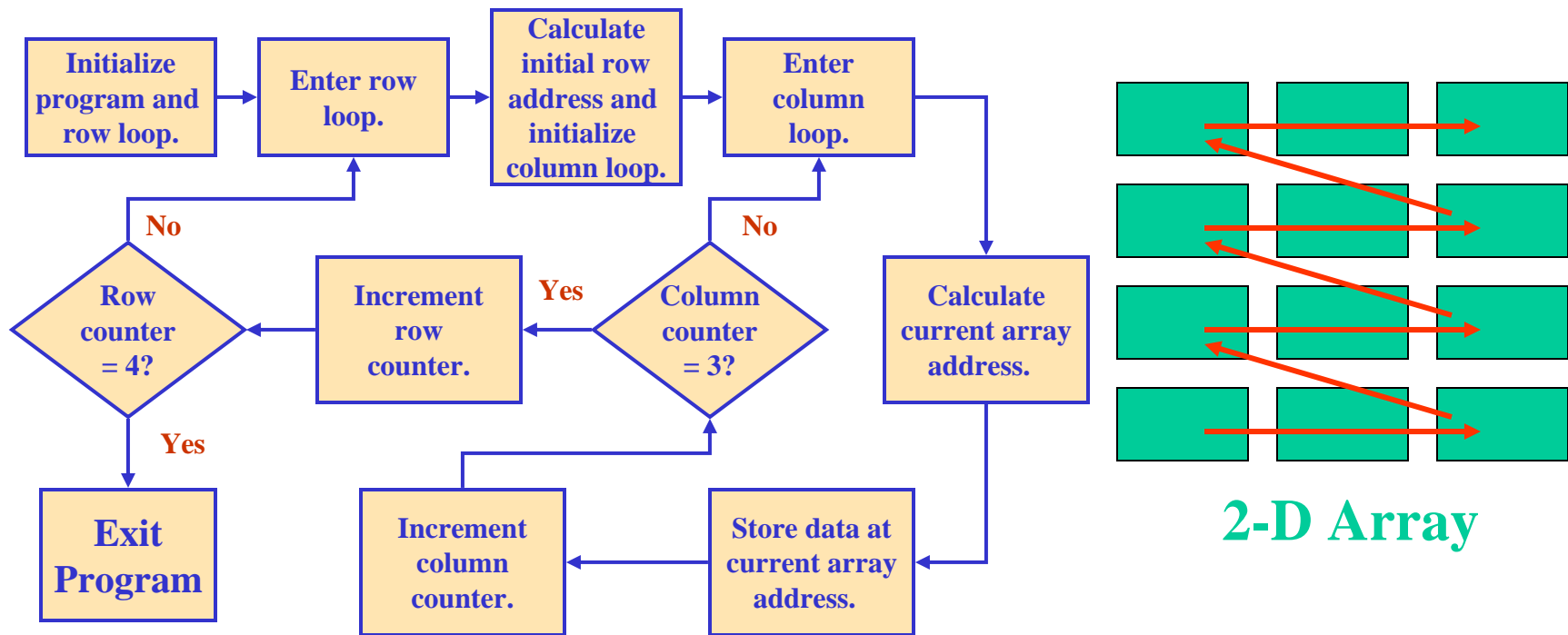


# Array Positioning: Mapping an Array Element in the 2-D Array Program to Its Actual Position in Memory



- The 2-D array is a construct; it does not exist except in the program.
- The MIPS computer only knows how to address bytes in memory.
- The program must provide a mapping function from our conceptual 2-D array space to the real byte addresses in linear memory.

# Program Flow Chart; Filling a 2-D Array



- **Filling a 2-D array requires keeping up with row and column positions.**

# Pointers

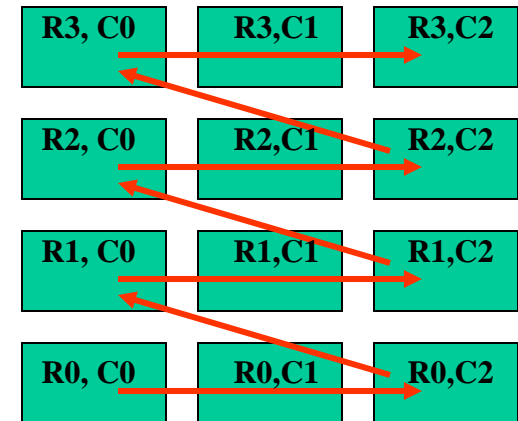
- What is a pointer? A pointer is a directional vector that points to a memory element (as in an array).
- A pointer can be used to access a 2-dimensional array elements, thereby simplifying the array fill.
- There is good and bad news about pointers:
  - **Pointer-based fills use more memory than index computation.**
  - **However, pointers result in a lower instruction count, and most multiplication is avoided (multiplication is time-consuming).**
- A pointer approach avoids multiplies for row and column offsets:
  - **Starting a new row simply involves adding 12 bytes, in our case.**
  - **In a row, changing columns involves adding 4 bytes (4 bytes/word).**
  - **An element position is just (base address) + (current row offset) + (current column offset).**

# Fill Analysis Using Pointers

- The position of any element in the array is its row/column coordinate. Each column position past 0 = +4 bytes; each row higher than 0 = +12 bytes added to the offset.
- The memory address of an array word is:  

$$\text{element address} = \text{base add.} + r_{\text{offset}} + c_{\text{offset}}$$

$$\text{Or, element address} = \text{base add.} + (12 \text{ bytes/row above } 0) + (4 \text{ bytes/col. past } 0).*$$



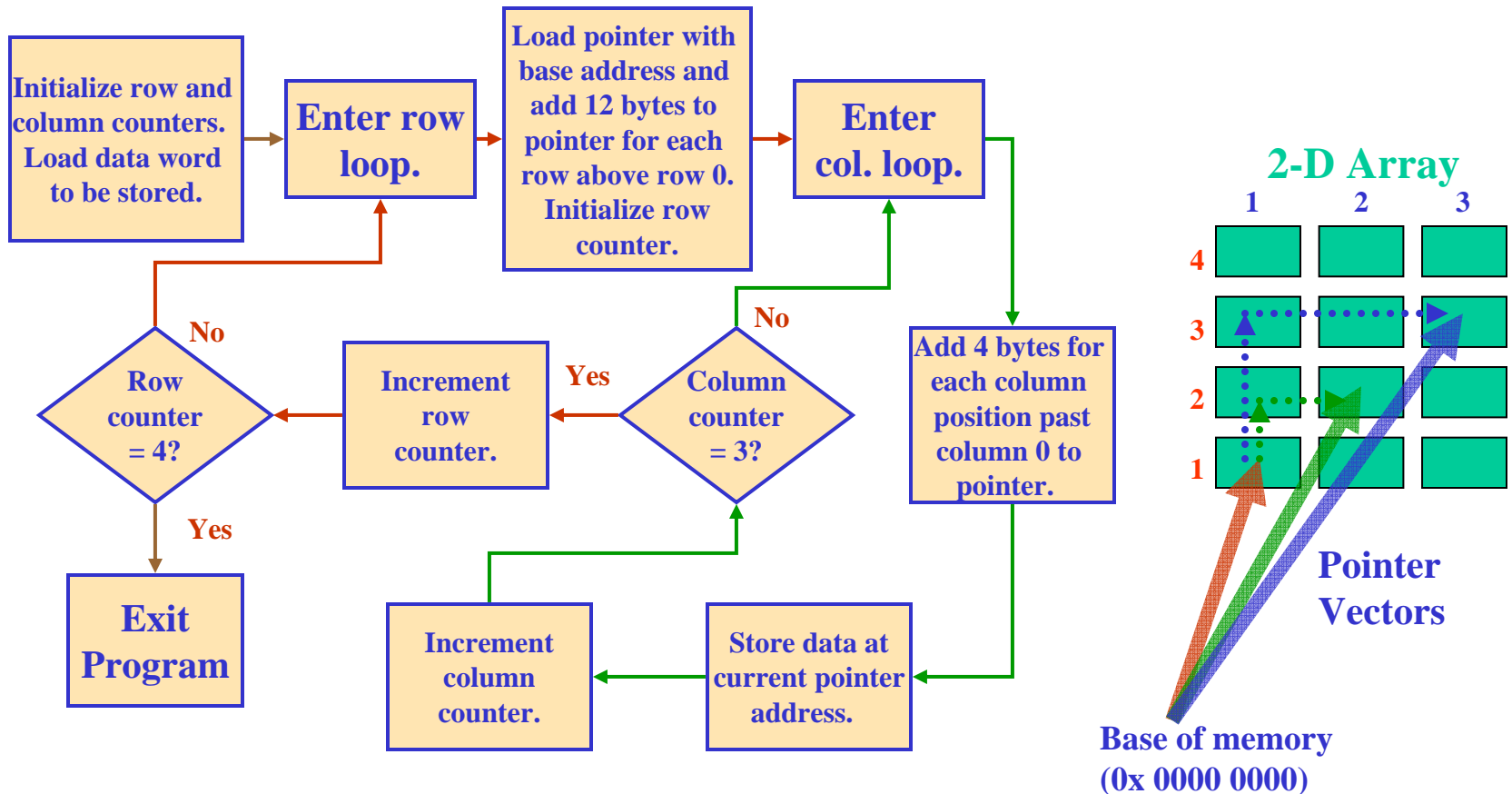
2-D Array

\*Example: The address of the  $R_2, C_1$  array element = base address + (12 + 12) + 4, Assuming the array starts at the start of data memory (0x 1001 0000), then the real address of the  $R_2, C_1$  array element in memory is:  $0x\ 10010000 + 24 + 4 = 0x\ 10010000 + 28 = 0x\ 10010000 + 0x\ 1c = 0x\ 1001\ 001c$ . Note that the “base address” is that of  $R_2, C_1$ .

## Array Analysis (4)

- We can now flesh out the loop program shown earlier.
- That is, the 2-D program:
  - **Initializes program and row loop (sets row count to 0).**
  - **Enters the row loop.**
  - **Initializes the column loop functions (sets column count to 0).**
  - **Enters the column loop.**
  - **Stores three elements, adds 4 bytes column offset after each store.**
  - **Exits the column loop.**
  - **Checks loop count and adds a row offset if loop not yet done.**
  - **Re-enters the column loop and repeats column procedure.**
  - **Continues until the last row is complete, then exits the loop.**
- A more-detailed flow chart is shown on the next slide.

# 2D Array Loop Using Pointers



## Filling a 2-D Array

- **Lecture 15 Demo Program 2**
- **This program initializes the elements of a 2-D array using a pointer.**
- **Note how the use of the pointer subtly changes the program and improves performance:**
  - **Slightly fewer instructions (would be many more for a large program!).**
  - **No multiplies.**

```

#      Lecture 15 Demo Program 2, Utilizing Pointers
# SPIM code to initialize the elements of a 2-d array using pointers
# Register assignments: $t0 = pointer to current beginning of row,
# $t1 = row counter and index, $t2 = column counter and index,
# $t3 = pointer to current address to store data, $t4 = value to be stored in each array element

```

```

.data

```

```

ar2: .word 0:12      # array of 12 integers (4 bytes each)

```

```

.text

```

```

main: la $t0,ar2     # set pointer to start of array
      move $t1,$0    # initialize row counter/index
      li $t4,0xf9e8d7c6 # put value to be loaded in array in $t4
rloop: move $t2,$0    # initialize column counter/index
      move $t3,$t0   # initialize col. pointer to 1st element of row

cloop: sw $t4,0($t3)  # store val in current array element
      addi $t2,$t2,1  # increment column counter/index by 1
      beq $t2,3,nxtrow # go to next row if column counter = 3
      addi $t3,$t3,4  # increment the column pointer
      j cloop         # go back and do another column
nxtrow: addi $t1,$t1,1 # Increment row counter/index by 1
      beq $t1,4,end   # leave row loop if row counter = 4
      add $t0,$t0,12  # increment the beginning-of-row pointer by
                        # the number of bytes in a row (12)
      j rloop        # start next row

end:li $v0,10        # reach here if row loop is done
      syscall        # end of program

```

## Exercise 2

- **Study the program to the right. It is outputting characters from the string, but not all of them.**
  1. **What is the actual character string that is output? It will, of course, be a subset of the string labeled “number.”**
  2. **Unfortunately, the program will not run properly, because two instructions are wrong. What are they?**

```
.text
main:  la $t2,number
      li $v0,11
load:  lw $a0,0($t2)
      beqz $a0,done
      blt $a0,0x30,filter
      bgt $a0,0x39,filter
      syscall
nextlet: addi $t2,$t2,1
      j load
filter: blt $a0,0x61,nextlet
      bgt $a0,0x7a,nextlet
      syscall
      j main
done:  li $v0,10
      syscall
.data
number: .asciiz "1234abcdEF56GHi89"
```

# Homework

- **As usual:**
  - Write down the two or three most important things learned today and add to your list.
  - Write down two or three concepts or ideas that are not clear. After review, if you still have questions, see me during office hours.
- **Read: Pervin 4-4.3 and Patterson and Hennessy 2.6.**
- **Start work on Homework #10.**
- **Also, copy or list Lecture 15 Demo Program 2 into Notepad, and run in PCSPIM (be sure to single-step!).**
- **Write the following program: After declaring the following data:**  
`.word 0xfe819837`, write a simple loop to examine each hex digit of the number declared and find all the “8” digits. Print out your answer. Since there are two 8’s, you can easily check your work! If you need help, see me during office hours.