Loops in SPIM

- Iterative loops are easy to structure in a higher-level language, in which instruction sets may be repeated innumerable times via "do," "if," or "for."
- In assembly language, there are no such commands; loop details must be designed by the programmer.
- That is, in assembly language, the programmer must do all the "dirty work" of creating the loop.
- This includes writing loop software, keeping track of the number of loop passes, and deciding via a test when the looping software has completed its job.

Loops (2)

- **Requirements for an assembly language loop:**
  - **Entry function to get into the loop ("call function").**
  - **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.**

A Loop Program: Reversing a String

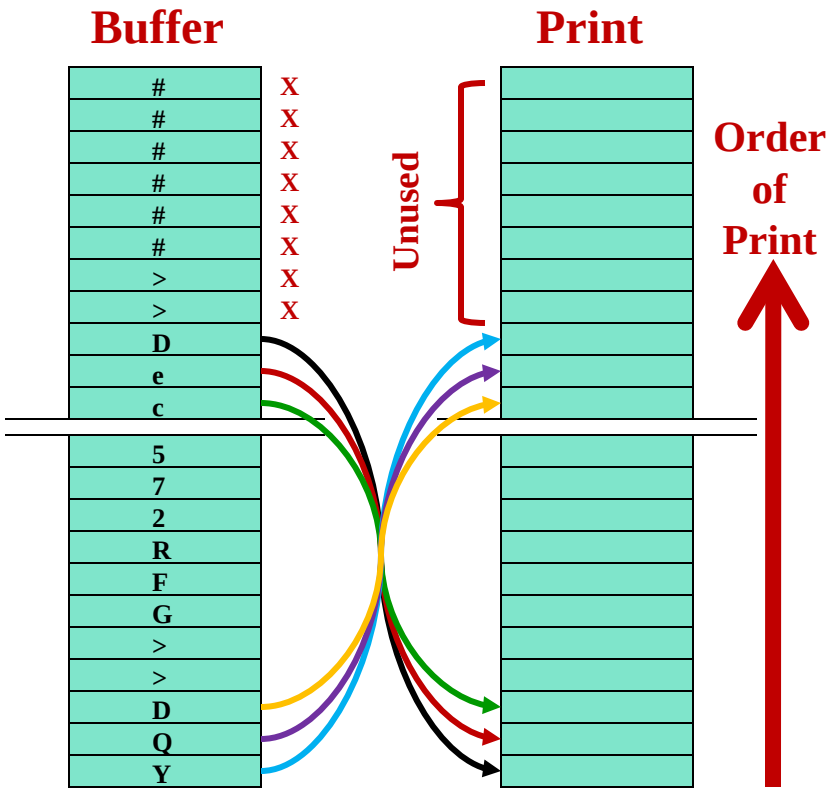
- **The following is an example of building a loop program.**
- **Problem: Assume a data block (ASCII characters) is transmitted to your MIPS computer to be displayed, but it was transmitted in reverse order. The sequence must be inverted to print out.**
- **Further, transmission control characters must be stripped out of the data before displaying, including *, &, <, >, and #, the last of which is used to denote end-of-transmission.**
- **This is not a big problem today, as data transmission protocols are very consistent. However assume that it's a few decades ago, when data transmission was very unpredictable and standards were few.**
- **The problem is to invert the string of characters and print to the console, stripping out the control characters.**

Analysis

- **To print out reversed characters in a string, there are two options:**
  - **Start at the far end of the characters and come forward, printing a character at a time.**
  - **Reverse the string first, then print out all at once.**
- **For this example, the second option is chosen: Reverse the entire string before printing.**
- **We assume that another program received the transmission, and stored the characters in a memory area whose initial byte address is "buffer." Further, assume that the number of characters received and stored is always 96. If less than 96 characters are needed for data, the remaining characters are #'s (= EOT).**
- **The following program is a variation of a program in Pervin, Chapter Four.**

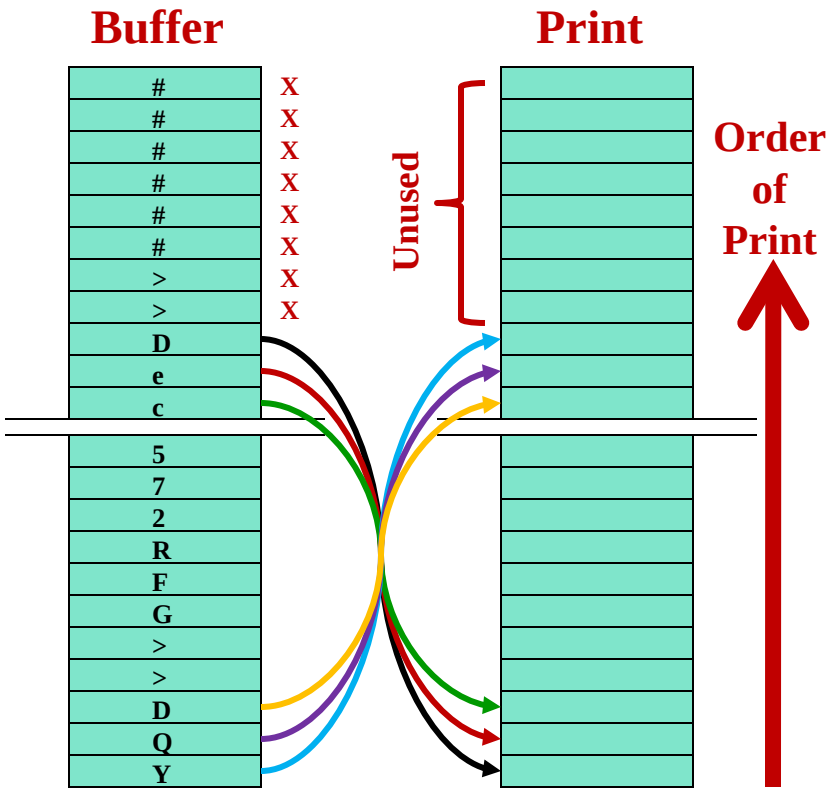
Analysis (2)

- Character string in "buffer" must be reversed.
- However, the control and format characters must be stripped out first. These are NOT printed.
- Since the buffer is reversed, the program starts at the TOP.
- Each character is tested, and the five control characters are eliminated as they are encountered.



Analysis (3)

- As printable characters are encountered, they are stored into the print area, starting at the lowest memory location (syscall four initiates a printout of characters at the lowest address of the character list).
- The print area is not used entirely, as the printable character total is much smaller.
- When characters are reversed, the set is printed to the console.



Register Choices

- **We need four registers:**
  - **$t0 – Pointer to character address in "buffer" (received block)**
  - **$t1 – Pointer to character address in reversed data block**
  - **$t2 – Counter; counts number of times through the loop**
  - **$t3 – Contains current character being analyzed**
- **We need to define and reserve two blocks of data:**
  - **"buffer" – The block of reversed data to be processed**
  - **"display" – The reserved area for the printable data**

Program Initiation

- There is a need for two "pointers"* one to the buffer and one to the print area.
- Since characters in "buffer" are reversed, we start at the last one first (pointer $t0).
- The other pointer holds the address of the first vacant character position in the print area (pointer $t1).
- We also clear the counter ($t2).

```
main:   la $t0,buffer      # First character address
        add $t0,$t0,95     # plus 95, as 96 characters
                           # always transmitted.
        la $t1,display     # Load base address of data
                           # space for reversed block.
        move $t2,$0        # Clear counter register.
```

* A "pointer" is a vector or number that identifies a memory location.

"Housekeeping"

- We need to discard all control characters.
- We identify the control characters by their hex codes, so we eliminate them, as seen in the program segment to the right.

```
loop:   lb $t3,0($t0)      # Loop begins here.
        beq $t3,35,next    # Test and eliminate
        beq $t3,38,next    # characters #, &, *,
        beq $t3,42,next    # <, and >.
        beq $t3,60,next
        beq $t3,62,next
```

(Note: "next" is the label for the program part that gets the next character)

Character Placement

- When a printable character is identified, the character is stored in order in the print memory area.
- The counter is incremented, and if not yet to 96, the loop is set up to get the next character.

```
        sb $t3,0($t1)       # Store valid character
                            # in display block.
        addi $t1,$t1,1      # Set print pointer to
                            # character address.

next:   addi $t2,$t2,1      # Increment counter
        beq $t2,96,print    # Analysis done yet? *
        sub $t0,$t0,1       # Set block pointer to
                            # next character
                            # address.
        j loop              # Go to analyze next
                            # character.
```

* If not, increment pointers and continue.
If so, print out reversed block.

Print Routine

- When all 96 characters are identified, the now-correctly-ordered list of printable characters is printed to the display.
- The program then stops.

```
print:  la $a0,display  # Print reversed block
        li $v0,4
        syscall

        li $v0,10
        syscall        # Program over
```

```
#                       Lecture 15 Demo Program 1, Reversing a Character String for Display
        .data
buffer:         .ascii "2.12<< :tsaceroF** ,0.32<< :tnerruC** MR**&& ;0.25>> :tsaceroF** ,6.94>> :tnerruC** GK**########"
display:        .space 80

        .text
main:   la $t0,buffer           # Load last character address of data block to be reversed
        add $t0,$t0,95          # (First character address plus 95, as 96 characters always transmitted)
        la $t1,display          # Load base address of data space for reversed block
        move $t2,$0             # Initialize counter to 0

loop:   lb $t3,0($t0)           # Loop begins here; load the next byte
        beq $t3,35,next         # Test and eliminate characters #, &, *, <, and >
        beq $t3,38,next
        beq $t3,42,next
        beq $t3,60,next
        beq $t3,62,next
        sb $t3,0($t1)             # Store valid character in display block
        addi $t1,$t1,1            # Set display space pointer to next character address

next:addi $t2,$t2,1             # Increment counter
        beq $t2,96,print        # Analysis done yet? If not, increment pointers and continue
        sub $t0,$t0,1           # Set block pointer to next character address
        j loop                  # Return to loop for next character
                                # (Do not store a character here, so do not increment $t1)
print:  la $a0,display          # Print reversed block
        li $v0,4
        syscall
        li $v0,10
        syscall                 # Program over
```

Program 2

- **Declare the ASCII string ab12Cd, labeled "str."**
- **Construct a program to load the bytes in memory labeled "str" into $a0 and output them only if a byte 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.**

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

Program 2 Solution

```
        .text
main:   li $v0,11
        la $t1,str
loop:   lb $a0,0($t1)
        beqz $a0,done
        blt $a0,0x61,next
        bgt $a0,0x7a,next
        syscall
next:   addi $t1,$t1,1
        j loop

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

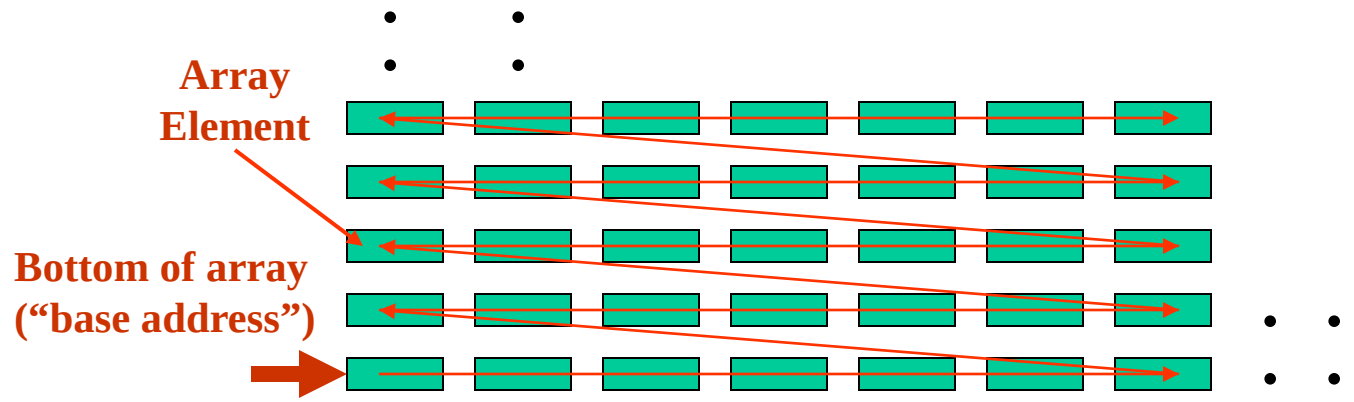
"Nested" Loops

- A more complex loop problem is one of two loops, one inside the other (sometimes referred to as "nested" loops).
- A good example: Populating a two-dimensional array.
- An array is a set of data blocks (bytes, words, arrays[!], etc.) of equal size, stored contiguously in memory.
- Example: 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.
- Filling a 2-D array is a good nested loop example.

Problem Rationale

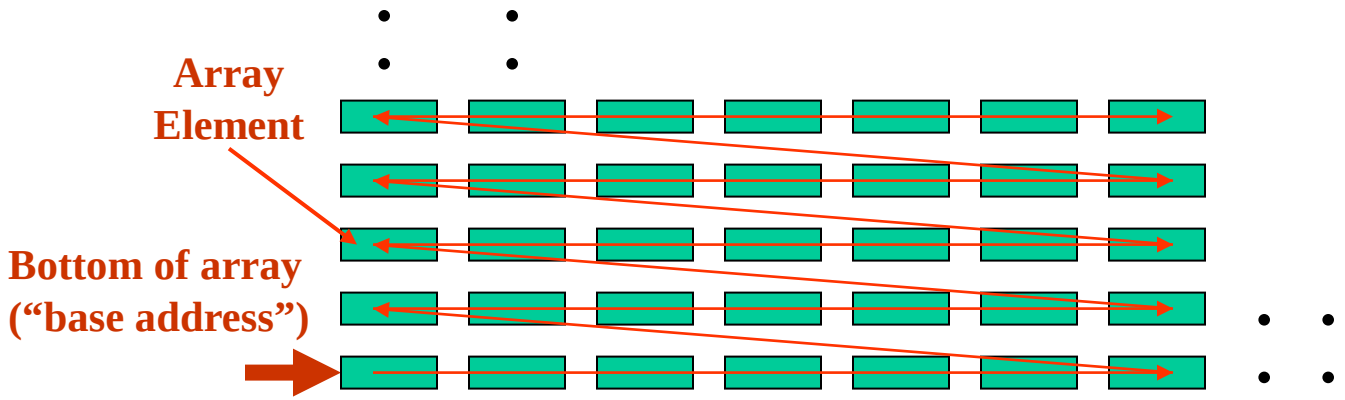
- Often, in a large determinant calculation, many elements are 0 or have some default value.
- It is relatively easy to populate an array by storing a default value in all array locations and then changing the relatively few array elements that have other values.
- The second loop example is such an exercise.
- In Lecture 15 Demo Program 2, a 2-D Array representing a 20x20 determinant is populated with a default value.
- Such a large array (400 32-bit numbers, or 1600 bytes) represents the type of problem for which a computer is very useful.

Filling a Two-Dimensional Array



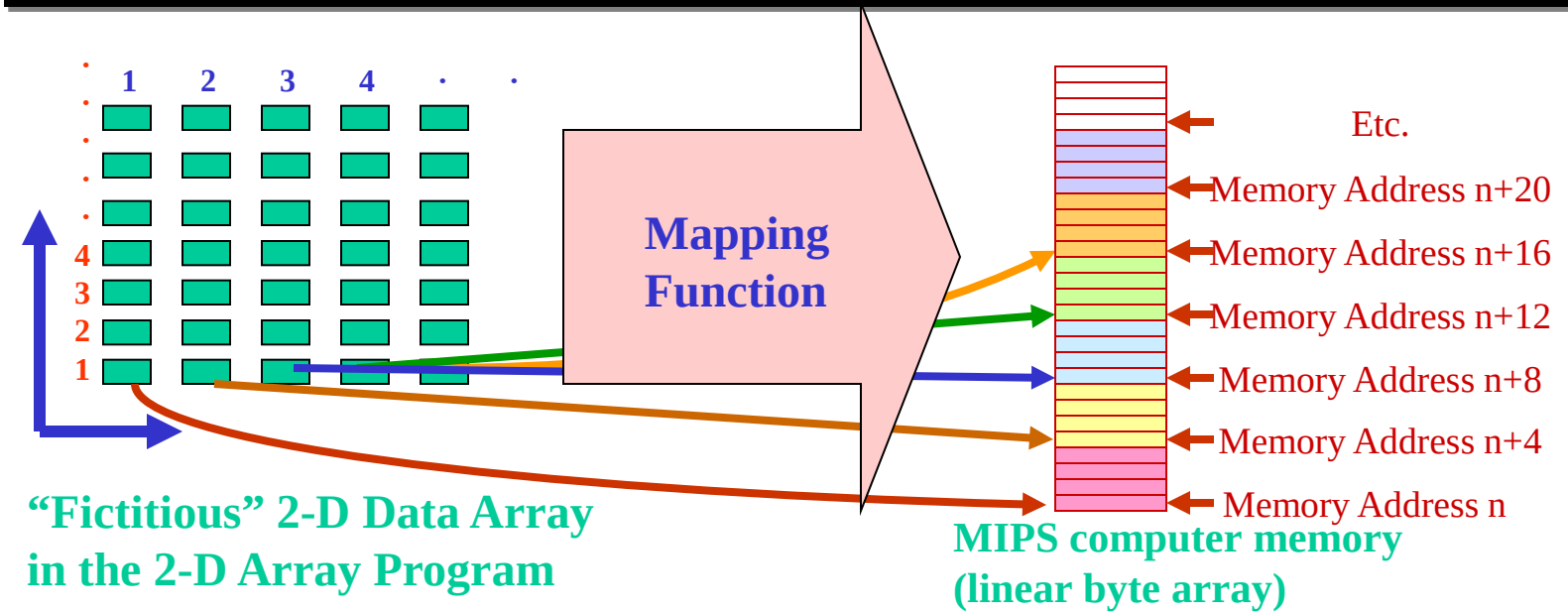
- When dealing with a 2-D figure (like a 2-dimensional determinant), it is usually worthwhile to operate on it as a 2-D figure, even though it could theoretically be treated as a linear list.
- A 2-D approach is used here. Assuming a 20-row, 20-column array or matrix, the storage method is shown.
- We will need two loops to provide an orderly fill – an "outer" row loop and an "inner" column loop ("row major order").

Filling a Two-Dimensional Array (2)



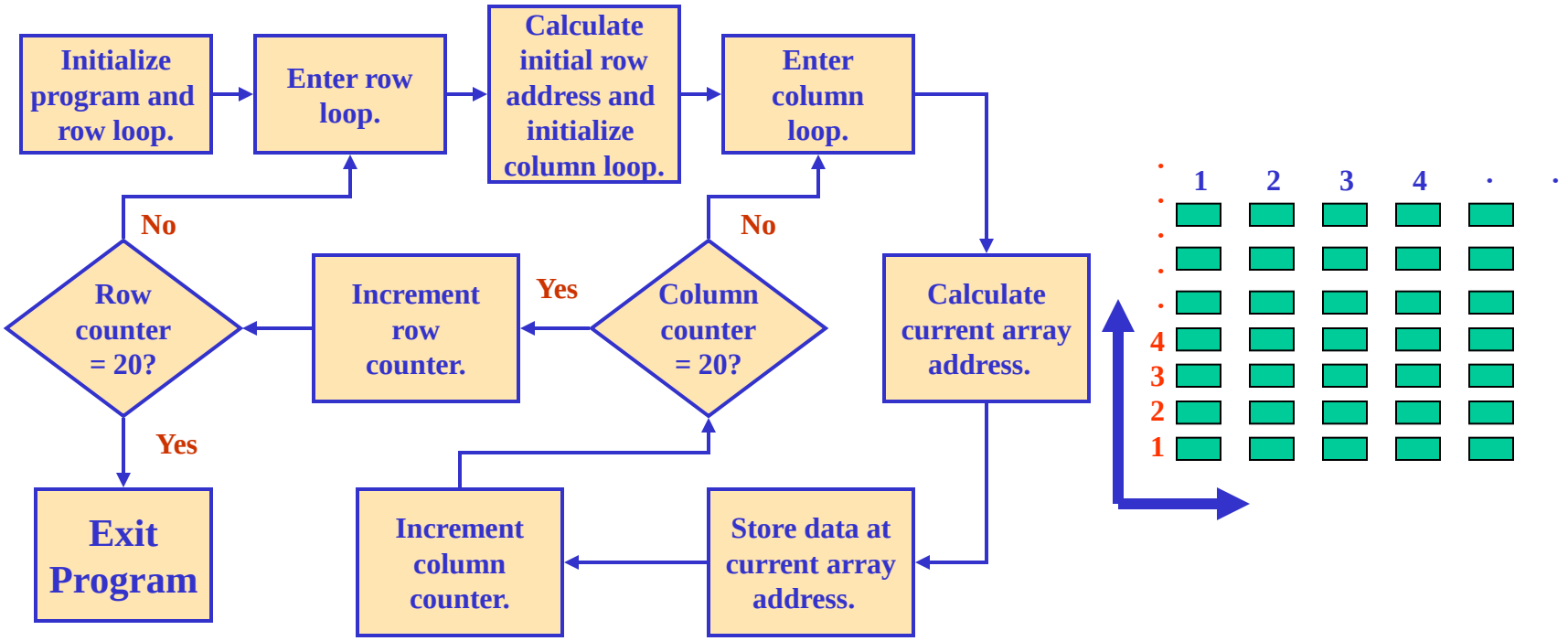
- Fill procedure: Column positions in the first row are filled, then the second row is started, etc. (Could also choose one column if we wish and fill it row-by-row ["column-major order"].)
- The two loops will be "nested" -- that is, the column fill loop is inside the row loop, so that a row is chosen (starting at the bottom), and then the column locations in that row are filled. After a row is filled, the column loop exits to the row loop, which starts a new row.

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.

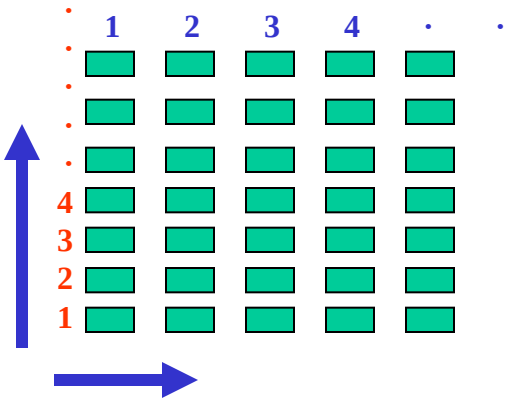
Program Flow Chart; Filling a 2-D Array



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

Fill Analysis Using a Pointer

- As in the first example program, we use a pointer, simply a register used as a directional vector, that points to a memory element (in this case, an array element).
- The position of an array element is its row/column coordinate. Each column position past column 1 = +4 bytes; each row higher than row 1 = +80 bytes (+20 elements or numbers per row = 80 bytes).
- The memory address of an array word is:
  element address = base add. + r_{offset} + c_{offset}
  Or, element address = base add. + (80 bytes/ row past row 1) + (4 bytes/col. past col. 1).*

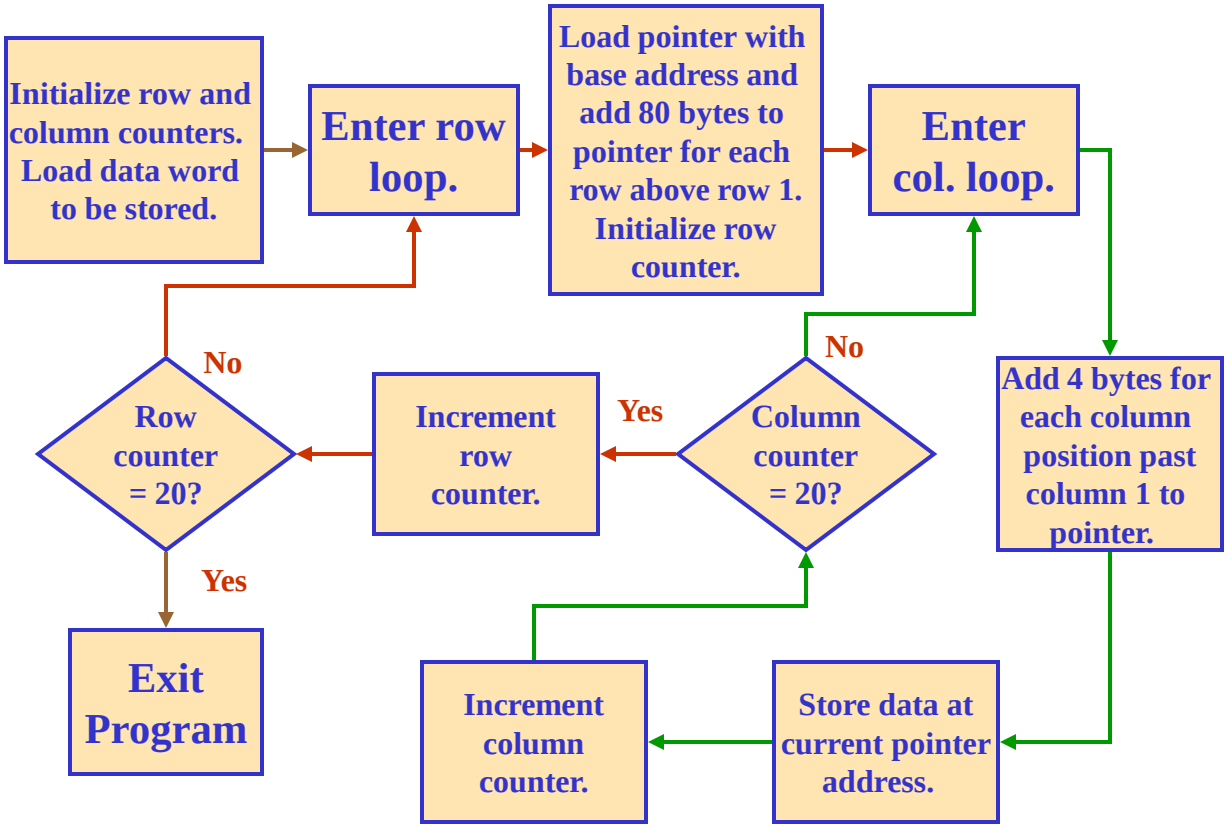


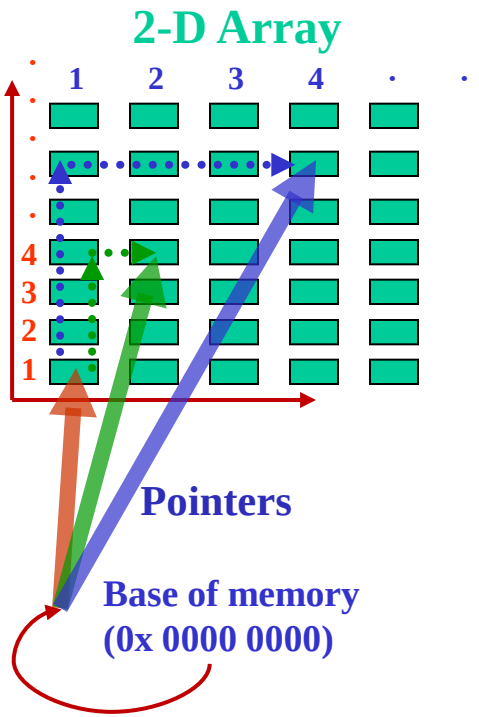
*Example: The address of the R_4, C_3 array element = base address + (80x3) + (2x4). Assuming the array starts at 0x 1001 0000, then the real address of the R_4, C_3 is:
0x 1001000 + 240 + 8 =
0x 10010000 + f0 +8 =
0x 100100f8

Array Analysis (4)

- **Fleshing out the loop flow chart shown earlier in terms of events:**
  - **Initialize program and row loop (sets row count to 0).**
  - **Enter the row loop.**
  - **Initialize the column loop functions (sets column count to 0).**
  - **Enter the column loop.**
  - **Store twenty elements, adding 4 bytes/column offset after each store.**
  - **Exit the column loop.**
  - **Check loop count and add row offset if row loop not yet done.**
  - **Re-enter the column loop and repeat column procedure.**
  - **Continue until the last row is complete, then exit the loop.**
- **The refined flow chart is shown on the next slide.**

2D Array Loop Using Pointers





Nested Loop Program

- **Lecture 15 Demo Program 3**
- **Initializes the elements of a 2-D array using a pointer.**
- **Note how "nested" loops operate:**
  - **Outer loop initialized (counter = 0), then entered.**
  - **Inner loop initialized, then entered.**
  - **Stores begin.**
  - **After row complete (20 words), exit column loop to row loop.**
  - **Continue until columns = 20, rows = 20.**

```
#          Lecture 15 Demo Program 3, Nested Loops
# SPIM code to initialize the elements of a 2-d array using a pointer
#    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:        .space 1600         # Array of 400 integers (4 bytes each) in 20X20 array

            .text
main:       la $t0,ar2          # Initialize pointer to start of array
            move $t1,$0         # Initialize row counter/index
            li $t4,0x4ff6       # 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 value in current array element
            addi $t2,$t2,1      # Increment column counter/index by 1
            beq $t2,20,nxtrow   # Go to next row if column counter = 20
            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 $t1,20,end      # Leave row loop if row counter = 20
            add $t0,$t0,80      # Increment the beginning-of-row pointer by
                                #    the number of bytes in a row
            j rloop             # Start next row

end:        li $v0,10           # Reach here if row loop is done
            syscall             # End of program
#
#    End of file Lecture 15 Demo Program 3
```

Program 4

- **Construct a program that is a variation on the first exercise. The data declaration below is the one to put in your program.**
- **Your program should output ONLY the numbers and the lower-case characters, ignoring the upper-case characters.**
- **My program took 15 lines, including the data print out commands.**

```
                .data
number:         .asciiz "1234abcdEF56GHi89"
```

Program 4 Solution

The character output is: 1234abcd56i89.

```
        .text
main:   la $t2,number
        li $v0,11
load:   lb $a0,0($t2)
        beqz $a0,done
        blt $a0,0x30,filter
        bgt $a0,0x39,filter
        syscall
nexlet: addi $t2,$t2,1
        j load
filter: blt $a0,0x61,nexlet
        bgt $a0,0x7a,nexlet
        syscall
        j nexlet

done:   li $v0,10
        syscall
        .data
number: .asciiz "1234abcdEF56GHi89"
```

SPIM Skill Improvement

- **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.**