Arrays & Strings

Textbook Section 6.1, page 200

Arrays - What are they?

An array is an **indexed** collection of values of the same type.

An array is defined by:

- its start address label
- the number of elements size
- the size of an element type

General form of an array declaration:

```
<type> <name>[<size>];
```

Arrays - What are they?

General form of an array declaration:

```
<type> <name>[<size>];
```

Meaning: “declare a variable called name that is an array of <size> elements, with each element of type <type>”

```
int numbers[3];
```

Create a variable called “numbers” that is an array of three elements, each element is of type integer, on a typical 32 bit architecture each integer is 4 bytes

This is a request for 3 * 4 = 12 bytes of memory.

Arrays

In the simplest terms an array is simply a block of memory which is composed of elements that are all the same size.

```
int numbers[3] = { 3, 2, 1 };
```

Assuming numbers starts at address $A004_{16}$:

- It requires 12 bytes of memory
- Has elements at locations $A004_{16}$, $A008_{16}$ and $A00C_{16}$
- The next available address is $A010_{16}$

Arrays

```
char my_string[256];
```

Assuming my_string starts at address $A004_{16}$

- requires 256 bytes of memory
- Has elements at locations $A004_{16}$, $A005_{16}$,... to $A103_{16}$
- The next available address is $A104_{16}$

Arrays - Declaring

To declare an array in 68000 asm, use the directives:

```
; declare an array with the given list of values
dc.<size> <values>
```

```
; declare an array with num-values spots
ds.<size> <num-values>
```

```
; declare an array with num-values spots each initialized to value
dcb.<size> <num-values>,<value>
```

Arrays

E.g.

```
numbers:      dc.l 1,2,3      ; 12 bytes
my_string:    ds.b 256        ; 256 character (bytes)
array:        dcb.b 10,$FF    ; 10 bytes init’d to $FF
```

Loading Address Registers

- Address registers are required to be able to use indirect addressing
- Two command in 68000 allow you to load an address into an address register
  - The **lea** command (load effective address)
    - lea *label*,a_n
  - The **movea** command (move will be replaced by movea)
    - `movea.l` *`label`*`,a`$_n$
- **IMPORTANT:** All addresses are always longwords (32-bits), when altering address registers the size operation must be `.l (L not 1)`!!

Loading Address Registers - lea versus move.l

- In general you should use **lea** when loading an address into an address register
- There are X reasons for using lea
  1. lea is faster, it takes fewer clock cycles
  2. lea can only be applied to size `.l (.L not .1)`
  3. lea does not have immediate addressing
     - cannot confuse #var with var
- You will not be penalized for using move(a), but it is not preferred

Arrays - Processing

```
; sum = 0
; for (i = 0, i < NUM_ELS, i++)
;     sum = sum + array[i]
; d0 = sum
; d1 = loop counter
; a0 = pointer to current element
```

Arrays - Processing

```
NUM_ELS equu 4                  ; num. elements in array1
init:       clr.w   d0 ; sum = 0
            move.w  #NUM_ELS-1,d1    ; init. loop counter
            lea array1,a0       ; j = 0
sum_loop:                            ; for i = NUM_ELS-1 downto -1
            add.w   (a0),d0     ; sum = sum + array[j]
            adda.l  #2,a0            ; j++
            dbra    d1,sum_loop
sum_done:   ...
array1: ds.w NUM_ELS
;           dc.w 1,2,3,4
```

Arrays - Things to Notice

- There is no label[index] notation in 68K assembly
- The array label addresses ONLY the first element of the block
- Initializing the array index is loading the array address into an address register
- In assembly language the loop control and array index in a counted loop are decoupled, LCV is i (d1), but array index is j (a0)

Arrays - Things to Note

- Moving to the next array element, i.e. index++, is done by changing the value in the address register – by the data size – 1, 2 or 4.
  - This is a significant source of error – since most programmers are used to using index++ and always add 1.
  - Remember pointer arithmetic from 1633 and ++ on pointers works on data size!
  - Problem solved by providing appropriate addressing mode(s)!
- When processing an array the programmer needs to consider errors
  - ex. summing student grades

Arrays - What is the problem with this code?

```
MAX_STUDENTS equ 100 ; max num. students in class

init:           clr.w     d0                  ; sum = 0
                move.w    num_students,d1     ; actual # of students
                beq       sum_done            ; if no students
                sub.w     #1,d1               ; init. loop counter
                lea       students,a0         ; j = 0
sum_loop:                                     ; for i = num_students-1; downto -1
                add.b     (a0),d0             ; sum = sum + students[j]
                adda.l    #1,a0               ; j++
                dbra      d1,sum_loop
sum_done: ...
num_students:   dc.w      0
students:       ds.b      MAX_STUDENTS
```

Arrays - The Problem

The problem is the grade sum can overflow

```
sum_loop:                       ; for i = num_students-1 downto -1
            add.b (a0),d0       ; sum = sum + students[j]
            adda.l #1,a0        ; j++
            dbra d1,sum_loop
```

What is the result if the first three student grades are all 100?

How can this be fixed?

Arrays - Incorrect Solution

Changing to add.w will NOT solve the problem. It will simply add another problem – two student grades will be combined into a single value. Why?

```
sum_loop:                              ; for i = num_students-1 downto -1
            add.w   (a0),d0        ; sum = sum + students[j]
            adda.l  #1,a0              ; j++
            dbra        d1,sum_loop
sum_done: ...
num_students:   dc.w        0
students:       ds.b        MAX_STUDENTS
```

Arrays - The Solution

The solution is to convert each grade into a word. This requires using another register/variable

```
init:           clr.w     d0                 ; sum = 0
                clr.w     d7                 ; conversion space - larger size
                move.w    num_students,d1    ; actual # of students
                beq       sum_done           ; if no students
                sub.w     #1,d1              ; init. loop counter
                lea       students,a0        ; j = 0
sum_loop:                                    ; for i = num_students-1; downto -1
                move.b    (a0),d7            ; sum = sum + students[j]
                add.w     d7,d0              ; since cleared hi byte - 0x00
                adda.l    #1,a0              ; j++
                dbra      d1,sum_loop
```

This works for unsigned values. What would need to be done to work on signed values?

Register Indirect with Post-Increment

Syntax: (An)+

Semantics:

- *An* is dereferenced, and the given value is used in the operation.
- After the operation the address is incremented by 1, 2 or 4 (based on operation size).

This is often used for stepping **forwards** through an array.

Register Indirect with Post-Increment

```
; d0 = sum
; d1 = loop counter
; a0 = pointer to current element
NUM_ELS equu 4              ; num. elements in array1
init:       clr.w   d0              ; sum = 0
            move.w  #NUM_ELS-1,d1   ; init. loop counter
            lea array1,a0       ; point to 1st array element
sum_loop:   add.w   (a0)+,d0        ; repeat update sum and advance
            dbra    d1,sum_loop ; until countdown complete
sum_done: …
```

Register Indirect with Pre-Decrement

Syntax: -(An)

Semantics:

- Before the operation the address is decremented by 1, 2 or 4
  - based on operation size
- An is dereferenced, and the given value is used in the operation.

This is often used for stepping **backwards** through an array.

Register Indirect with Pre-Decrement

```
; d0 = number to be printed
; d1 = loop counter
; a0 = pointer to previous element
NUM_ELS         equ         4                   ; num. elements in array1
init:           move.w      #NUM_ELS-1,d1       ; init. loop counter
                lea         array1,a0           ; point to 1st array element
                clr.l       d2                  ; convert w to l
                move.w      #NUM_ELS,d2
                lsl.l       #1,d2               ; offset = NUM_ELS * 2
                adda.l      d2,a0               ; point to element after array
loop:           move.w      -(a0),d0            ; for each value in array
                jsr         write_num           ; print number
                dbra        d1,loop
```

Strings

A **string** is an array of characters. In other words, a string is an array of bytes, each interpreted as an ASCII value.

Strings may be **fixed-length**: the size is hard-coded and all strings must use this length – unused array elements are **padded** with spaces. This method is rarely used.

Now, we consider the two ways of representing variable-length strings:

- null-terminated (as done in C/C++)
- byte-counted (as done in Pascal)

Null Terminated Strings

```
CR          equu    13
LF          equ     10
NULL    equ     0

str1:       dc.b    "hello, world!",CR,LF,NULL  ; 15 bytes, excl. null
str2:       dc.b    "another one",NULL          ; 11 bytes, excl. null
str3:       dc.b    'y','e','t',' '         ; 11 bytes, excl. null
dc.b        "another"
dc.b        NULL
```

Null Terminated Strings - How to Output

```
#The following code outputs str1:

                lea         str1,a0
write_string:   move.b      (a0)+,d0
                beq     done_write
                bsr     write_char
                bra     write_string
done_write:     …
```

Byte Counted Strings

A string is an array of bytes, where byte 0 is reserved for the string length count.

Note: string length is limited in range to 0–255.

E.g.

```
str1:       dc.b 15,"hello, world!",CR,LF
str2:       dc.b 11,'another one'
str3:       dc.b str3_end-str3-1            ; let assembler do the math!
dc.b        "yet another"
str3_end:
```

In this course we will only use null-terminated strings.