CS220 – Logic Design
AS09-Using Arrays

Outline

- Defining Arrays
- Accessing Array Elements
- Multidimensional Arrays
- The String (Array) Instructions
An array is stored in a contiguous block of memory. Each element of an array must be the same size (byte, word, etc). The address of any element can be computed as:

\[
\text{address} = (\text{address of first element}) + (\text{element index})* (\text{size of element})
\]

where the index of the first element is zero.
Arrays can be defined in the `.data` and `.bss` segments as follows:

```
.section .data
a1: .int 0, 1, 2, 3, 4  # 5 ints
a2: .fill 50,2, 0       # 50 shorts

.section .bss
a3: .fill 100, 4        # 100 ints
a4: .fill 20            # 20 bytes
```
Local Arrays

To define a local (on stack) array, subtract the proper number of bytes from **ESP**.

To reserve space for a character, two ints, and a 50 element short array we need $1 \times 1 + 2 \times 4 + 50 \times 2 = 109$ bytes. For efficiency, this should be rounded up to the nearest multiple of 4 ($112$) to keep ESP aligned on a 4 byte boundary. The **enter** instruction can be used to reserve space as follows:

`enter $112, 0   # Push BSP, adjust ESP`
Local Arrays

The variables can be arranged in a variety of ways. (The programmer must keep track!)

```
-1(%ebp)  char
          unused
-8(%ebp)  int1
-12(%ebp) int2
-112(%ebp) short array

short array
  -100(%ebp)  char
  -101(%ebp)  unused
  -108(%ebp)  int1
  -112(%ebp)  int2
```
Assuming that **arr1** labels the beginning of a byte array and **arr2** labels the beginning of an **int** array in either the **.data** or **.bss** segments, we can access elements using:

```assembly
movb arr1,%al      # AL  = arr1[0]
movb arr1+10,%al   # AL  = arr1[10]
movl arr2,%eax     # EAX = arr2[0]
movl arr2+4,%eax   # EAX = arr2[1]
movl arr2+24,%eax  # EAX = arr2[6]
```
In an instruction such as:

\[
\text{movb arr1+10,}\%al \quad \text{# AL = arr1[10]}
\]

it is \texttt{gas} that does the arithmetic \texttt{arr1+10} during assembly. The label \texttt{arr1} represents a number (the \texttt{offset} from the beginning of the \texttt{data} section to the first element in the array) and \texttt{arr1+10} is just another number. All operand arithmetic is performed by \texttt{gas} during assembly.
gas can do some pretty fancy stuff with labels and operand arithmetic:

```assembly
xarr: .int 1, 2, 3, 4, 5
xend:
 .section .text
# Find number of elements in array
movl $(xend-xarr)/4,%ecx
# Move third element into EAX
movl xarr+2*4,%eax
# Move address of third element to EAX
movl $(xarr+2*4),%eax
```
Similarly we can access the 50 element local short array at -100(%ebp) via:

```assembly
    movw -100(%ebp),%ax  # AX=arr[0]
    movw -98(%ebp),%ax   # AX=arr[1]
    movw -100+33*2(%ebp),%ax # AX=arr[33]
```

The last instruction is equivalent to:

```assembly
    movw -44(%ebp),%ax  # AX=arr[33]
```

Typically we use a variable array index and indexed or indirect addressing to access elements in an array.
For example, we could add the elements in the local 50 element short array using:

```assembly
movl   $0,%eax     # sum (EAX) = 0
movl   $0,%ecx     # i (ECX) = 0
beg_while:
    cmpl   $50,%ecx    # while (i<50)
    jge    end_while
    movw   -100(%ebp,%ecx,2),%dx
    movswl %dx,%edx    # sign extend
    addl   %edx,%eax   # sum += arr[i]
    incl   %ecx
    jmp    beg_while
end_while:
```

10
Similar methods can be used when working with arrays is the **data** and **bss** sections.

When working with arrays you must be careful with word sizes and beginning and ending array addresses. (Watch for off-by-one type errors at the ends of the array.)

Here is a function that returns the sum of all of the elements in an **int** array. The function takes an array pointer and the array length as arguments.
AS09-Using Arrays
Accessing Array Elements

_add_array:    # Uses only EAX, ECX, EDX
enter $0,$0        # no locals
movl 8(%ebp),%ecx  # ECX = &arr[0]
movl $0,%eax       # EAX(sum) = 0
movl $0,%edx       # EDX(i) = 0

beg_while:
cmpl 12(%ebp),%edx # while(i<n)
jge end_while
addl (%ecx,%edx,4),%eax
incl %edx          # i++
jmp  beg_while:

end_while:
leave                  # Return in EAX
ret
A two-dimensional array of the form \( \text{a}[3][2] \) is stored in memory in **rowwise** order: \( \text{a}[0][0], \text{a}[0][1], \text{a}[1][0], \text{a}[1][1], \text{a}[2][0], \text{and a}[2][1] \).

The offset address of element \( \text{a}[i][j] \) from the beginning of the array can be computed as:

\[
\text{offset} = \text{elsize} \times (i \times \text{ncols} + j)
\]

where \textbf{elsize} is the size in bytes of each element and \textbf{ncols} is the total number of columns in the array.
Note that the number of columns, as well as the row and column index, must be known to calculate the address.

Elements of higher dimensional arrays \( b[4][3][2] \) are accessed similarly. All dimensions except the first are required to compute the address.
Avoid using multidimensional arrays in C++ (see Stroustrup C.7.2). Use arrays of pointers to arrays (or vectors or lists) instead. (It may be necessary to use multidimensional arrays when using legacy Fortran code.)

To allocate space for a 2D array as an array of pointers:

```cpp
double **A = new double*[nrows];
for (int i=0; i<nrows; i++)
    A[i] = new double[ncols];
```
The 80x86 processors provide a set of instructions for working with arrays. (The instructions are known as the *string instructions*.)

They use the *index* registers (ESI and EDI) to address data and then automatically increment (or decrement) the register. The DF (direction flag) flag in the FLAGS register determines if an index register is incremented or decremented.
The **clid** instruction clears DF which causes the register to increment when a string instruction is executed. **std** sets DF in which case the register is decremented.

The **lodss** instruction (where S is either b, w, or l) loads either AL, AX, or EAX from address (%esi).

The **stoss** instruction copies either AL, AX, or EAX to address (%edi).
Here is a code fragment that copies the 10 element int array1 to array2:

```
cld                   # Inc. ESI & EDI
movl $array1, %esi   # ESI=&array1[0]
movl $array2, %edi   # EDI=&array2[0]
movl $10, %ecx       # Set loop count
lp:
    lodsl            # EAX=(%ESI), ESI+=4
    stosl            # (%EDI)=EAX, EDI+=4
loop lp
```
The `lodss/stosss` pair is so common there is a single `movss` that can be used instead. The previous fragment can be rewritten as:

```assembly
    cld               # Inc. ESI & EDI
    movl $array1,%esi # ESI=&array1[0]
    movl $array2,%edi # EDI=&array2[0]
    movl $10,%ecx     # Set loop count

lp:
    movsl             # (%EDI)=(%ESI)
    # EDI+=4, ESI+=4
    loop lp
```

# Inc. ESI & EDI
# ESI=&array1[0]
# EDI=&array2[0]
# Set loop count

# (%EDI)=(%ESI)
# EDI+=4, ESI+=4
The *rep* instruction *prefix* can be used with *movsS* to repeat the instruction ECX times. We can rewrite the fragment once again as:

```assembly
cld               # Inc. ESI & EDI
movl $array1, %esi # ESI=&array1[0]
movl $array2, %edi # EDI=&array2[0]
movl $10, %ecx    # Set loop count

rep movsl         # (%EDI)=(%ESI)
                 # EDI+=4, ESI+=4
```
The `cmpss` instruction compares data at locations (%ESI) and (%EDI). Flags are set just as for the CMP instruction. ESI and EDI are automatically incremented (or decremented depending on DF).

The `scasS` instructions are similar but compare data in AL/AX/EAX with (%EDI).
The **repe** (repeat if equal) and **repne** (repeat if not equal) instruction prefixes can be used with the **cmpss** and **scasS** instructions. The instruction is then repeated ECX times or until the first unequal (**repe**) or equal (**repne**) data are found. (**repz** and **repp** are synonyms.)

The following slide shows a code segment that is used to compare two memory blocks to see if they are **equal**.
# Compare two blocks of memory

cld

movl $blk1, %esi  # addr of 1st block
movl $blk2, %edi  # addr of 2nd block
movl size, %ecx  # size in bytes

repe cmpsb        # repeat while equal
je equal          # jmp if blocks equal

# code to perform if not equal

jmp onward

equal:

# code to perform if equal

onward: