CS 271
Computer Architecture &
Assembly Language

Lecture 15
2/22/22, Tuesday
Odds and Ends

• Clarifications
  • Avoid line-by-line comments
  • Post-condition: register changed + more...

• Final Project will be posted before Thursday’s lecture
  • Due Tuesday, March 15th 11:59 pm
  • More info later
Lecture Topics:

- Data-Related Operators
- Multi-Dimensional Arrays
- String Processing
- Lower-Level Programming
- How ReadInt Works
Data-Related Operators
Data-Related Operators

- OFFSET Operator
- PTR Operator
- TYPE Operator
- LENGTHOF Operator
- SIZEOF Operator
OFFSET Operator

- OFFSET returns the distance in bytes, of a label from the beginning of its enclosing segment

- The operating system adds the segment address (from the segment register)
OFFSET Examples

• Assume that the data segment begins at 00404000h:

```assembly
.data
bVal BYTE ?
wVal WORD ?
dVal DWORD ?
dVal2 DWORD ?
.code
...  
mov esi,OFFSET bVal    ; ESI = 00404000
mov esi,OFFSET wVal    ; ESI = 00404001
mov esi,OFFSET dVal    ; ESI = 00404003
mov esi,OFFSET dVal2   ; ESI = 00404007
```
PTR Operator

- Overrides the default type of a label (variable)
- Provides the flexibility to access part of a variable.

```
.data
myDouble DWORD 12345678h
.code
...
mov ax,myDouble        ; error - why?
mov ax,WORD PTR myDouble ; loads 5678h
mov WORD PTR myDouble,1357h ; saves 1357h
```
PTR Operator Examples

- Recall that **little endian** order is used when storing data in memory.
- In memory:

<table>
<thead>
<tr>
<th>78h</th>
<th>56h</th>
<th>34h</th>
<th>12h</th>
</tr>
</thead>
</table>

  ```
  mov al, BYTE PTR myDouble ; AL = 78h
  mov al, BYTE PTR [myDouble+1] ; AL = 56h
  mov al, BYTE PTR [myDouble+2] ; AL = 34h
  mov ax, WORD PTR myDouble ; AX = 5678h
  mov ax, WORD PTR [myDouble+2] ; AX = 1234h
  ```
• PTR can also be used to combine elements of a smaller data type and move them into a larger operand. The IA-32 CPU will automatically reverse the bytes.

```
data
myBytes BYTE 12h,34h,56h,78h
.code
...
mov ax,WORD PTR myBytes ; AX = 3412h
mov ax,WORD PTR [myBytes+2] ; AX = 7856h
mov eax,DWORD PTR myBytes ; EAX = 78563412h
```
TYPE Operator

- The *TYPE* operator returns the size, in bytes, of a single element of a data declaration.

```plaintext
.data
var1 BYTE ?
var2 WORD ?
var3 DWORD ?
var4 QWORD ?
.code
... 
mov eax,TYPE var1 ; 1
mov eax,TYPE var2 ; 2
mov eax,TYPE var3 ; 4
mov eax,TYPE var4 ; 8
```
LENETHOF Operator

• The LENGTHOF operator counts the number of elements in a single data declaration.

```
.data
byte1 BYTE 10,20,30 ; 3
list1 WORD 30 DUP(?) ; 30
list2 DWORD 30 DUP(?) ; 30
list3 DWORD 1,2,3,4 ; 4
digitStr BYTE "1234567",0 ; 8
.code
...
mov ecx,LENGTHOF list1 ; ecx contains 30
```
The `sizeof` operator returns a value that is equivalent to multiplying `lengthof` by `type`, i.e., size in bytes.

```assembly
.data
byte1 BYTE 10,20,30 ; 3
list1 WORD 30 DUP(?) ; 60
list2 DWORD 30 DUP(?) ; 120
list3 DWORD 1,2,3,4 ; 16
digitStr BYTE "1234567",0 ; 8
.code
...
mov ecx,SIZEOF list1 ; ecx contains 60
```
Spanning Multiple Lines

• A data declaration spans multiple lines if each line (except the last) ends with a comma.
• The `LENGTHOF` and `SIZEOF` operators include all lines belonging to the declaration:

```
.data
list    DWORD    10,20,
          30,40,
          50,60

.code
...
mov eax,LENGTHOF list    ; 6
mov ebx,SIZEOF list      ; 24
```
Spanning Multiple Lines

- In the following example, list identifies only the first DWORD declaration.
- Compare the values returned by `LENGTHOF` and `SIZEOF` here to those in the previous slide:

```
.data
list    DWORD    10,20
        DWORD    30,40
        DWORD    50,60

.code
...
    mov    eax,LENGTHOF list   ; 2
    mov    ebx,SIZEOF list      ; 8
```
Index Scaling

• You can scale an indirect or indexed operand to the offset of an array element. This is done by multiplying the index by the array’s TYPE.

```plaintext
.data
listB BYTE 1,2,3,4,5,6,7
listW WORD 8,9,10,11,12,13
listD DWORD 14,15,16,17,18,19,20,21
.code
...
mov esi,5
mov al,listB[esi*TYPE listB] ; al = 6
mov bx,listW[esi*TYPE listW] ; bx = 13
mov edx,listD[esi*TYPE listD] ; edx = 19
```
Pointers

• You can declare a **pointer variable** that contains the offset of another variable

```assembly
.data
list      DWORD       100 DUP(?)
ptr       DWORD       list
.code
...
    mov esi,ptr
    mov eax,[esi]    ; EAX = @ list
```

• The effect is the same as `mov esi, OFFSET list`

• Note: `[ptr]` is an invalid reference!! Why?
Pointers

• You can declare a **pointer variable** that contains the offset of another variable

```
.data
list     DWORD     100 DUP(?)
ptr      DWORD     list
.code
...
    mov esi,ptr
    mov eax,[esi]    ; EAX = @ list

; C/C++ version:
int list[100];
int* ptr = list;
```
Summing an Integer Array

- The following code calculates the sum of an array of 32-bit integers (register indirect mode).

```
.data
intList  DWORD 100h,200h,300h,400h
ptrD     DWORD intList
.code
...
  mov esi,ptrD                        ; address of intList
  mov ecx,LENGTHOF intList            ; loop counter
  mov eax,0                            ; init the accumulator
L1:
  add eax,[esi]                        ; add an integer
  add esi,TYPE intList                 ; point to next integer
  loop L1                              ; repeat until ECX = 0
```
Summing an Integer Array

• Alternate code (indexed mode)

```assembly
.data
intList DWORD 100h,200h,300h,400h
.code
    ... ; set up ecx
    mov esi,0
    mov eax,0 ; zero the accumulator
L1:
    add eax, intList[esi*TYPE intList]
    inc esi
    loop L1
```
Multi-Dimensional Arrays
String Processing
Two-Dimensional Array (Matrix)

• Example declaration:

   Matrix DWORD 5 DUP (3 DUP(?))
   ;15 elements

• A matrix is an array of arrays

• Row major order
  • Row index first (5 rows, 3 columns)
    • i.e., 5 rows, 3 elements per row

• Example HLL reference: Matrix[0][2]
  • Last element in first row ... etc.

• In assembly language, it’s just a set of contiguous memory locations
Two-Dimensional Array (Matrix)

• An element’s address is calculated as the base address plus an offset
  • BaseAddress + elementSize * [(row# * elementsPerRow) + column#]

• Example: Suppose Matrix is at address 20A0h

  • The address of Matrix[3][1] is
    \[20A0h + 4 \times [(3 \times 3) + 1] = 20C8h\]
Matrix Addresses (hexadecimal)

- Matrix elements are arranged in sequential addresses in row-major order

<table>
<thead>
<tr>
<th>Matrix</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20A0</td>
<td>20A4</td>
<td>20A8</td>
</tr>
<tr>
<td>1</td>
<td>20AC</td>
<td>20B0</td>
<td>20B4</td>
</tr>
<tr>
<td>2</td>
<td>20B8</td>
<td>20BC</td>
<td>20C0</td>
</tr>
<tr>
<td>3</td>
<td>20C4</td>
<td><strong>20C8</strong></td>
<td>20CC</td>
</tr>
<tr>
<td>4</td>
<td>20D0</td>
<td>20D4</td>
<td>20D8</td>
</tr>
</tbody>
</table>
Higher Dimensions

• A 3-dimensional array is an array of matrices
• A 4-dimensional array is an array of 3-dimensional arrays
• ... etc., no theoretical limit
  • Practically and readability rule
• Address calculations can be extrapolated from matrix address calculations
• Contiguous memory in “highest-dimension” major order
String Primitives

- A string is an array of BYTE
- In most cases, an extra byte is needed for the zero-byte terminator
- MASM has some “string primitives” for manipulating strings byte-by-byte
  - Most important are:
    - `lodsb` ; load string byte
    - `stosb` ; store string byte
    - `cld` ; clear direction flag
    - `std` ; set direction flag
- There are many others
  - Explore on your own
lodsb and stosb

- **lodsb**
  - Moves byte at [esi] into the AL register
  - Increments esi if direction flag is 0
  - Decrements esi if direction flag is 1

- **stosb**
  - Moves byte in the AL register to memory at [edi]
  - Increments edi if direction flag is 0
  - Decrements edi if direction flag is 1
cld and std

• **cld**
  - Sets direction flag to 0
  - Causes *esi* and *edi* to be incremented by *lodsb* and *stosb*
  - Used for moving “forward” through an array

• **std**
  - Sets direction flag to 1
  - Causes *esi* and *edi* to be decremented by *lodsb* and *stosb*
  - Used for moving “backward” through an array
Demo

• Shows capitalizing and reversing a string
Lower-Level Programming
How ReadInt Works
Lower-Level Programming

- **All** keyboard input is *character*
  - Digits are character codes 48-57
  - ‘0’ is character number 48
  - ‘1’ is 49 ... ‘9’ is 57

- Cannot do arithmetic with string representations

- What does *ReadInt* do? (Irvine’s library)
  - Gets a string of digits (characters)
  - Converts digits to numeric values

- How does *ReadInt* do it?
ReadInt Algorithm (pseudo-code)

get str
x = 0
for k = 0 to (len(str)-1)
    if 48 <= str[k] <= 57
        x = 10 * x + (str[k] – 48)
    else
        break