CPSC-240 Computer Organization and Assembly Language

Chapter 8

Addressing Modes

Instructor: Yitsen Ku, Ph.D.
Department of Computer Science,
California State University, Fullerton, USA

CALIFORNIA STATE UNIVERSITY FULLERTON™

Outline

- Address and Values
- Register Mode Addressing
- Immediate Mode Addressing
- Memory Mode Addressing
- Example Program, List Summation
- Example Program, Pyramid Areas and Volumes

Addressing Modes

Addressing Modes

- The addressing modes are the supported methods for accessing a value in memory using the address of a data item being accessed (read or written). This might include the name of a variable or the location in an array.
- The basic addressing modes are:
  - Register
  - Immediate
  - Memory

Address and Values

CALIFORNIA STATE UNIVERSITY FULLERTON

Address and Values

- On a 64-bit architecture, addresses require 64-bits.
- As noted in the previous chapter, the only way to access memory is with the brackets(**[]**'s). Omitting the brackets will not access memory and instead obtain the address of the item. For example:

```
mov rax, qword [var1]     ; value of var1 in rax
mov rax, var1             ; address of var1 in rax
```

Address and Values

- When accessing memory, in many cases the operand size is clear. For example, the following instruction will move a double-word from memory.

```
mov eax, [rbx]
```

- However, for some instructions the size can be ambiguous. For example, the following instruction is ambiguous since it is not clear if the memory being accessed is a byte, word, or double-word.

```
inc [rbx]                ; error
```

Address and Values

- In such a case, operand size must be specified with either the *byte*, *word*, or *dword*, *qword* size qualifier. For example, each instruction requires the size specification in order to be clear and legal.

```
inc byte [rbx]
inc word [rbx]
inc dword [rbx]
```

Register Mode Addressing

Register Mode Addressing

- Register mode addressing means that the operand is a CPU register (**eax**, **ebx**, etc.). For example:

  mov eax, ebx

- Both **eax** and **ebx** are in register mode addressing.

Immediate Mode Addressing

Immediate Mode Addressing

- Immediate mode addressing means that the operand is an immediate value. For example:

  mov eax, 123

- The destination operand, **eax** , is register mode addressing.
- The **123** is immediate mode addressing. It should be clear that the destination operand in this example cannot be immediate mode.

Memory Mode Addressing

Memory Mode Addressing

- Memory mode addressing means that the operand is a location in memory (accessed via an address). This is referred to as *indirection* or *dereferencing*.
- The most basic form of memory mode addressing has been used extensively in the previous chapter. Specifically, the following instruction will access the memory location of the variable ***qNum*** and retrieve the value stored there.

```
mov rax, qword [qNum]
```

Memory Mode Addressing

```
qNum        qword       65
.
Mov  rax, qword [qNum]       ; retrieve value of qNum
```

rax $\boxed{\texttt{0x00000000 00000065}}$

```
qNum        qword       65
.
Mov  rax, qNum          ; get qNum address
```

rax $\boxed{\texttt{0x00000000 006000e0}}$

	value	address
	xx	0x6000ec
	xx	0x6000eb
	xx	0x6000ea
	xx	0x6000e9
	xx	0x6000e8
	xx	0x6000e7
	xx	0x6000e6
	xx	0x6000e5
	xx	0x6000e4
	0x00	0x6000e3
	0x00	0x6000e2
	0x00	0x6000e1
qNum	0x65	0x6000e0

Memory Mode Addressing

- When accessing arrays, a more generalized method is required. Specifically, an address can be placed in a register and indirection performed using the register (instead of the variable name).
- For example, assuming the following declaration:

```
lst     dd     101, 103, 105, 107
```

Value	Address	Offset	Index
00	0x6000ef	lst + 15	
00	0x6000ee	lst + 14	
00	0x6000ed	lst + 13	
6b	0x6000ec	lst + 12	**lst[3]**
00	0x6000eb	lst + 11	
00	0x6000ea	lst + 10	
00	0x6000e9	lst + 9	
69	0x6000e8	lst + 8	**lst[2]**
00	0x6000e7	lst + 7	
00	0x6000e6	lst + 6	
00	0x6000e5	lst + 5	
67	0x6000e4	lst + 4	**lst[1]**
00	0x6000e3	lst + 3	
00	0x6000e2	lst + 2	
00	0x6000e1	lst + 1	
65	0x6000e0	lst + 0	**lst[0]**

lst →

Indirect Address Mode

- The first element of the array could be accessed as follows:

```
mov   eax, dword [lst]      ; eax = 0x00000065
```

- Another way to access the first element is as follows:

```
mov   rbx, lst              ; rbx = 0x00000000 006000e0
mov   eax, dword [rbx]      ; eax = 0x00000065
```

Base Address Mode

- The first element of the array could be accessed as follows:

```
mov   eax, dword [lst]     ; eax = 0x00000065
```

- Another way to access the first element is as follows:

```
mov   rbx, lst             ; rbx = 0x00000000 006000e0
mov   eax, dword [rbx]     ; eax = 0x00000065
```

Base+Index Address Mode

- There are several ways to access the array elements. One is to use a base address and add a displacement. For example, given the initializations:

```
mov   rbx, lst        ; rbx = 0x00000000 006000e0
mov   rsi, 8          ; rsi = 0x00000000 00000008
```

- Each of the following instructions access the third element (105 in the above list).

```
mov   eax, dword [lst+8]     ; eax = 0x00000069
mov   eax, dword [rbx+8]     ; eax = 0x00000069
mov   eax, dword [rbx+rsi]   ; eax = 0x00000069
```

General Format of Memory Addressing

- The general format of memory addressing is as follows:

  [baseAddr + (indexReg * scaleValue) + displacement]

- Where ***baseAddr*** is a register or a variable name. The ***indexReg*** must be a register.
- The ***scaleValue*** is an immediate value of 1, 2, 4, 8 (1 is legal, but not useful).
- The ***displacement*** must be an immediate value. The total represents a 64-bit address.

General Format Address Mode

- Some example of memory addressing for the source operand are as follows:

```
mov   eax, dword [var1]
mov   rax, qword [rbx+rsi]
mov   ax, word [lst+4]
mov   bx, word [lst+rdx+2]
mov   rcx, qword [lst+(rsi*8)]
mov   al, byte [buff-1+rcx]
mov   eax, dword [rbx+(rsi*4)+16]
```

General Format Address Mode

- For example, to access the 3rd element of the previously defined double-word array (which is index 2 since index's start at 0):

```
mov   rsi, 2                        ; index=2
mov   eax, dword [lst+rsi*4]        ; get lst[2]
```

Example Program

List Summation

Example Program, List Summation

```
; Simple example to the sum and average for
; a list of numbers.
; *****************************************************
; Data declarations
section         .data
; -----
; Define constants
EXIT_SUCCESS    equu 0                  ; successful operation
SYS_exit        equ 60                  ; call code for terminate
; -----
; Define Data.
section         .data
        lst     dd      1002, 1004, 1006, 1008, 10010
        len     dd      5
        sum     dd      0
```

Example Program, List Summation

```
; ******************************************************
section         .text
global _start
_start:
; -----
; Summation loop.
        mov     ecx, dword [len]        ; get length value
        mov     rsi, 0                  ; index=0
sumLoop:
        mov     eax, dword [lst+(rsi*4)]        ; get lst[rsi]
        add     dword [sum], eax        ; update sum
        inc     rsi                     ; next item
        loop    sumLoop
; -----
; Done, terminate program.
last:
        mov     rax, SYS_exit           ; call code for exit
        mov     rdi, EXIT_SUCCESS       ;exit with success
        syscall
```

Lab Activity

Lab Activity

Given the following variable declarations and code fragment:

```
        list    dd      2, 7, 4, 5, 6, 3
        mov     rbx, list
        mov     rsi, 1
        mov     rcx, 2
        mov     eax, 0
        mov     edx, dword [rbx+4]
lp:     add     eax, dword [rbx+rsi*4]
        add     rsi, 2
        loop    lp
        imul    dword [rbx]
```

What would be in the **eax**, **edx**, **rcx**, and **rsi** registers after execution? Show answer in hex, full register size. *Note*, pay close attention to the register sizes (32-bit vs. 64-bit).

Homework

Homework

Create a program to sort a list of numbers. Use the following bubble sort algorithm:

```
for ( i = (len-1) to 0 ) {
        swapped = false
for ( j = 0 to i-1 )
        if ( lst(j) > lst(j+1) ) {
                tmp = lst(j)
                lst(j) = lst(j+1)
                lst(j+1) = tmp
                swapped = true
        }
        if ( swapped = false ) exit
        }
```

Use the debugger to execute the program and display the final results. Create a debugger input file to show the results.

Thanks