**Q1 Sec 7.9.1 Do problems 1, 2**

**1.**

**In the following code sequence, show the value of AL after each shift or rotate instruction has executed:**

```
mov al,0D4h
shr al,1                    ; a.
mov al,0D4h
sar al,1                    ; b.
mov al,0D4h
sar al,4                    ; c.
mov al,0D4h
rol al,1                    ; d.
```

Answer:

A. 01101010b
B. 11101010b
C. 11111101b
D. 10101001b

*Work shown below:

Sec 2.9.1

1. Show the value of AL after each shift or rotate instruction

```
mov al, 0D4h
shr al, 1
```

a)

(13×16)+(4×16⁰) = 212

| Dividendo | Quotient | Remainder |
|---|---|---|
| 212/2 | 106 | 0 |
| 106/2 | 53 | 0 |
| 53/2 | 26 | 1 |
| 26/2 | 13 | 0 |
| 13/2 | 6 | 1 |
| 6/2 | 3 | 0 |
| 3/2 | 1 | 1 |
| 1/2 | 0 | 1 |

→ 11010100b

11010100 → shr 1 → 01101010b (a)

```
mov al, 0D4h
sar al, 1
```

b)

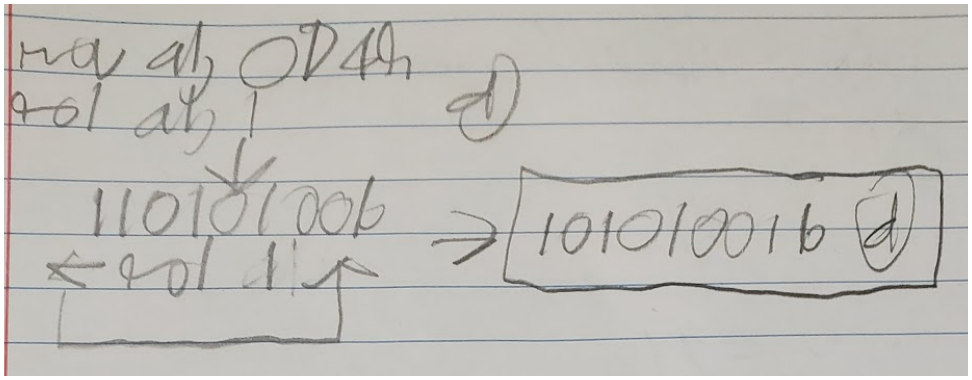
11010100b → sar 1 → 11101010b (b)

```
mov al, 0D4h
sar al, 4
```

c)

11010100b → sar 4 → 11111101b (c)

cont'd




*Answer from answer key:

0D4h = 1101 0100

a) 1101 0100 → SHR al, 1 → 0110 1010 = 6Ah

b) 1101 0100 → SAR al, 1→1110 1010 = 0EAh

c) 1101 0100 → SAR al, 4 →1111 1101 = 0FDh

d) 1101 0100 → ROL al, 1 → 1010 1001 = 0A9h

**2.**

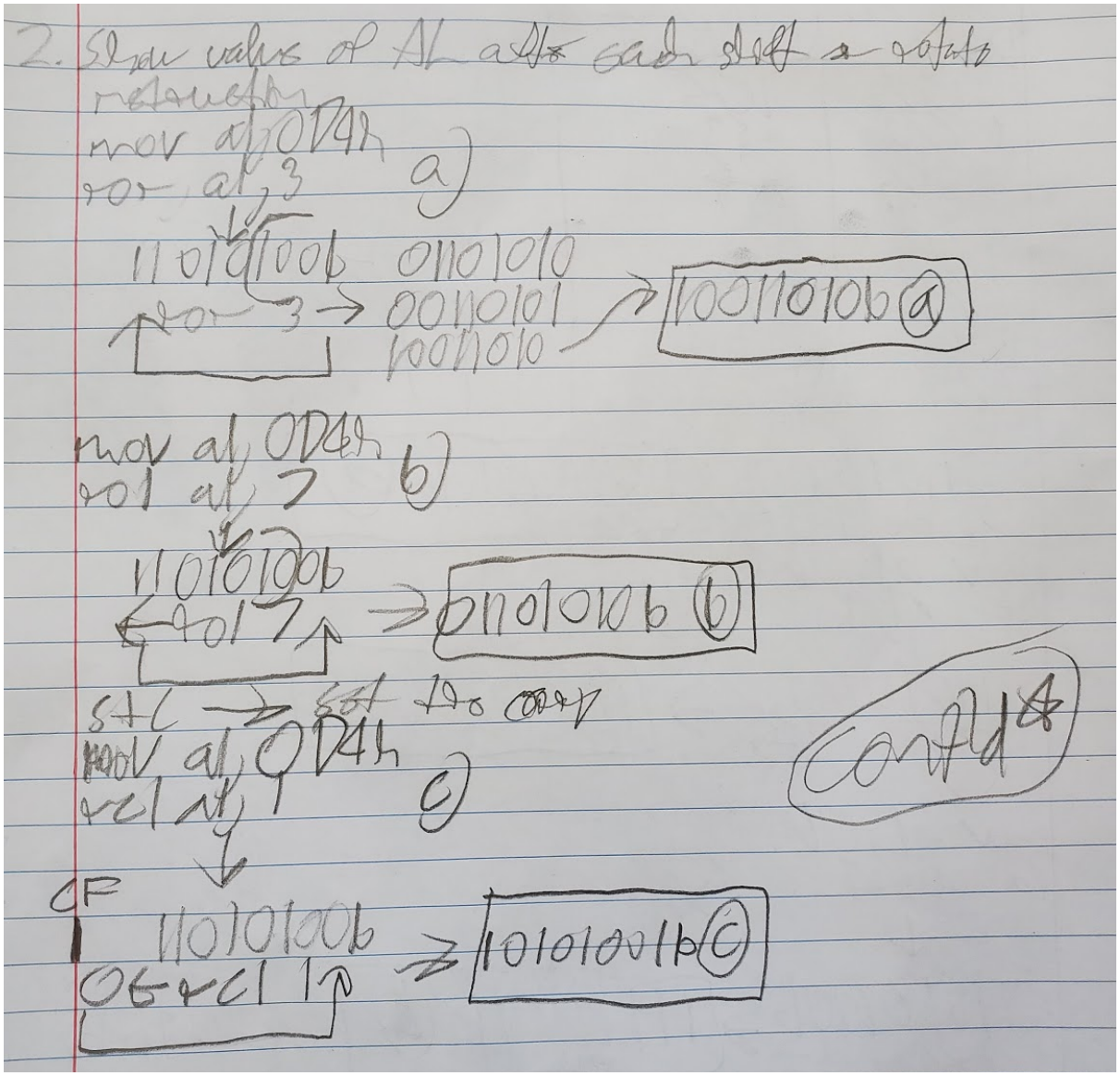
**In the following code sequence, show the value of AL after each shift or rotate instruction has executed:**

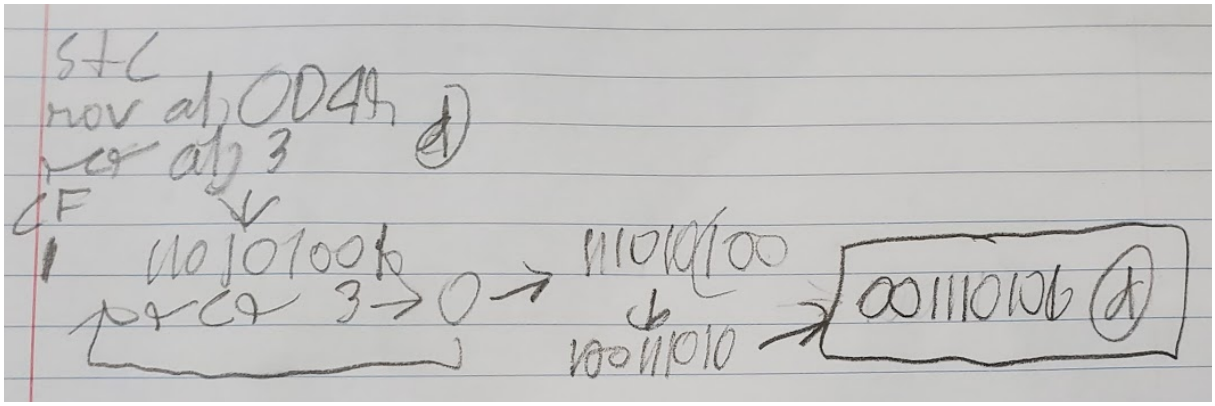
```
mov al,0D4h
ror al,3                    ; a.
mov al,0D4h
rol al,7                    ; b.
stc
mov al,0D4h
rcl al,1                    ; c.
stc
mov al,0D4h
rcr al,3                    ; d.
```

Answer:

A. 10011010b
B. 01101010b
C. 10101001b
D. 00111010b

*Work shown below:




*Answer from answer key:

0D4h = 1101 0100

a) 1101 0100 → ROR al, 3 → 1001 1010 = 9Ah

b) 1101 0100 → ROL al, 7→0110 1010 = 6Ah
c) First STC is executed so carry flag is 1 initially. 1101 0100 → RCL al, 1 →1010 1001 = 0A9h
d) First STC is executed so carry flag is 1 initially. 1101 0100 → RCR al, 3 → 0011 1010 = 3Ah

**Q2 Sec 7.9.2 Do problems 3, 4, 5, 11**

**3.**

**Write a logical shift instruction that multiplies the contents of EAX by 16.**

Answer:

shl EAX,4

*An excerpt below just for reference.

*Bitwise Multiplication* *Bitwise multiplication* is performed when you shift a number's bits in a leftward direction (toward the MSB). For example, SHL can perform multiplication by powers of 2. Shifting any operand left by $n$ bits multiplies the operand by $2^n$. For example, shifting the integer 5 left by 1 bit yields the product of $5 \times 2^1 = 10$:

```
mov  dl,5      Before: 0 0 0 0 0 1 0 1 = 5
shl  dl,1      After:  0 0 0 0 1 0 1 0 = 10
```

If binary 00001010 (decimal 10) is shifted left by two bits, the result is the same as multiplying 10 by $2^2$:

```
mov  dl,10                ; before:  00001010
shl  dl,2                 ; after:   00101000
```

**4.**

**Write a logical shift instruction that divides EBX by 4.**

shr EBX,2

*An excerpt below just for reference.

*Bitwise Division* *Bitwise division* is accomplished when you shift a number's bits in a rightward direction (toward the LSB). Shifting an unsigned integer right by $n$ bits divides the operand by $2^n$. In the following statements, we divide 32 by $2^1$, producing 16:

```
mov  dl,32     Before: 0 0 1 0 0 0 0 0 = 32
shr  dl,1      After:  0 0 0 1 0 0 0 0 = 16
```

In the following example, 64 is divided by $2^3$:

```
mov  al,01000000b         ; AL = 64
shr  al,3                 ; divide by 8, AL = 00001000b
```

Division of signed numbers by shifting is accomplished using the SAR instruction because it preserves the number's sign bit.

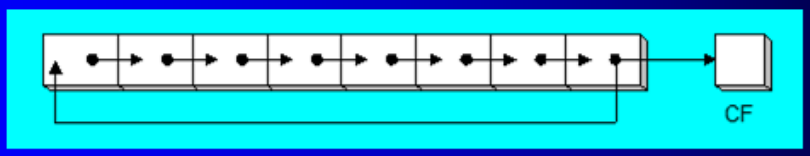
**5.**

**Write a single rotate instruction that exchanges the high and low halves of the DL register.**

Answer:
ror DL,4 *From answer key
or
rol DL,4
*Some excerpts below just for reference.

## ROR Instruction

- ROR (rotate right) shifts each bit to the right
- The lowest bit is copied into both the Carry flag and into the highest bit
- No bits are lost

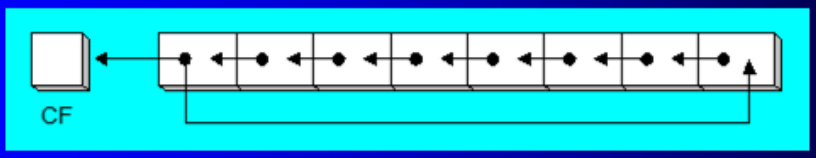



```
mov al,11110000b
ror al,1                    ; AL = 01111000b

mov dl,3Fh
ror dl,4                    ; DL = F3h
```

## ROL Instruction

- ROL (rotate) shifts each bit to the left
- The highest bit is copied into both the Carry flag and into the lowest bit
- No bits are lost




```
mov al,11110000b
rol al,1                    ; AL = 11100001b

mov dl,3Fh
rol dl,4                    ; DL = F3h
```

Irvine, Kip R. Assembly Language for x86 Processors 7/e, 2015.

11

**11.**

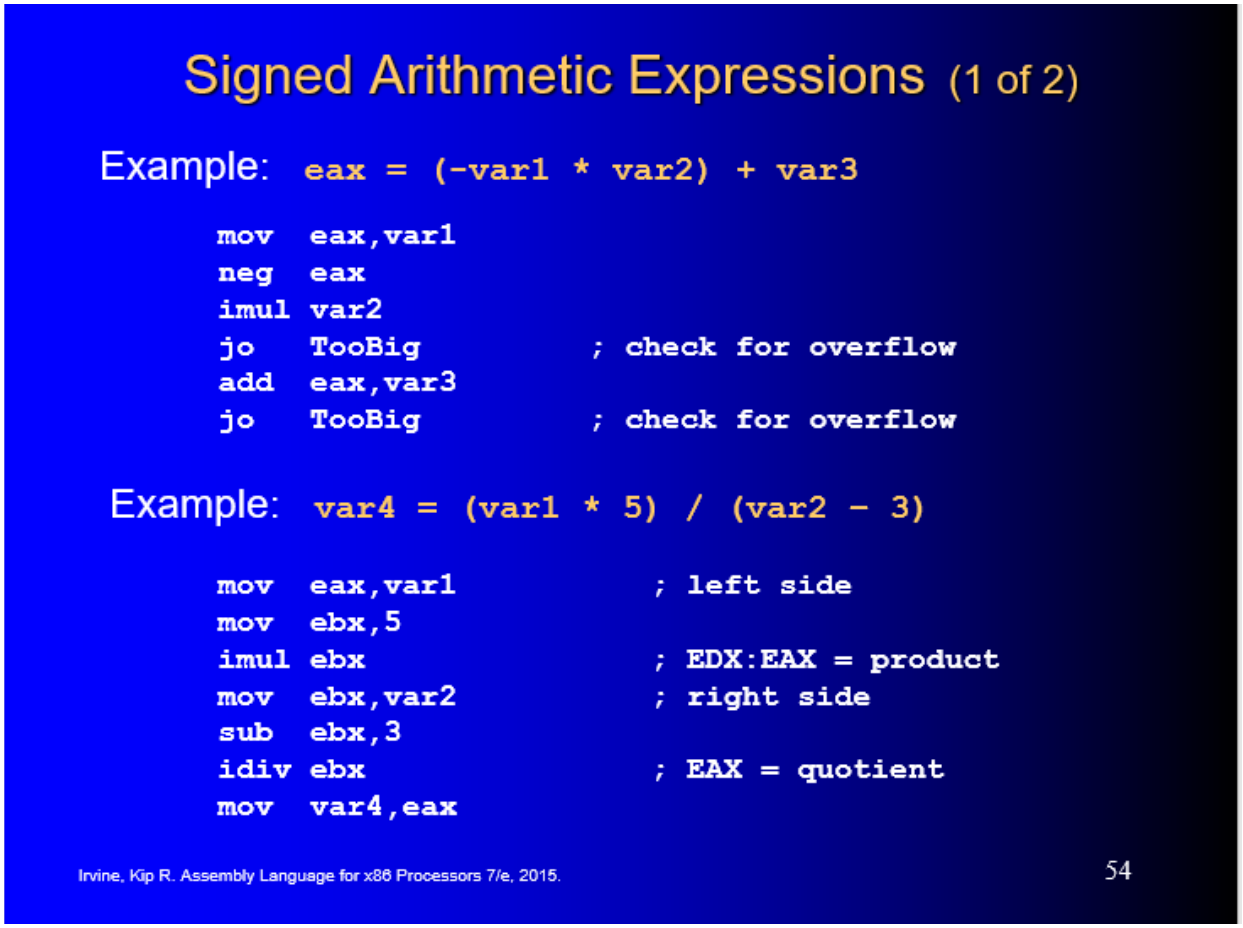
**Implement the following C++ expression in assembly language, using 32-bit unsigned operands:**

```
val1 = (val2 * val3) / (val4 - 3)
```

Answer:

mov eax,val2
mov ebx,val3
imul ebx
mov ebx,val4
sub ebx,3
idiv ebx
mov val1,eax

*An excerpt below just for reference.




*Answer from answer key:

MOV eax, val2
MUL val3
MOV ebx, val4
SUB ebx, 3
DIV, ebx
MOV val1, eax

**Q3. Sec 8.10.1 Do problems 3;**

**3.**

**How does a program using the STDCALL calling convention clean up the stack after a procedure call?**

Answer:

We supply an integer parameter to the RET instruction, which in turn adds 8 to ESP after returning to the calling procedure. The integer must equal the number of bytes of stack space consumed by the procedure's parameters. It should be pointed out that STDCALL, like C, pushes arguments onto the stack in reverse order. By having a parameter in the RET instruction,

STDCALL reduces the amount of code generated for subroutine calls (by one instruction) and ensures that calling programs will never forget to clean up the stack.

*Answer from answer key:

A number on the RET instruction corresponds to the number of bytes added to the stack pointer to clear parameters from the stack. This happens after returning to the calling procedure.

**Q4 Sec 8.10.2 Do Problems 1,2**

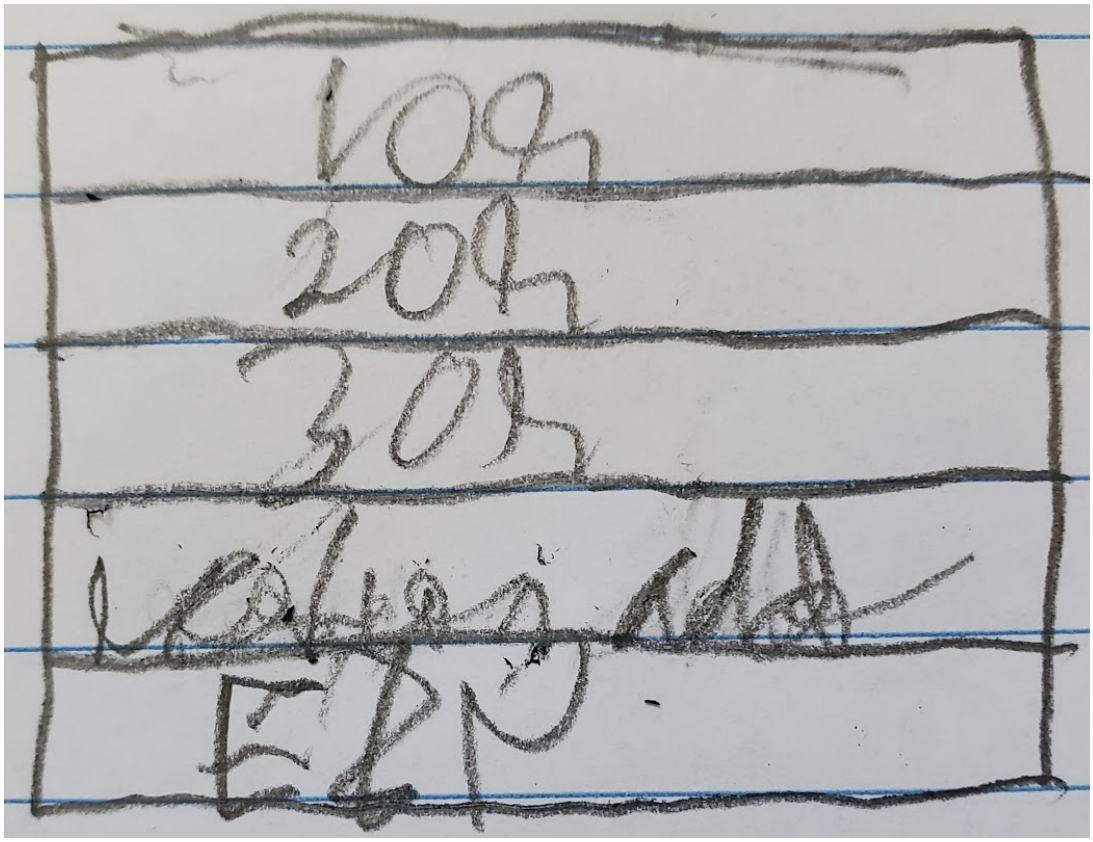
**1.**

**Here is a calling sequence for a procedure named AddThree that adds three doublewords (assume that the STDCALL calling convention is used):**

```
push 10h
push 20h
push 30h
call AddThree
```

**Draw a picture of the procedure's stack frame immediately after EBP has been pushed on the runtime stack.**

*Work shown below:




*An excerpt below just for reference.

### 8.2.3 Accessing Stack Parameters

High-level languages have various ways of initializing and accessing parameters during function calls. We will use the C and C++ languages as an example. They begin with a *prologue* consisting of statements that save the EBP register and point EBP to the top of the stack. Optionally, they may push certain registers on the stack whose values will be restored when the function returns. The end of the function consists of an *epilogue* in which the EBP register is restored and the RET instruction returns to the caller.

*AddTwo Example* The following **AddTwo** function, written in C, receives two integers passed by value and returns their sum:

```
int AddTwo( int x, int y )
{
     return x + y;
}
```

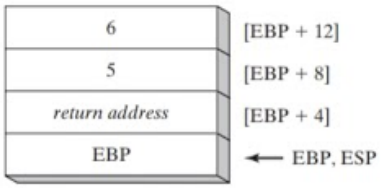
Let's create an equivalent implementation in assembly language. In its prologue, **AddTwo** pushes EBP on the stack to preserve its existing value:

```
AddTwo PROC
     push  ebp
```

Next, EBP is set to the same value as ESP, so EBP can be the base pointer for AddTwo's stack frame:

```
AddTwo PROC
     push  ebp
     mov   ebp,esp
```

After the two instructions execute, the following figure shows the contents of the stack frame. A function call such as AddTwo(5, 6) would cause the second parameter to be pushed on the stack, followed by the first parameter:




AddTwo could push additional registers on the stack without altering the offsets of the stack parameters from EBP. ESP would change value, but EBP would not.

*Answer from answer key:

ESP points to EBP and each row above corresponds to +4, +8 … bytes so the return address is EBP + 4 and 10h is EBP + 16

**2.**

**Create a procedure named AddThree that receives three integer parameters and calculates and returns their sum in the EAX register.**

Answer:

```
AddThree PROC
        push EBP
        mov EBP,ESP
        mov EAX,[EBP + 16]
        add EAX,[EBP + 12]
```

```
        add EAX,[EBP + 8]
        pop EBP
        Ret 12
AddThree ENDP
```

*Answer from answer key:

```
PROC AddThree
PUSH ebp
MOV ebp, esp
MOV eax, [ebp + 8]
ADD eax, [ebp + 12]
ADD eax, [ebp + 16]
POP ebp
RET 12
AddThree ENDP
```

**Q5. Sec 9.9.1 Do Problems 1, 2, 12**

**1.**

**Which Direction flag setting causes index registers to move backward through memory when executing string primitives?**

Answer:

STD *(value = 1)

*An excerpt below just for reference.

*Direction Flag* String primitive instructions increment or decrement ESI and EDI based on the state of the Direction flag (see Table 9-2). The Direction flag can be explicitly modified using the CLD and STD instructions:

```
CLD       ; clear Direction flag (forward direction)
STD       ; set Direction flag (reverse direction)
```

Forgetting to set the Direction flag before a string primitive instruction can be a major headache, since the ESI and EDI registers may not increment or decrement as intended.

Table 9-2 Direction Flag Usage in String Primitive Instructions.

| Value of the Direction Flag | Effect on ESI and EDI | Address Sequence |
|---|---|---|
| Clear | Incremented | Low-high |
| Set | Decremented | High-low |

**2.**

**When a repeat prefix is used with STOSW, what value is added to or subtracted from the index register?**

Answer:

2 (STOSW is for AX, which 16 bits, so it is a word or 2 bytes)

*An excerpt below just for reference.

### 9.2.4 STOSB, STOSW, and STOSD

The STOSB, STOSW, and STOSD instructions store the contents of AL/AX/EAX, respectively, in memory at the offset pointed to by EDI. EDI is incremented or decremented based on the state of the Direction flag. When used with the REP prefix, these instructions are useful for filling all elements of a string or array with a single value. For example, the following code initializes each byte in **string1** to 0FFh:

```
.data
Count = 100
string1 BYTE Count DUP(?)
.code
mov   al,0FFh                  ; value to be stored
mov   edi,OFFSET string1       ; EDI points to target
mov   ecx,Count                ; character count
cld                            ; direction = forward
rep   stosb                    ; fill with contents of AL
```

**12.**

**In the FillArray procedure from the Binary Search example in Section 9.5, why must the Direction flag be cleared by the CLD instruction?**

Answer:

We clear the direction flag to increment ESI and EDI. This is to go from lower range to upper range integers when we fill an array.

*An excerpt below just for reference.

*FillArray* Following is a listing of the module containing the FillArray procedure:

```
; FillArray Procedure                    (FillArray.asm)

INCLUDE Irvine32.inc

.code
;-----------------------------------------------------------
FillArray PROC USES eax edi ecx edx,
     pArray:PTR DWORD,          ; pointer to array
     Count:DWORD,               ; number of elements
     LowerRange:SDWORD,         ; lower range
     UpperRange:SDWORD          ; upper range
```

```
;
; Fills an array with a random sequence of 32-bit signed
; integers between LowerRange and (UpperRange - 1).
; Returns: nothing
;-----------------------------------------------------------
     mov   edi,pArray           ; EDI points to the array
     mov   ecx,Count            ; loop counter
     mov   edx,UpperRange
     sub   edx,LowerRange       ; EDX = absolute range (0..n)
     cld                        ; clear direction flag
L1:  mov   eax,edx              ; get absolute range
     call  RandomRange
     add   eax,LowerRange       ; bias the result
     stosd                      ; store EAX into [edi]
     loop  L1

     ret
FillArray ENDP
END
```

*Answer from answer key:

We are starting from index zero and want to move forward in the array with the STOSD instruction so we clear the direction register to make sure we move forward if it happened to be set before.

**Q6. Sec 9.9.2 Do Problems 1, 2, 4**

**1.**

**Show an example of a base-index operand in 32-bit mode.**

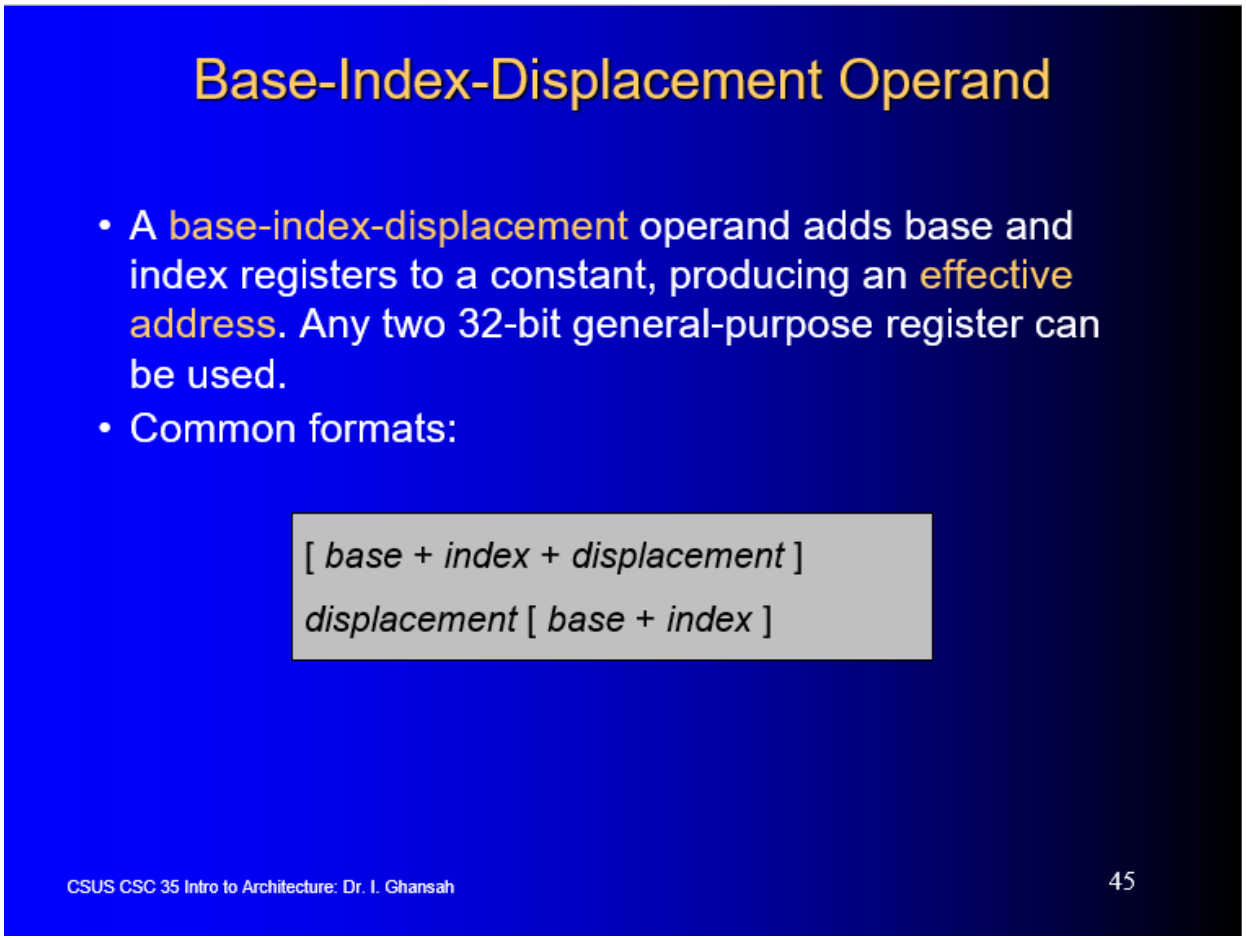
Answer:

[ebx + esi]

*Some excerpts below just for reference.

# Base-Index Operand

- A base-index operand adds the values of two registers (called base and index), producing an effective address. Any two 32-bit general-purpose registers may be used. *(Note: esp is not a general-purpose register)*
  - In 64-bit mode, you use 64-bit registers for bases and indexes
- Base-index operands are great for accessing arrays of structures. (A structure groups together data under a single name. )




**2.**

**Show an example of a base-index-displacement operand in 32-bit mode.**

Answer:

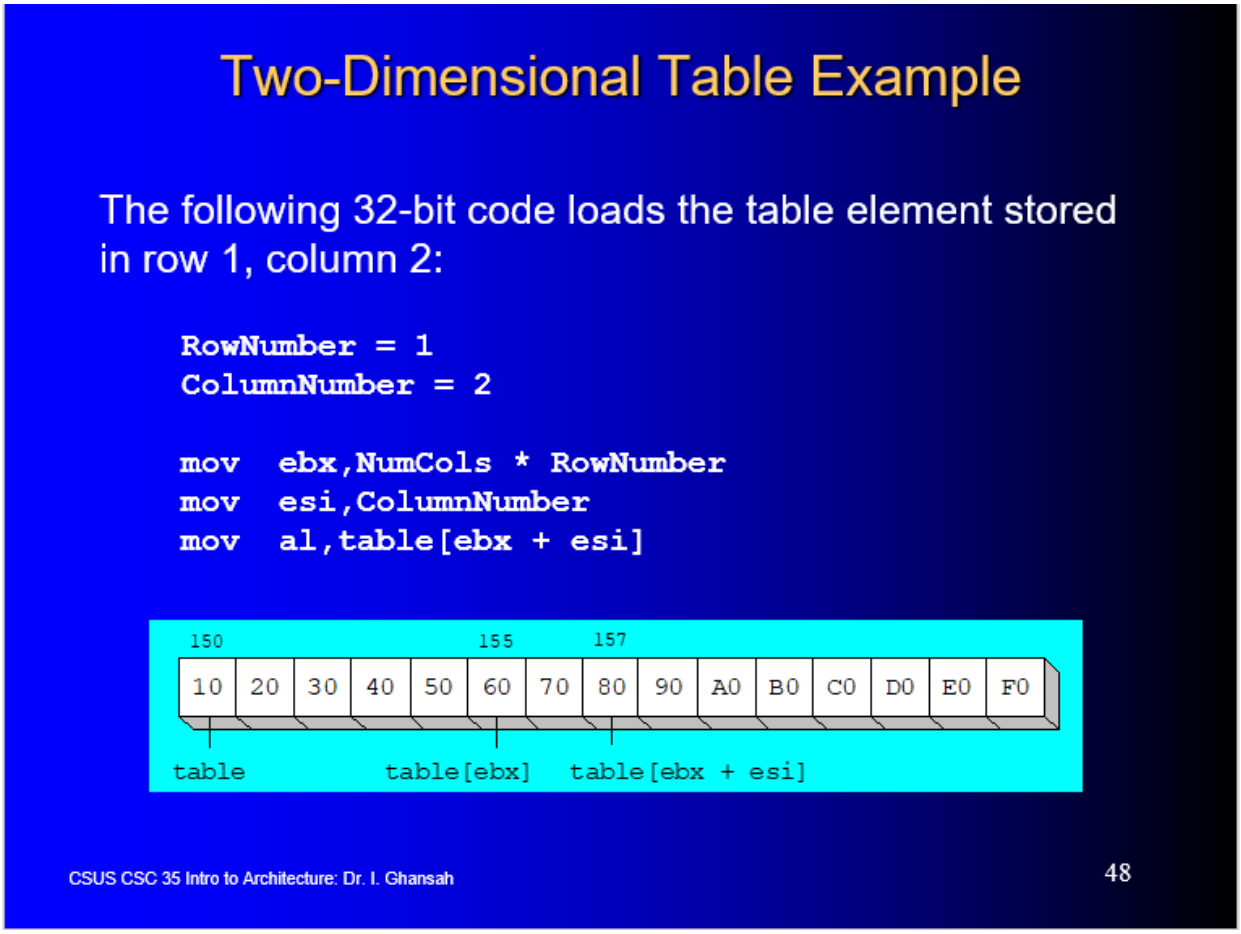
val[ebx + esi]

*Some excerpts below just for reference.

# Base-Index-Displacement Operand

- A base-index-displacement operand adds base and index registers to a constant, producing an effective address. Any two 32-bit general-purpose register can be used.
- Common formats:

| |
|---|
| [ *base* + *index* + *displacement* ] |
| *displacement* [ *base* + *index* ] |




*Answer from answer key:

val1[ebx + esi]; where val1 is some variable

**4.**

**Write instructions using CMPSW that compare two arrays of 16-bit values named sourcew and targetw.**

Answer:

mov ECX, LENGTHOF sourcew

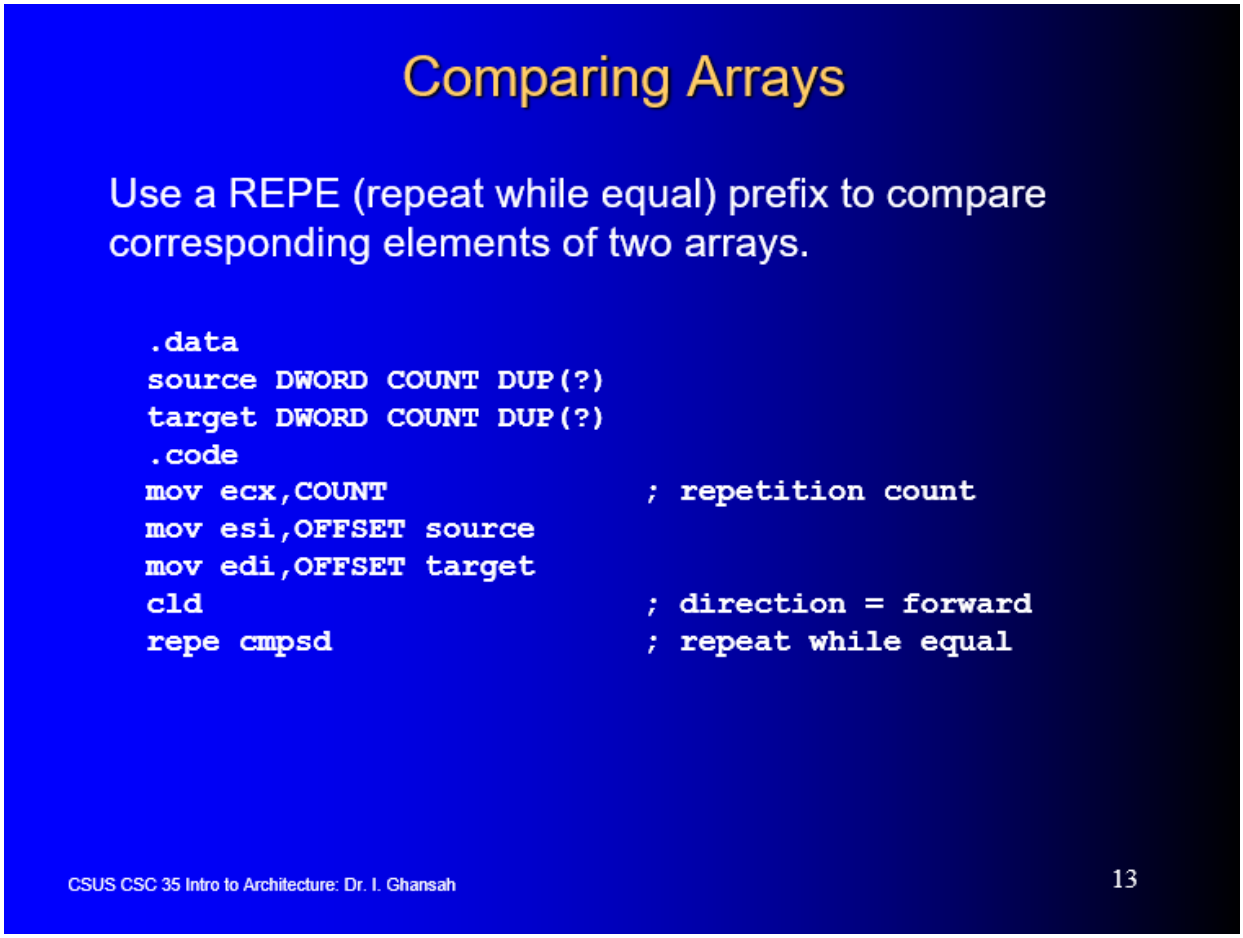
mov ESI, OFFSET sourcew

mov EDI, OFFSET targetw

CLD

CMPSW

*An excerpt below just for reference.




*Answer from answer key:

MOV esi, OFFSET sourcew
MOV edi, OFFSET targetw
MOV ecx, LENGTHOF sourcew; assume sourcew and targetw are the same length
CLD
repe CMPSW