Assembly language programs use conditional instructions to implement high-level statements such as IF statements and loops. Each of the conditional statements involves a possible transfer of control (jump) to a different memory.
address. A transfer of control, or branch, is a way of altering the order in which statements are
executed. There are two basic types of transfers:

- **Unconditional Transfer:** Control is transferred to a new location in all cases; a new address
is loaded into the instruction pointer, causing execution to continue at the new address. The
JMP instruction does this.
- **Conditional Transfer:** The program branches if a certain condition is true. A wide variety of conditional transfer instructions can be combined to create conditional logic structures.

The CPU interprets true/false conditions based on the contents of the ECX and Flags registers.

In the following example, we add 1 to AX each time the loop repeats. When the loop ends, AX = 5 and ECX = 0:

```
        mov  ax,0
        mov  ecx,5
L1:
        inc  ax
        loop L1
```

A common programming error is to inadvertently initialize ECX to zero before beginning a loop. If this happens, the LOOP instruction decrements ECX to FFFFFFFFh, and the loop repeats 4,294,967,296 times! If CX is the loop counter (in real-address mode), it repeats 65,536 times.

Rarely should you explicitly modify ECX inside a loop. If you do, the LOOP instruction may not work as expected. In the following example, ECX is incremented within the loop. It never reaches zero, so the loop never stops:

```
top:
        .
        .
        inc  ecx
        loop top
```

(a) what is the largest possible backward jump? -128

(b) what is the largest possible forward jump? +127

Q) Use a loop to add the numbers in an array.

```
INCLUDE Irvine32.inc
.data
arr1 WORD 5, 10, 15
result WORD 0
.code
main PROC
xor ax, ax
mov esi, OFFSET arr1
mov cx, 3
    L1:
        add ax, [esi]
        add esi, 2
        sub cx, 1
        jnz L1

mov [result], ax
exit
main ENDP
END main
```

Jnz it jumps back to execute the loop when the Zero flag is not set.
When Zero flag is set to 1 loop ends and execute the next instruction which is mov [result], ax
Zero flag is set when cx=0

C code:

```
For (int i=3i>0;i++)
{
Result += arr[i];
}
```

Q) Use a loop to add the numbers in an array. Use cmp instruction.

```
INCLUDE Irvine32.inc
.data
arr1 WORD 5h, 10h
result WORD 0
.code
main PROC
xor ax, ax
mov esi, 0
    L1:
        add ax, [arr1+esi]
        add esi, 2
        cmp esi, 4
        jne L1

mov [result], ax
exit
main ENDP
END main
```

cmp esi,4 ; checks so that any garbage value is not added.

If esi = 4 then jne will not be executed rather move instruction will be executed

Loop continues if esi and 4 are not equal.

Same question using TYPE :

```
INCLUDE Irvine32.inc
.data
arr1 WORD 5h, 10h
result WORD 0
.code
main PROC
xor ax, ax
mov esi, 0
    L1:
       add ax, [arr1+esi]
       add esi, TYPE arr1
       cmp esi, 4
       jne L1
mov [result], ax
exit
main ENDP
END main
```

Same question using TYPE and loop L1:

```
INCLUDE Irvine32.inc
.data
arr1 WORD 5h, 10h
result WORD 0
.code
main PROC
xor ax, ax
mov ecx, LENGTHOF arr1
mov esi, 0
    L1:
       add ax, [arr1+esi]
       add esi, TYPE arr1
       loop L1
mov [result], ax
exit
main ENDP
END main
```

loop L1 automatically subtracts the ecx— and checks if it is zero then it execute the move instruction. If ecx is not zero then it executes the loop body.

NOTE TO REMEMBER: loop L1 first subtracts ecx then compares

So if value of ecx is already 1 then it would be an infinite loop.

Q) Copying a string

```
TITLE Copying a String
INCLUDE Irvine32.inc
.data
source BYTE "This is me Kinza",0
target BYTE SIZEOF source DUP(0)
.code
main PROC
mov esi, 0
mov ecx, SIZEOF source
L1:
    mov al, source[esi]
    mov target[esi], al
    inc esi
    loop L1
exit
main ENDP
END main
```

Q) What will be the value of eax?

```
TITLE what will be the value of eax
INCLUDE Irvine32.inc
.code
main PROC
mov eax, 0
mov ecx, 10
L1:
    mov eax, 3
    mov ecx, 5
       L2:
       add eax, 5  ≤ 1ms elapsed
       loop L2

loop L1

exit
main ENDP
END main
```

In this assembly program, the value of EAX will be determined by the operations inside the nested loops.

Here's a step-by-step breakdown:

1. **Initial Setup:**
   - EAX is initialized to 0.
   - ECX is initialized to 10.
2. **First Loop (L1):**
   - Inside the first loop, EAX is set to 3 and ECX is set to 5.
3. **Second Loop (L2):**
   - In the second loop (L2), EAX is incremented by 5 each time, and the loop instruction decrements ECX by 1 and repeats until ECX becomes 0.
   - The second loop will execute 5 times, adding 5 to EAX each time. So after the second loop finishes, the value of EAX will be: (EAX = 3 + (5 times 5) = 3 + 25 = 28).

4. Return to First Loop (L1):

- The outer loop (L1) decrements ECX (which was originally set to 10).
- However, since ECX is re-initialized to 5 every time inside the loop and it never reaches zero to stop the outer loop, the program enters an infinite loop.

Thus, in this infinite loop, after each iteration of L2, EAX will keep being set to 28.
Since the program loops infinitely without ever exiting, EAX will always be 28 after each execution of the second loop.

Q) Loop to print pattern.

```
TITLE to print the pattern
INCLUDE Irvine32.inc
.data
numstars DWORD 5
.code
main PROC
mov ecx, numstars
        outerloop:
             push ecx
             mov edx, numstars
             sub edx, 1

                  innerloop:
                       mov al, '*'
                       call WriteChar
                       dec edx
                       jnz innerloop
            call Crlf
            pop ecx
            loop outerloop
exit
main ENDP
END main
```

```
****
****
****
****
****
```

Q) Loop to print pattern.

```
TITLE to print the pattern
INCLUDE Irvine32.inc
.data
numstars DWORD 5
.code
main PROC
mov ecx, numstars
        outerloop:
             push ecx
             mov edx, numstars
             sub ecx, 1
             sub edx, ecx
                  innerloop:
                       mov al, '*'
                       call WriteChar
                       dec edx
                       jnz innerloop
            call Crlf
            pop ecx
            loop outerloop
exit
main ENDP
END main
```

```
*
**
***
****
*****
```

```
TITLE to print the pattern
INCLUDE Irvine32.inc
.data
numstars DWORD 5
.code
main PROC
mov ecx, numstars
mov ebx, 1
        outerloop:
             push ecx
             mov edx, ebx
             mov esi, 1
                  innerloop:
                       mov eax, esi
                       call WriteInt
                       inc esi
                       dec edx
                       jnz innerloop
            call Crlf
            pop ecx
            inc ebx
            loop outerloop
exit
main ENDP
END main
```

```
+1
+1+2
+1+2+3
+1+2+3+4
+1+2+3+4+5
```

C code:

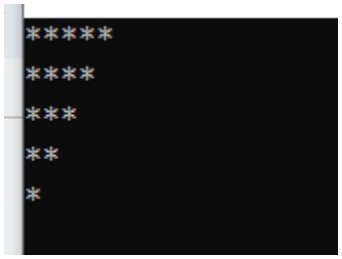
```c
// C program to print right half pyramid pattern of star
#include <stdio.h>

int main()
{
    int rows = 5;

    // first loop for printing rows
    for (int i = 0; i < rows; i++) {

        // second loop for printing character in each rows
        for (int j = 0; j <= i; j++) {
            printf("* ");
        }
        printf("\n");
    }
    return 0;
}
```

```c
int main()
{
    int rows = 5;

    // first loop for printing rows
    for (int i = 0; i < rows; i++) {

        // second loop for printing number in each rows
        for (int j = 0; j <= i; j++) {
            printf("%d ", j + 1);
        }
        printf("\n");
    }
    return 0;
}
```

```
*           |   1
* *         |   1 2
* * *       |   1 2 3
* * * *     |   1 2 3 4
* * * * *   |   1 2 3 4 5
```

Q) Loop to print pattern.

```
TITLE to print the pattern
INCLUDE Irvine32.inc
.data
numstars DWORD 5
.code
main PROC
mov ecx, numstars
mov ebx, 0
        outerloop:
             push ecx
             mov edx, numstars
             sub edx, ebx
             inc ebx
                  innerloop:
                       mov al, '*'
                       call WriteChar
                       dec edx
                       jnz innerloop
            call Crlf
            pop ecx
            loop outerloop
exit
main ENDP
END main
```



```
TITLE to print the pattern
INCLUDE Irvine32.inc
.data
numstars DWORD 5
.code
main PROC
mov ecx, numstars
mov ebx, numstars
        outerloop:
             push ecx
             mov edx, ebx
             mov esi, 1
                  innerloop:
                       mov eax, esi
                       call WriteInt
                       inc esi
                       dec edx
                       jnz innerloop
            call Crlf
            pop ecx
            dec ebx
            loop outerloop
exit
main ENDP
END main
```

```
+1+2+3+4+5
+1+2+3+4
+1+2+3
+1+2
+1
```

C code:

```c
int main()
{
    int rows = 5;

    // first loop to print all rows
    for (int i = 0; i < rows; i++) {

        // first inner loop to print the * in each row
        for (int j = 0; j < rows - i; j++) {
            printf("* ");
        }
        printf("\n");
    }
}
```

```c
int main()
{
    int rows = 5;

    // first loop to print all rows
    for (int i = 0; i < rows; i++) {

        // first inner loop to print the numbers in each row
        for (int j = 0; j < rows - i; j++) {
            printf("%d ", j + 1);
        }
        printf("\n");
    }
}
```

```
* * * * *    |   1 2 3 4 5
* * * *      |   1 2 3 4
* * *        |   1 2 3
* *          |   1 2
*            |   1
```

Q) Equilateral triangle

```
INCLUDE Irvine32.inc
.data
space BYTE ' '
star BYTE '*'
.code
main PROC
mov ecx,5 ;no of rows
mov ebx,1 ;no of stars
mov edi,4 ;no of spaces
```

```
l1:
mov edx,ecx
mov ecx,edi
cmp ecx,0 ;for last row no space so it'll run infinite
   jz lNOSPACE
       lspace:
       mov al,space
       call writeChar
       loop lspace
   lNOSPACE:
   mov ecx,ebx
   lstar:
   mov al,star
   call writeChar
   loop lstar

 call crlf
 add ebx,2
 sub edi,1
 mov ecx, edx
loop l1
```

```
exit
main ENDP
end main
```

```
    *
   ***
  *****
 *******
*********
```

Q) Diamond pattern

```
INCLUDE Irvine32.inc
.data
space BYTE ' '
star BYTE '*'
half DWORD 1 ; whether 1st or 2nd half
.code
main PROC
mov ecx,5 ;no of rows
mov ebx,1 ;no of stars
mov edi,4 ;no of spaces
```

```
l1:
mov edx,ecx
mov ecx,edi
cmp ecx,0 ;for last row no space so it'll run infinite
jz lNOSPACE
lspace:
mov al,space
call writeChar
loop lspace
lNOSPACE:
mov ecx,ebx
lstar:
mov al,star
call writeChar
loop lstar

cmp half,1
jz half1

jmp half2
half1:
call crlf
add ebx,2
sub edi,1
mov ecx, edx
loop l1
```

```
loop l1
cmp half,1 ;i.e first half done
jz l2
half2:
call crlf
sub ebx,2
add edi,1
mov ecx, edx
loop l1
jmp finish ;i.e sec half done
l2:
mov ecx,4
mov ebx,7
mov edi,1
mov half,2
jmp l1
finish:

exit
main ENDP
end main
```

```
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
```

Q) Use a loop to reverse the elements of array. [2 ways]

```
TITLE Reverse an array
INCLUDE Irvine32.inc
.data
arr DWORD 1, 2, 3, 4, 5, 6, 7, 8          ; Array of integers
arrsize = LENGTHOF arr                     ; Number of elements in the array
elementsize = TYPE arr                     ; Size of each element in bytes

.code
main PROC
    ; Initialize pointers
    mov esi, OFFSET arr                    ; ESI points to the first element of the array
    mov edi, OFFSET arr                    ; EDI will point to the last element
    add edi, (arrsize - 1) * elementsize   ; EDI points to the last element of the array
```

```
reverse_loop:
    cmp esi, edi                           ; Check if ESI has crossed EDI
    jge done                               ; Exit if pointers cross or meet

    ; Swap the elements at ESI and EDI
    mov eax, [esi]                         ; Load the value from ESI into EAX
    mov ebx, [edi]                         ; Load the value from EDI into EBX
    mov [esi], ebx                         ; Store EBX at ESI (swap)
    mov [edi], eax                         ; Store EAX at EDI (swap)

    ; Move pointers inward
    add esi, elementsize                   ; Move ESI to the next element
    sub edi, elementsize                   ; Move EDI to the previous element

    jmp reverse_loop                       ; Repeat the loop
```

```
done:
    mov esi, OFFSET arr                   ; Reset ESI to the start of the array
    mov ecx, arrsize                      ; ECX holds the number of elements
    print_loop:
    mov eax, [esi]                        ; Load the current array element into EAX
    call WriteDec                         ; Print the integer in EAX
    call Crlf                             ; Print a new line for each element
    add esi, elementsize                  ; Move to the next element
    loop print_loop                       ; Loop until ECX becomes 0 (all elements printed)

    exit
main ENDP
END main
```

```
8
7
6
5
4
3
2
1
```

```
TITLE Reverse an array
INCLUDE Irvine32.inc
.data
arr DWORD 1, 2, 3, 4, 5, 6, 7, 8          ; Array of integers
elementsize = TYPE arr                     ; Size of each element in bytes
arraysize = SIZEOF arr                     ; Total size of the array in bytes
arrcount = LENGTHOF arr                    ; Number of elements in the array (using LENGTHOF)
```

```
.code
main PROC
    ; Initialize pointers
    mov esi, OFFSET arr                    ; ESI points to the first element of the array
    mov edi, OFFSET arr                    ; EDI will point to the last element
    add edi, arraysize - elementsize       ; EDI points to the last element of the array

reverse_loop:
    cmp esi, edi                           ; Check if ESI has crossed EDI
    jge done                               ; Exit if pointers cross or meet

    ; Swap the elements at ESI and EDI
    mov eax, [esi]                         ; Load the value from ESI into EAX
    mov ebx, [edi]                         ; Load the value from EDI into EBX
    xchg eax, ebx
    mov [esi], eax                         ; Store EBX at ESI (swap)
    mov [edi], ebx                         ; Store EAX at EDI (swap)

    ; Move pointers inward
    add esi, elementsize                   ; Move ESI to the next element
    sub edi, elementsize                   ; Move EDI to the previous element

    jmp reverse_loop                       ; Repeat the loop
```

```
done:
    mov esi, OFFSET arr                   ; Reset ESI to the start of the array
    mov ecx, arrcount                     ; ECX holds the number of elements

print_loop:
    mov eax, [esi]                        ; Load the current array element into EAX
    call WriteDec                         ; Print the integer in EAX
    call Crlf                             ; Print a new line for each element
    add esi, elementsize                  ; Move to the next element
    loop print_loop                       ; Loop until ECX becomes 0 (all elements printed)
    exit
main ENDP
END main
```

```
8
7
6
5
4
3
2
1
```

Q) Copy a String backwards.

```
TITLE Copy a String Backwards
INCLUDE Irvine32.inc

.data
source BYTE "Hello, World!", 0      ; Source string with null terminator
target BYTE SIZEOF source DUP(?) ; Destination string (same size as source)
sourcesize = LENGTHOF source - 1        ; Exclude the null terminator from length
```

```
.code
main PROC
    ; Initialize pointers
    mov esi, OFFSET source          ; ESI points to the first character of source string
    mov edi, OFFSET target         ; EDI points to the first character of destination string
    add esi, sourcesize - 1        ; Move ESI to point to the last character of the source string
    mov ecx, sourcesize            ; ECX holds the number of characters to copy (excluding null terminator)
```

```
reverse_loop:
    mov al, [esi]                       ; Load character from source into AL
    mov [edi], al                       ; Store the character in the destination string
    dec esi                             ; Move ESI to the previous character
    inc edi                             ; Move EDI to the next character
    loop reverse_loop                   ; Repeat until all characters are copied

    ; Add null terminator to destination string
    mov BYTE PTR [edi], 0
```

```
    ; Print the reversed string
    mov edx, OFFSET target          ; Load the address of the destination string into EDX
    call WriteString                    ; Print the reversed string
    call Crlf                           ; Print a newline

    exit
main ENDP
END main
```

```
!dlroW ,olleH
```

Q) Fibonacci sequence

```
TITLE Fibonacci sequence
INCLUDE Irvine32.inc
.code
main PROC
    ; Initializing the first two Fibonacci numbers
    mov eax, 1              ; Fib(1)
    mov ebx, 1              ; Fib(2)

    ; Display the first Fibonacci number (Fib(1))
    call DumpRegs

    ; Display the second Fibonacci number (Fib(2))
    call DumpRegs

    ; Calculate Fibonacci sequence for n = 3 to n = 7
    mov ecx, 7              ; We need to calculate 5 more values (Fib(3) to Fib(7))
```

```
fib_loop:
    mov edx, eax            ; Temporarily store the previous Fib(n-2) in EDX
    add eax, ebx            ; EAX = Fib(n) = Fib(n-1) + Fib(n-2)
    mov ebx, edx            ; EBX = Fib(n-1), which is the old Fib(n-2)

    ; Display the current Fibonacci number in EAX
    call DumpRegs

    ; Decrease loop counter and continue
    dec ecx
    jnz fib_loop

    ; Exit the program
    exit
main ENDP
END main
```

```
EAX=00000001  EBX=00000001  ECX=007310AA  EDX=007310AA
ESI=007310AA  EDI=007310AA  EBP=009CFEF4  ESP=009CFEE8
EIP=0073366F  EFL=00000246  CF=0  SF=0  ZF=1  OF=0  AF=0  PF=1


EAX=00000001  EBX=00000001  ECX=007310AA  EDX=007310AA
ESI=007310AA  EDI=007310AA  EBP=009CFEF4  ESP=009CFEE8
EIP=00733674  EFL=00000246  CF=0  SF=0  ZF=1  OF=0  AF=0  PF=1


EAX=00000002  EBX=00000001  ECX=00000007  EDX=00000001
ESI=007310AA  EDI=007310AA  EBP=009CFEF4  ESP=009CFEE8
EIP=00733684  EFL=00000202  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=0


EAX=00000003  EBX=00000002  ECX=00000006  EDX=00000002
ESI=007310AA  EDI=007310AA  EBP=009CFEF4  ESP=009CFEE8
EIP=00733684  EFL=00000206  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=1


EAX=00000005  EBX=00000003  ECX=00000005  EDX=00000003
ESI=007310AA  EDI=007310AA  EBP=009CFEF4  ESP=009CFEE8
EIP=00733684  EFL=00000206  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=1


EAX=00000008  EBX=00000005  ECX=00000004  EDX=00000005
ESI=007310AA  EDI=007310AA  EBP=009CFEF4  ESP=009CFEE8
EIP=00733684  EFL=00000202  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=0


EAX=0000000D  EBX=00000008  ECX=00000003  EDX=00000008
ESI=007310AA  EDI=007310AA  EBP=009CFEF4  ESP=009CFEE8
EIP=00733684  EFL=00000202  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=0


EAX=00000015  EBX=0000000D  ECX=00000002  EDX=0000000D
ESI=007310AA  EDI=007310AA  EBP=009CFEF4  ESP=009CFEE8
EIP=00733684  EFL=00000212  CF=0  SF=0  ZF=0  OF=0  AF=1  PF=0
```

4. Direct-Offset Addressing

Insert the following variables in your program:

```
.data
Uarray WORD 1000h,2000h,3000h,4000h
Sarray SWORD -1,-2,-3,-4
```

Write instructions that use direct-offset addressing to move the four values in **Uarray** to the EAX, EBX, ECX, and EDX registers. When you follow this with a **call DumpRegs** statement (see Section 3.2), the following register values should display:

```
EAX=00001000  EBX=00002000  ECX=00003000  EDX=00004000
```

Next, write instructions that use direct-offset addressing to move the four values in **Sarray** to the EAX, EBX, ECX, and EDX registers. When you follow this with a **call DumpRegs** statement, the following register values should display:

```
EAX=FFFFFFFF  EBX=FFFFFFFE  ECX=FFFFFFFD  EDX=FFFFFFFC
```

```
TITLE Direct offset addressing
INCLUDE Irvine32.inc
.data
Uarray WORD 1000h,2000h,3000h,4000h
Sarray SWORD -1,-2,-3,-4
.code
main PROC
    ; Load values from Uarray into EAX, EBX, ECX, EDX using direct-offset addressing
    movzx eax,  [Uarray]     ; Move Uarray[0] to EAX (zero extend)
    movzx ebx, [Uarray+2]   ; Move Uarray[1] to EBX (zero extend)
    movzx ecx, [Uarray+4]   ; Move Uarray[2] to ECX (zero extend)
    movzx edx,  [Uarray+6]   ; Move Uarray[3] to EDX (zero extend)

    ; Display the register values for Uarray
    call DumpRegs

    ; Load values from Sarray into EAX, EBX, ECX, EDX using direct-offset addressing
    movsx eax,  [Sarray]     ; Move Sarray[0] to EAX (sign extend)
    movsx ebx, [Sarray+2]   ; Move Sarray[1] to EBX (sign extend)
  ;  movsx ecx, [Sarray+4]   ; Move Sarray[2] to ECX (sign extend)
    movsx edx, [Sarray+6]   ; Move Sarray[3] to EDX (sign extend)

    ; Display the register values for Sarray
    call DumpRegs
exit
main ENDP
END main
```

```
EAX=00001000  EBX=00002000  ECX=00003000  EDX=00004000
ESI=004A10AA  EDI=004A10AA  EBP=00AFFE30  ESP=00AFFE24
EIP=004A3681  EFL=00000246  CF=0  SF=0  ZF=1  OF=0  AF=0  PF=1


EAX=FFFFFFFF  EBX=FFFFFFFE  ECX=00003000  EDX=FFFFFFFC
ESI=004A10AA  EDI=004A10AA  EBP=00AFFE30  ESP=00AFFE24
EIP=004A369B  EFL=00000246  CF=0  SF=0  ZF=1  OF=0  AF=0  PF=1
```

Q) Shifting the Elements in an Array

Using a loop and indexed addressing, write code that rotates the members of a 32-bit integer array forward one position. The value at the end of the array must wrap around to the first position. For example, the array [10,20,30,40] would be transformed into [40,10,20,30].

Here i have used an additional register ebx to resolve the constant value error.
ebx is used instead of ecx-1.

```
TITLE Rotate 32-bit Array forward
INCLUDE Irvine32.inc
.data
arr DWORD 10, 20, 30, 40
arrsize = LENGTHOF arr
arrt = TYPE arr
.code
main PROC
mov esi, OFFSET arr
mov ecx, arrsize
mov ebx, ecx
sub ebx, 1
mov eax, [esi+ebx * 4]
dec ecx
mov ebx, 0
```

```
      shiftloop:
      mov ebx, ecx
      sub ebx, 1
      mov edx, [esi + ebx * 4]
      mov [esi + ecx * 4], edx
      dec ecx
      jnz shiftloop

mov [esi], eax
```

```
        mov ecx, arrsize
        mov esi, OFFSET arr
        printloop:
        mov eax, [esi]
        call WriteInt
        call Crlf
        add esi, 4
        loop printloop

 exit
 main ENDP
 END main
```

```
+40
+10
+20
+30
```

Q) Summing the Gaps between Array Values

Write a program with a loop and indexed addressing that calculates the sum of all the gaps between successive array elements. The array elements are doublewords, sequenced in non decreasing order. So, for example, the array {0, 2, 5, 9, 10} has gaps of 2, 3, 4, and 1, whose sum equals 10.

```
TITLE Summing the gaps
INCLUDE Irvine32.inc
.data
arr DWORD 0, 2, 5, 9, 10
arrsize = LENGTHOF arr
arrt = TYPE arr
.code
main PROC
mov esi, OFFSET arr
mov ecx, arrsize
dec ecx
mov edx, 0
```

```
L1:
   mov eax, [esi + ecx *4]
   dec ecx
   mov ebx, [esi + ecx * 4]
   inc ecx
   sub eax, ebx
   add edx, eax
   loop L1
   mov eax, edx
   call WriteDec
    call Crlf
exit
main ENDP
END main
```

```
10
```

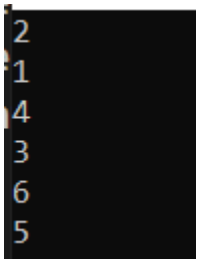
Q) Exchanging Pairs of Array Values

Write a program with a loop and indexed addressing that exchanges every pair of values in an array with an even number of elements. Therefore, item i will exchange with item i+1, and item i+2 will exchange with item i+3, and so on.

```
TITLE Exchanging the pairs
INCLUDE Irvine32.inc
.data
arr BYTE 1, 2, 3, 4, 5, 6
arrlen = LENGTHOF arr
arrsize = TYPE arr
.code
main PROC
mov esi, 0
mov ecx, arrlen
```

```
L1:
mov al, [arr + esi]
mov bl, [arr + esi +arrsize]
mov [arr + esi], bl
mov [arr + esi + 1], al
add esi, arrsize*2
loop L1
```

```
mov esi, OFFSET arr
mov ecx, LENGTHOF arr
 printloop:
      movzx eax, BYTE PTR [esi]
      call WriteDec
      call Crlf
      add esi, arrsize
      loop printloop

    exit
main ENDP
END main
```



Q) Converting from Big Endian to Little Endian

Write a program that uses the variables below and MOV instructions to copy the value from bigEndian to littleEndian, reversing the order of the bytes. The number's 32-bit value is understood to be 12345678 hexadecimal

.data

bigEndian BYTE 12h,34h,56h,78h

littleEndian DWORD?

```
1       ; Copy and reverse the bytes from
            bigEndian to littleEndian
2       mov al, bigEndian       ; Load the
            first byte (12h) from bigEndian
            into AL
3       mov ah, bigEndian+1     ; Load the
            second byte (34h) into AH
4       xchg al, ah             ; Swap AL
            and AH
5       mov bx, ax              ; Move the
            swapped bytes into BX (34h12h)
6
7       mov al, bigEndian+2     ; Load the
            third byte (56h) into AL
8       mov ah, bigEndian+3     ; Load the
            fourth byte (78h) into AH
9       xchg al, ah             ; Swap AL
            and AH
10      mov cx, ax              ; Move the
            swapped bytes into CX (78h56h)
11
12      mov word ptr littleEndian, cx ;
            Store CX (78h56h) into the
            first two bytes of littleEndian
13      mov word ptr littleEndian+2, bx ;
            Store BX (34h12h) into the last
            two bytes of littleEndian
```

Task: 1 Write a program that uses a loop to calculate the first ten numbers of Fibonacci sequence.

```
TITLE Fibonacci sequence
INCLUDE Irvine32.inc
.code
main PROC
    mov eax, 1
    mov ebx, 1
    call DumpRegs

    call DumpRegs
    mov ecx, 8

fibloop:
    mov edx, eax
    add eax, ebx
    mov ebx, edx
    call DumpRegs
    dec ecx
    jnz fibloop

    exit
main ENDP
END main
```

```
EAX=00000001  EBX=00000001  ECX=00AA10AA  EDX=00AA10AA
ESI=00AA10AA  EDI=00AA10AA  EBP=008FF79C  ESP=008FF790
EIP=00AA366F  EFL=00000246  CF=0  SF=0  ZF=1  OF=0  AF=0  PF=1


EAX=00000001  EBX=00000001  ECX=00AA10AA  EDX=00AA10AA
ESI=00AA10AA  EDI=00AA10AA  EBP=008FF79C  ESP=008FF790
EIP=00AA3674  EFL=00000246  CF=0  SF=0  ZF=1  OF=0  AF=0  PF=1


EAX=00000002  EBX=00000001  ECX=00000008  EDX=00000001
ESI=00AA10AA  EDI=00AA10AA  EBP=008FF79C  ESP=008FF790
EIP=00AA3684  EFL=00000202  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=0


EAX=00000003  EBX=00000002  ECX=00000007  EDX=00000002
ESI=00AA10AA  EDI=00AA10AA  EBP=008FF79C  ESP=008FF790
EIP=00AA3684  EFL=00000206  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=1
```

```
EAX=00000005  EBX=00000003  ECX=00000006  EDX=00000003
ESI=00AA10AA  EDI=00AA10AA  EBP=00BAF8D0  ESP=00BAF8C4
EIP=00AA3684  EFL=00000206  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=1


EAX=00000008  EBX=00000005  ECX=00000005  EDX=00000005
ESI=00AA10AA  EDI=00AA10AA  EBP=00BAF8D0  ESP=00BAF8C4
EIP=00AA3684  EFL=00000202  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=0


EAX=0000000D  EBX=00000008  ECX=00000004  EDX=00000008
ESI=00AA10AA  EDI=00AA10AA  EBP=00BAF8D0  ESP=00BAF8C4
EIP=00AA3684  EFL=00000202  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=0


EAX=00000015  EBX=0000000D  ECX=00000003  EDX=0000000D
ESI=00AA10AA  EDI=00AA10AA  EBP=00BAF8D0  ESP=00BAF8C4
EIP=00AA3684  EFL=00000212  CF=0  SF=0  ZF=0  OF=0  AF=1  PF=0


EAX=00000022  EBX=00000015  ECX=00000002  EDX=00000015
ESI=00AA10AA  EDI=00AA10AA  EBP=00BAF8D0  ESP=00BAF8C4
EIP=00AA3684  EFL=00000216  CF=0  SF=0  ZF=0  OF=0  AF=1  PF=1


EAX=00000037  EBX=00000022  ECX=00000001  EDX=00000022
ESI=00AA10AA  EDI=00AA10AA  EBP=00BAF8D0  ESP=00BAF8C4
EIP=00AA3684  EFL=00000202  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=0
```

Task: 2 write a program that uses a nested loop to implement following patterns.

1	1111	4321	1234
11	111	432	123
111	11	43	12
1111	1	4	1

```
TITLE to print the pattern
INCLUDE Irvine32.inc
.data
numlines DWORD 4
.code
main PROC
mov ecx, numlines
mov ebx, numlines
        outerloop:
             mov edx, ebx
             mov esi, 1
                  innerloop:
                       mov eax, esi
                       call WriteDec
                       inc esi
                       dec edx
                       jnz innerloop
            call Crlf
            dec ebx
            loop outerloop
exit
main ENDP
END main
```

```
1234
123
12
1
```

```
TITLE to print the pattern
INCLUDE Irvine32.inc
.data
numlines DWORD 4
.code
main PROC
mov ecx, numlines
mov ebx, numlines
        outerloop:
             mov edx, ebx
             mov esi, 1
                  innerloop:
                       mov eax, esi
                       call WriteDec
                       dec edx
                       jnz innerloop
            call Crlf
            dec ebx
            loop outerloop
exit
main ENDP
END main
```

```
1111
111
11
1
```

```
TITLE to print the pattern
INCLUDE Irvine32.inc
.data
numlines DWORD 4
.code
main PROC
mov ecx, numlines
mov ebx, numlines
        outerloop:
             mov edx, ebx
             mov esi, 4
                  innerloop:
                       mov eax, esi
                       call WriteDec
                       dec esi
                       dec edx
                       jnz innerloop
            call Crlf
            dec ebx
            loop outerloop
exit
main ENDP
END main
```

```
4321
432
43
4
```

```
TITLE to print the pattern
INCLUDE Irvine32.inc
.data
numlines DWORD 4
.code
main PROC
mov ecx, numlines
mov ebx, 1
        outerloop:
             mov edx, ebx
             mov esi, 1
                  innerloop:
                      mov eax, esi
                       call WriteDec
                       dec edx
                       jnz innerloop
            call Crlf
            inc ebx
            cmp ebx, numlines
            jbe outerloop
exit
main ENDP
END main
```

```
1
11
111
1111
```

Q) Compute the formula

```
TITLE (a + b)^3 Implementation
INCLUDE Irvine32.inc
.data
    a DWORD ?                ; Placeholder for variable a
    b DWORD ?                ; Placeholder for variable b
    a_cubed DWORD ?          ; To store a^3
    b_cubed DWORD ?          ; To store b^3
    three_a2b DWORD ?        ; To store 3a^2b
    three_ab2 DWORD ?         ; To store 3ab^2
    result DWORD ?           ; To store final result
```

```
.code
main PROC
    ; Compute a^3
    mov eax, a               ; Load a into EAX
    imul eax, a              ; EAX = a * a (a^2)
    imul eax, a              ; EAX = a^2 * a (a^3)
    mov a_cubed, eax         ; Store a^3
    ; Compute b^3
    mov eax, b               ; Load b into EAX
    imul eax, b              ; EAX = b * b (b^2)
    imul eax, b              ; EAX = b^2 * b (b^3)
    mov b_cubed, eax         ; Store b^3
```

```
; Compute 3a^2b
mov eax, a                  ; Load a into EAX
imul eax, a                 ; EAX = a * a (a^2)
imul eax, b                 ; EAX = a^2 * b
mov ebx, eax
shl eax, 1
add eax, ebx
mov three_a2b, eax          ; Store 3a^2b
```

```
    ; Compute 3ab^2
    mov eax, b              ; Load b into EAX
   imul eax, b             ; EAX = b * b (b^2)
    imul eax, a             ; EAX = b^2 * a
    mov ebx, eax
    shl eax, 1
    add eax, ebx
    mov three_ab2, eax      ; Store 3ab^2
    ; Final result: a^3 + b^3 + 3a^2b + 3ab^2
    mov eax, a_cubed        ; Load a^3 into EAX
    add eax, b_cubed        ; Add b^3
    add eax, three_a2b      ; Add 3a^2b
    add eax, three_ab2      ; Add 3ab^2
    mov result, eax         ; Store the final result

    exit
main ENDP
END main
```

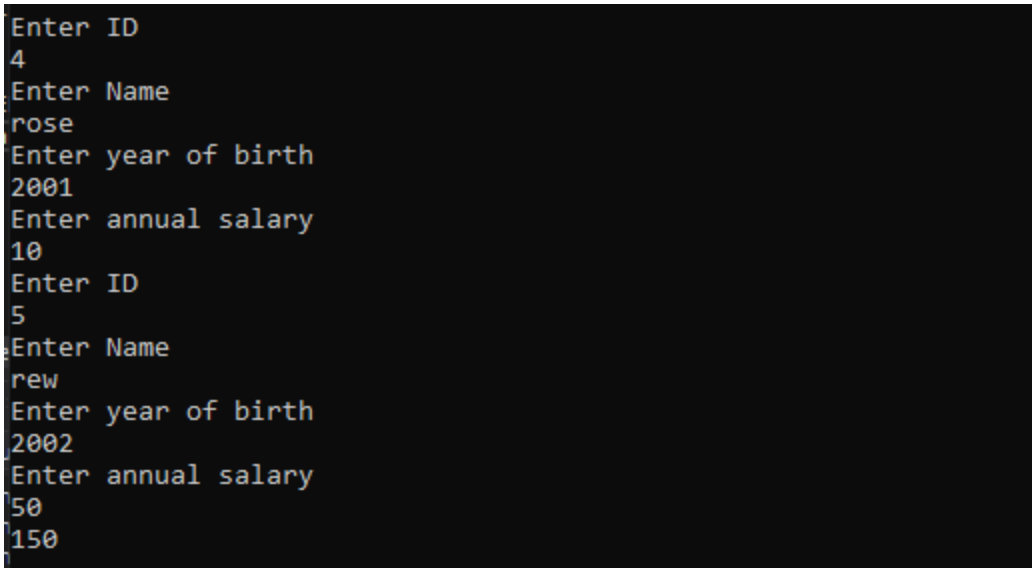
Task: 3 write a program to take input data for 5 employees and store it in appropriate variables. The program should ask for Employee ID, Name, Year of Birth & Annual Salary from the user. All variables should be stored in an array whose index represent employee number. The program should then calculate the annual salary for all employees by adding all the elements in AnnualSalary array.

```
TITLE Employee Data Input and Salary Calculation
INCLUDE Irvine32.inc
.data
empid DWORD 5 DUP(?)
namm DWORD 5 DUP(?)
yrofbirth DWORD 5 DUP(?)
annualsal DWORD 5 DUP(?)
msg1 BYTE "Enter ID",0
msg2 BYTE "Enter Name",0
msg3 BYTE "Enter year of birth",0
msg4 BYTE "Enter annual salary",0
TotalSalary DWORD ?
```

```
.code
main PROC
mov ecx, 5
mov ebx, 0
L1:
mov edx,offset msg1
call writeString
call crlf
call readint
mov [empid+ebx],eax

mov edx,offset msg2
call writeString
call crlf
mov edx,offset namm
call readstring

mov edx,offset msg3
call writeString
call crlf
mov edx,offset yrofbirth
call readstring
mov [yrofbirth+ ebx],eax

mov edx,offset msg4
call writeString
call crlf
call readint
mov [annualsal+ebx],eax
add ebx, 4
loop L1
```

```
mov ecx, 5
    mov eax, 0
    mov esi, OFFSET annualsal

salaryloop:
    add eax, [esi]
    add esi, 4
    loop salaryloop

    mov TotalSalary, eax

    mov edx, TotalSalary
    call WriteDec
    call Crlf

    exit
    main ENDP
    END main
```

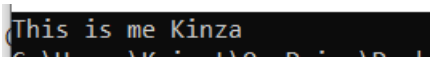
```
Enter ID
1
Enter Name
kinza
Enter year of birth
7124
Enter annual salary
40
Enter ID
2
Enter Name
jim
Enter year of birth
2004
Enter annual salary
30
Enter ID
3
Enter Name
jasy
Enter year of birth
2003
Enter annual salary
20
```



150 is printed which is the total annual salary.

Task: 4 Initialize an array named Source and use a loop with indexed addressing to copy a string represented as an array of bytes with a null terminator value in an array named as target.

```
TITLE Copying a String
INCLUDE Irvine32.inc
.data
source BYTE "This is me Kinza",0
target BYTE SIZEOF source DUP(0)
.code
main PROC
mov esi, 0
mov ecx, SIZEOF source
L1:
    mov al, source[esi]
    mov target[esi], al
    inc esi
    loop L1
    mov edx, OFFSET target
    call WriteString
exit
main ENDP
END main
```



Task: 5 Use a loop with direct or indirect addressing to reverse the elements of an integer array in place. Do not copy elements to any other array. Use SIZEOF, TYPE and LENGTHOF operators to make program flexible.

```
TITLE Reverse an array
INCLUDE Irvine32.inc
.data
arr DWORD 1, 2, 3, 4, 5, 6, 7, 8
arrsize = SIZEOF arr
elementsize = TYPE arr
arrcount = LENGTHOF arr
```

```
.code
main PROC
    mov esi, OFFSET arr
    mov edi, OFFSET arr
    add edi, arrsize - elementsize
reverse_loop:
    cmp esi, edi
    jge done

    mov eax, [esi]
    mov ebx, [edi]
    mov [esi], ebx
    mov [edi], eax
    add esi, elementsize
    sub edi, elementsize

    jmp reverse_loop
```

```
done:
    mov esi, OFFSET arr
    mov ecx, arrcount
    printloop:
    mov eax, [esi]
    call WriteDec
    call Crlf
    add esi, elementsize
    loop printloop

    exit
main ENDP
END main
```

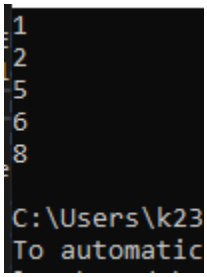
```
8
7
6
5
4
3
2
1
```

Task: 6 initialize a double word array consisting of elements 8, 5,1,2,6. Sort the given array in ascending order using bubble sort.

```
TITLE Sorting an Array
INCLUDE Irvine32.inc
.data
arr DWORD 8,5,1,2,6
swap BYTE 0
```

```
.code
main PROC
 mov ecx, LENGTHOF arr
      outerloop:
            mov [swap], 0
            mov ebx, 0
        innerloop:
        mov eax, [arr+ebx]
        mov edx, [arr+ebx+4]
        cmp eax, edx
        jbe noswap
        ;swap
       mov [arr + ebx], edx
       mov [arr + ebx + 4], eax
       mov [swap], 1
    noswap:
    add ebx, 4
    cmp ebx, (LENGTHOF arr - 1) * 4
    jl innerloop
    cmp [swap], 1
    je outerloop
```

```
    mov esi, OFFSET arr
    mov ecx, LENGTHOF arr
    printloop:
    mov eax, [esi]
    call WriteDec
    call Crlf
    add esi, 4
    loop printloop
exit
main ENDP
END main
```



Q) Compare source and destination: Here _x is taken as destination.

```
TITLE COMPARE NUMBERS
INCLUDE Irvine32.inc
.data
_x DWORD 10h
_y DWORD 20h
msg1 BYTE "Destination and source are equal",0
msg2 BYTE "Destination is less than source",0
msg3 BYTE "Destination is greater than source",0
```

```
.code
main PROC
mov eax, 0
mov ebx, 0
mov eax, _x
mov ebx, _y
```

```
cmp eax, ebx
je equal
jg greater
jl less
equal:
mov edx, OFFSET msg1
call WriteString
greater:
mov edx, OFFSET msg3
call WriteString
less:
mov edx, OFFSET msg2
call WriteString
exit
main ENDP
END main
```

```
Destination is less than source
```

Q) Take an array and iterate through it if x>arr[x] print msg if arr[x] == x.

```
TITLE Compare Array Elements
INCLUDE Irvine32.inc
.data
x DWORD 4
arr DWORD 0, 2, 4, 6, 5, 8, 6  ; Example array
msg1 BYTE "arr[x] is equal to x", 0
msg2 BYTE "x is less than arr[x]", 0
msg3 BYTE "x is greater than arr[x]", 0
```

```
.code
main PROC
    mov ebx, 0
    mov ebx, x
    mov esi,0
    mov ecx, LENGTHOF arr
```

```
L1:
    cmp esi, ecx
    jge end_loop
    mov eax, arr[esi*4]
    cmp ebx, eax
    je equal
    jg greater
    jl less
  next_iteration:
  inc esi
  jmp L1
```

```
 equal:
 mov edx, OFFSET msg1
 call WriteString
 call Crlf
 jmp next_iteration
  greater:
 mov edx, OFFSET msg3
 call WriteString
 call Crlf
  jmp next_iteration
  less:
 mov edx, OFFSET msg2
 call WriteString
call CrLf
 jmp next_iteration

end_loop:
exit
main ENDP
END main
```

```
x is greater than arr[x]
x is greater than arr[x]
arr[x] is equal to x
x is less than arr[x]
x is less than arr[x]
x is less than arr[x]
x is less than arr[x]
```

★ **1. Carry Flag**

Write a program that uses addition and subtraction to set and clear the Carry flag. After each instruction, insert the **call DumpRegs** statement to display the registers and flags. Using comments, explain how (and why) the Carry flag was affected by each instruction.

★ **2. Zero and Sign Flags**

Write a program that uses addition and subtraction to set and clear the Zero and Sign flags. After each addition or subtraction instruction, insert the **call DumpRegs** statement (see Section 3.2) to display the registers and flags. Using comments, explain how (and why) the Zero and Sign flags were affected by each instruction.

★ **3. Overflow Flag**

Write a program that uses addition and subtraction to set and clear the Overflow flag. After each addition or subtraction instruction, insert the **call DumpRegs** statement (see Section 3.2) to display the registers and flags. Using comments, explain how (and why) the Overflow flag was affected by each instruction. Include an ADD instruction that sets both the Carry and Overflow flags

```
TITLE Carry Flag Example
INCLUDE Irvine32.inc

.code
main PROC
    mov al, 0FFh        ; Load AL with 255 (max value for 8 bits)
    sub al, 1h          ; This will set the Carry flag (AL = 0, CF = 1)
    call DumpRegs       ; Display registers and flags

    sub al, 1h          ; This will clear the Carry flag (AL = 255, CF = 0)
    call DumpRegs       ; Display registers and flags

    sub al, 1h          ; This will not affect the Carry flag (AL = 254, CF = 0)
    call DumpRegs       ; Display registers and flags

    exit
main ENDP
END main
```

```
EAX=012FFA00  EBX=01018000  ECX=00E410AA  EDX=00E410AA
ESI=00E410AA  EDI=00E410AA  EBP=012FFA70  ESP=012FFA64
EIP=00E43669  EFL=00000257  CF=1  SF=0  ZF=1  OF=0  AF=1  PF=1


EAX=012FFAFF  EBX=01018000  ECX=00E410AA  EDX=00E410AA
ESI=00E410AA  EDI=00E410AA  EBP=012FFA70  ESP=012FFA64
EIP=00E43670  EFL=00000297  CF=1  SF=1  ZF=0  OF=0  AF=1  PF=1


EAX=012FFAFE  EBX=01018000  ECX=00E410AA  EDX=00E410AA
ESI=00E410AA  EDI=00E410AA  EBP=012FFA70  ESP=012FFA64
EIP=00E43677  EFL=00000282  CF=0  SF=1  ZF=0  OF=0  AF=0  PF=0
```

```
TITLE Zero and Sign Flags Example
INCLUDE Irvine32.inc

.code
main PROC
    mov al, 2          ; Load AL with 2
    sub al, 2          ; This will set the Zero flag (AL = 0, ZF = 1)
    call DumpRegs      ; Display registers and flags

    mov al, 0          ; Load AL with 0
    sub al, 1          ; This will set the Sign flag (AL = -1, SF = 1)
    call DumpRegs      ; Display registers and flags

    add al, 1          ; This clears the Sign flag and Zero flag (AL = 0, ZF = 1, SF = 0)
    call DumpRegs      ; Display registers and flags

    exit
main ENDP
END main
```

```
EAX=00EFFE00  EBX=00D72000  ECX=001810AA  EDX=001810AA
ESI=001810AA  EDI=001810AA  EBP=00EFFEA4  ESP=00EFFE98
EIP=00183669  EFL=00000246  CF=0  SF=0  ZF=1  OF=0  AF=0  PF=1


EAX=00EFFEFF  EBX=00D72000  ECX=001810AA  EDX=001810AA
ESI=001810AA  EDI=001810AA  EBP=00EFFEA4  ESP=00EFFE98
EIP=00183672  EFL=00000297  CF=1  SF=1  ZF=0  OF=0  AF=1  PF=1


EAX=00EFFE00  EBX=00D72000  ECX=001810AA  EDX=001810AA
ESI=001810AA  EDI=001810AA  EBP=00EFFEA4  ESP=00EFFE98
EIP=00183679  EFL=00000257  CF=1  SF=0  ZF=1  OF=0  AF=1  PF=1
```

```
TITLE Overflow Flag Example
INCLUDE Irvine32.inc

.code
main PROC
    mov al, 7Fh         ; Load AL with 127 (max positive value for signed 8-bit)
    add al, 1           ; This will set the Overflow flag (AL = -128, OF = 1)
    call DumpRegs       ; Display registers and flags

    mov al, -128        ; Load AL with -128
    sub al, 1           ; This will set the Overflow flag again (AL = 127, OF = 1)
    call DumpRegs       ; Display registers and flags

    mov al, 1
    sub al, 2           ; This will not set the Overflow flag (AL = -1, OF = 0)
    call DumpRegs       ; Display registers and flags

    exit
main ENDP
END main
```

```
EAX=005DFF80  EBX=00940000  ECX=006310AA  EDX=006310AA
ESI=006310AA  EDI=006310AA  EBP=005DFF54  ESP=005DFF48
EIP=00633669  EFL=00000A92  CF=0  SF=1  ZF=0  OF=1  AF=1  PF=0


EAX=005DFF7F  EBX=00940000  ECX=006310AA  EDX=006310AA
ESI=006310AA  EDI=006310AA  EBP=005DFF54  ESP=005DFF48
EIP=00633672  EFL=00000A12  CF=0  SF=0  ZF=0  OF=1  AF=1  PF=0


EAX=005DFFFF  EBX=00940000  ECX=006310AA  EDX=006310AA
ESI=006310AA  EDI=006310AA  EBP=005DFF54  ESP=005DFF48
EIP=0063367B  EFL=00000297  CF=1  SF=1  ZF=0  OF=0  AF=1  PF=1
```

Q) Compare source and destination by carry flags. Here _x is taken as the destination.

```
TITLE COMPARE NUMBERS
INCLUDE Irvine32.inc
.data
_x DWORD 10h
_y DWORD 20h
msg1 BYTE "Destination and source are equal",0
msg2 BYTE "Destination is less than source",0
msg3 BYTE "Destination is greater than source",0
.code
main PROC
mov eax, 0
mov ebx, 0
mov eax, _x
mov ebx, _y
cmp eax, ebx
jz equal
jc less
jmp greater
```

```
equal:
mov edx, OFFSET msg1
call WriteString
greater:
mov edx, OFFSET msg3
call WriteString
less:
mov edx, OFFSET msg2
call WriteString
exit
main ENDP
END main
```

```
Destination is less than source
```

Q) Comparison

```
TITLE Compare Signed and Unsigned Integers
INCLUDE Irvine32.inc

.data
msg_equal BYTE "The integers are equal", 0
msg_signed_greater BYTE "Signed: Destination is greater than source", 0
msg_signed_less BYTE "Signed: Destination is less than source", 0
msg_unsigned_greater BYTE "Unsigned: Destination is greater than source", 0
msg_unsigned_less BYTE "Unsigned: Destination is less than source", 0
```

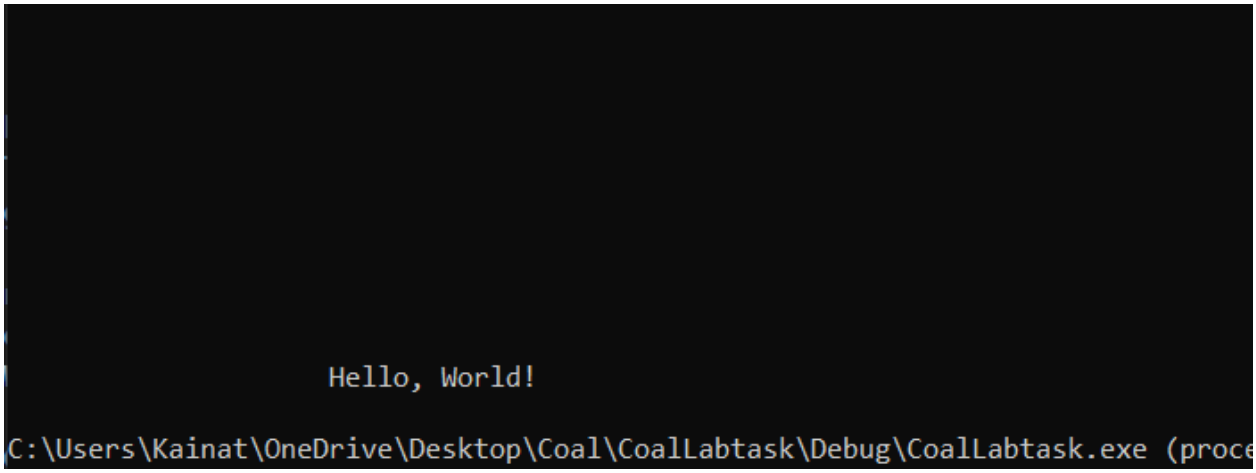
```
.code
main PROC
    ; Initialize the values of source and destination
    mov eax, -50          ; Set destination value (signed integer)
    mov ebx, -250          ; Set source value (unsigned integer)

    ; Perform the comparison
    cmp eax, ebx          ; Compare EAX and EBX

    ; Check if the values are equal
    je equal

    ; For signed comparison, check the Sign and Overflow flags
    js signed_less        ; If SF = 1, EAX is less (signed comparison)
    jo signed_greater     ; If OF = 1, EAX is greater (signed comparison)

    ; For unsigned comparison, if none of the above, use carry flag (CF)
    jb unsigned_less      ; If CF = 1, EAX is less (unsigned comparison)
    ja unsigned_greater   ; If CF = 0 and EAX is not equal, EAX is greater
```

```
equal:
    mov edx, OFFSET msg_equal
    call WriteString
    jmp end_program

signed_less:
    mov edx, OFFSET msg_signed_less
    call WriteString
    jmp end_program

signed_greater:
    mov edx, OFFSET msg_signed_greater
    call WriteString
    jmp end_program
```

```
unsigned_less:
    mov edx, OFFSET msg_unsigned_less
    call WriteString
    jmp end_program

unsigned_greater:
    mov edx, OFFSET msg_unsigned_greater
    call WriteString

end_program:
    call CrLf
    exit
main ENDP
END main
```

Q) Gotoxy (DH=row , DL=col)

```
TITLE Gotoxy Example
INCLUDE Irvine32.inc

.data
    msg BYTE "Hello, World!", 0 ; Message to display

.code
main PROC
    ; Set the cursor position using gotoxy
    mov dh, 10          ; Row (Y coordinate)
    mov dl, 20          ; Column (X coordinate)
    call gotoxy         ; Call gotoxy to move the cursor

    ; Print the message
    mov edx, OFFSET msg ; Load address of message
    call WriteString     ; Display the message

    call Crlf           ; Print a new line
    exit                ; Exit the program
main ENDP
END main
```

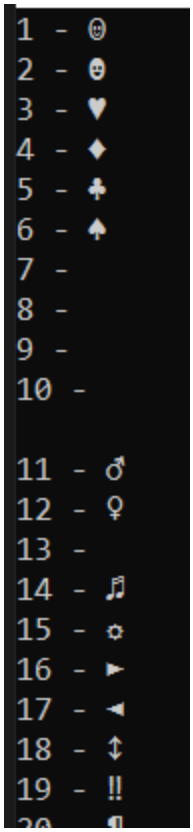


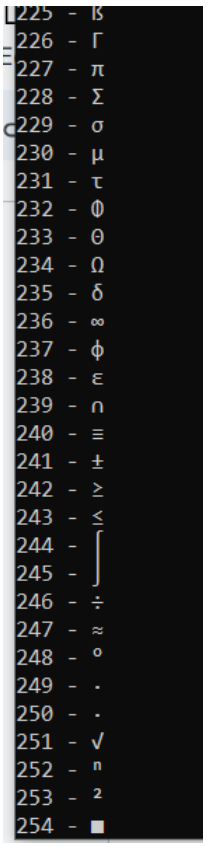
Q) use GetMaxXY

```
TITLE GetMaxXY Example
INCLUDE Irvine32.inc

.code
main PROC
    ; Call GetMaxXY to retrieve the maximum column and row sizes
    call GetMaxXY        ; Call GetMaxXY
    ; DX now contains the number of columns
    ; AX now contains the number of rows

    ; Display the number of rows (in AX) and columns (in DX)
    mov ax, ax           ; EAX was unnecessary; we can use AX directly for output
    call WriteDec        ; Print the number of rows
    call Crlf           ; Move to the next line

    mov dx, dx           ; Move column count to DX for output
    call WriteDec      ; Print the number of columns
    call Crlf           ; Move to the next line

    exit                 ; Exit the program
main ENDP
END main
```

Q) Call WriteDec prints 1-255, Call WriteString prints -, call WriteChar prints character

```
INCLUDE Irvine32.inc

.data
Dash BYTE " - ", 0  ; String to print after each number
.code
main PROC
    mov ecx, 1         ; Set loop counter (255 iterations)
    mov al, 1            ; Initialize AL to 1 (first character)
L1:
    mov eax, ecx         ; Move current loop counter value into EAX
    call WriteDec        ; Write the current value of EAX
    mov edx, OFFSET Dash ; Point EDX to the dash string
    call WriteString     ; Write the dash string
    call WriteChar       ; Write the current character in AL
    call Crlf            ; Print a newline

    inc al               ; Increment AL to get the next character
    inc ecx
    cmp ecx, 255         ; If ECX is not zero, jump back to L1
    jne L1
    exit                 ; Exit the program
main ENDP
END main
```


1 - ☺
2 - ☻
3 - ♥
4 - ♦
5 - ♣
6 - ♠
7 -
8 -
9 -
10 -
11 - ♂
12 - ♀
13 -
14 - ♫
15 - ☼
16 - ►
17 - ◄
18 - ↕
19 - ‼


225 - ß
226 - Γ
227 - π
228 - Σ
229 - σ
230 - µ
231 - τ
232 - Φ
233 - Θ
234 - Ω
235 - δ
236 - ∞
237 - φ
238 - ε
239 - ∩
240 - ≡
241 - ±
242 - ≥
243 - ≤
244 - ⌠
245 - ⌡
246 - ÷
247 - ≈
248 - °
249 - ∙
250 - ·
251 - √
252 - ⁿ
253 - ²
254 - ■

Q) use Call DumpMem to print array

DumpMem (ESI=Starting OFFSET, ECX=LengthOf, EBX=Type) Writes the block of memory to the console window in hexadecimal.

```
INCLUDE Irvine32.inc
.data
COUNT = 4
arrayD SDWORD 12345678h,1A4B2000h, 3434h, 7AB9h
.code
main PROC
; Display an array using DumpMem.
mov esi, OFFSET arrayD ; starting OFFSET
mov ebx, TYPE arrayD ; doubleword = 4 bytes
mov ecx, LENGTHOF arrayD ; number of units in arrayD
call DumpMem
call DumpRegs
exit
main ENDP
```

```
Dump of offset 00566000
------------------------------
12345678  1A4B2000  00003434  00007AB9

  EAX=010FFED8  EBX=00000004  ECX=00000004  EDX=005610AA
  ESI=00566000  EDI=005610AA  EBP=010FFE8C  ESP=010FFE80
  EIP=00563679  EFL=00000202  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=0
```

Q) SetTextColor (EAX= Foreground + (Background*16))

Sets the foreground and background colors of all subsequent text output to the console.

```
TITLE SetTextColor: Background & foreground colors are passed to EAX
INCLUDE Irvine32.inc
.data
str1 BYTE "Sample string in color", 0
.code
main PROC
mov eax, yellow + (blue*16)
call SetTextColor
mov edx, OFFSET str1
call WriteString
call DumpRegs
exit
main ENDP
END main
```

```
Sample string in color
  EAX=0000001E  EBX=002FC000  ECX=00E410AA  EDX=00E46000
  ESI=00E410AA  EDI=00E410AA  EBP=001BFEFC  ESP=001BFEF0
  EIP=00E43679  EFL=00000202  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=0

C:\Users\Kainat\OneDrive\Desktop\Coal\CoalLabtask\Debug\CoalLabtask.exe (process 13632) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically cl
le when debugging stops.
Press any key to close this window . . .
```

```
Sample string in color
  EAX=0000001E  EBX=002FC000  ECX=00E410AA  EDX=00E46000
  ESI=00E410AA  EDI=00E410AA  EBP=001BFEFC  ESP=001BFEF0
  EIP=00E43679  EFL=00000202  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=0

C:\Users\Kainat\OneDrive\Desktop\Coal\CoalLabtask\Debug\CoalLabtask.exe (process 13632) exited with code
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically
le when debugging stops.
Press any key to close this window . . .
```

MsgBox (EDX=OFFSET String, EBX= OFFSET Title)

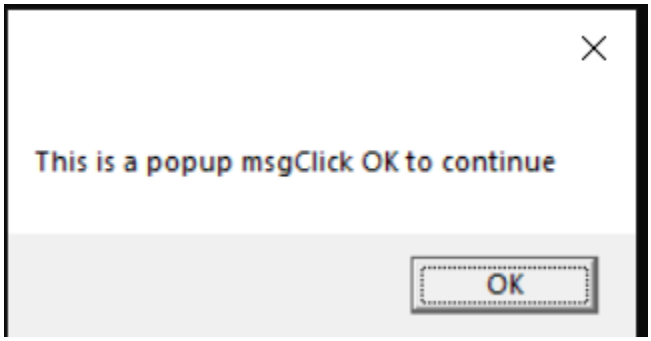
Displays a pop-up message box.

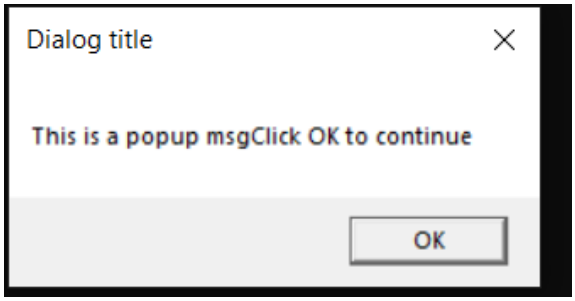
MsgBoxAsk (EDX=OFFSET String, EBX= OFFSET Title) Displays a yes/no question in a pop-up message box.

(EAX=6 YES, EAX=7 NO)

Q) MSG BOX

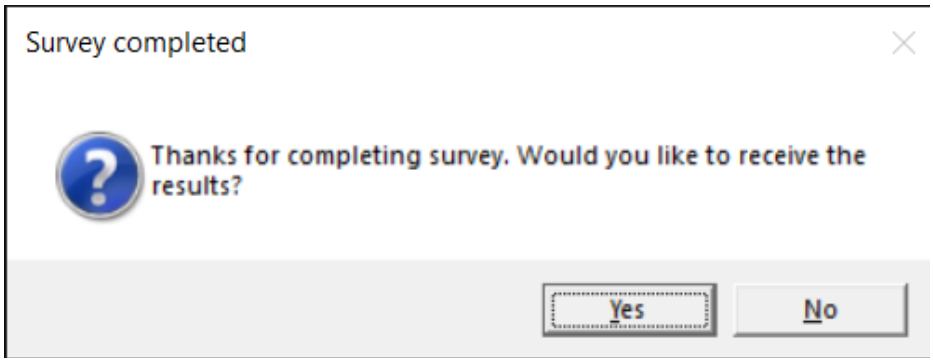
```
TITLE MSG BOX
Include Irvine32.inc
.data
caption1 BYTE "Dialog title",0
caption2 BYTE "This is a popup msg"
         BYTE "Click OK to continue",0
.code
main PROC
mov ebx, 0
mov edx, OFFSET caption2
call MsgBox
mov ebx, OFFSET caption1
mov edx, OFFSET caption2
call MsgBox
exit
main ENDP
END main
```





Q)MsgBoxAsk: Offset of question string is passed in EDX. Offset of caption is passed in EBX. Selected value is returned in EAX (IF YES equal to 6 or IFNO equal to 7)

```
TITLE MSG BOX ASK
Include Irvine32.inc
.data
caption1 BYTE "Survey completed",0
caption2 BYTE "Thanks for completing survey. "
         BYTE "Would you like to receive the results?",0
.code
main PROC
mov ebx, OFFSET caption1
mov edx, OFFSET caption2
call MsgBoxAsk
call DumpRegs
exit
main ENDP
END main
```



YES:

```
EAX=00000006  EBX=00576000  ECX=005710AA  EDX=00576011
ESI=005710AA  EDI=005710AA  EBP=0115F7D4  ESP=0115F7C8
EIP=00573674  EFL=00000246  CF=0  SF=0  ZF=1  OF=0  AF=0
```

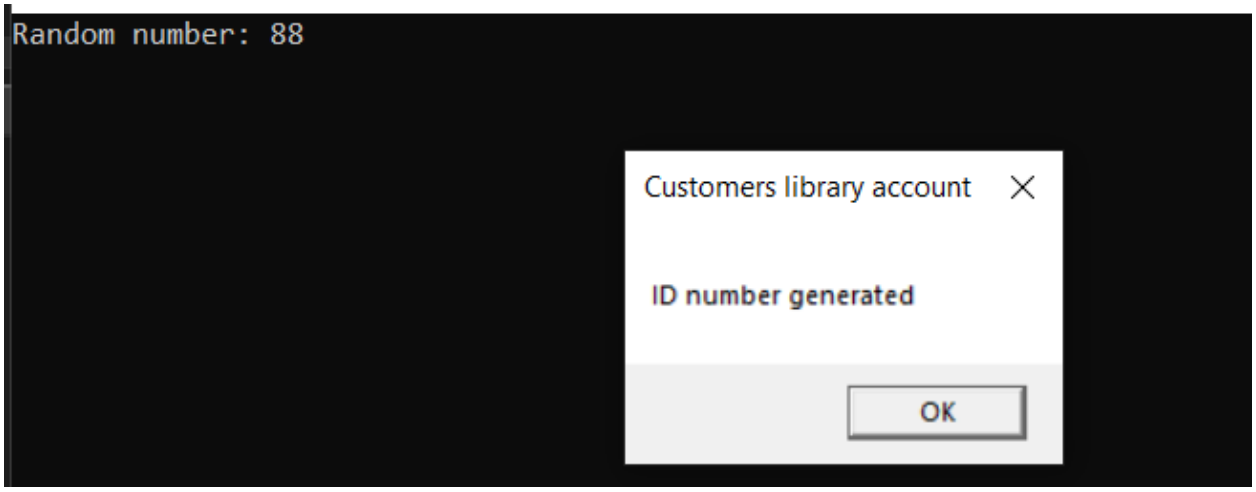
NO:

```
EAX=00000007  EBX=00576000  ECX=005710AA  EDX=00576011
ESI=005710AA  EDI=005710AA  EBP=0046FDB0  ESP=0046FDA4
EIP=00573674  EFL=00000246  CF=0  SF=0  ZF=1  OF=0  AF=0  PF=1
```

PAST PAPER QS:

Q) Msg box

```
TITLE MSG BOX
Include Irvine32.inc
.data
msg BYTE "Random number: ",0
caption1 BYTE "Customers library account",0
caption2 BYTE "ID number generated",0
.code
main PROC
call Randomize
mov eax, 100                ; Set range (0 to 99)
call RandomRange
mov edx, OFFSET msg
call WriteString
call WriteDec
call Crlf

mov ebx, OFFSET caption1
mov edx, OFFSET caption2
call MsgBox
call DumpRegs
exit
main ENDP
END main
```



Q) Looping ans summing the index.

```
TITLE Q3
Include Irvine32.inc
.data
msg1 BYTE "Enter sales count for cake type 1 ",0
msg2 BYTE "Enter sales count for cake type 2 ",0
msg3 BYTE "Enter sales count for cake type 3 ",0
type1 DWORD 4 DUP(?)
type2 DWORD 4 DUP(?)
type3 DWORD 4 DUP(?)
total1 DWORD ?
total2 DWORD ?
total3 DWORD ?
msg4 BYTE "Total for cake type 1 is: ",0
msg5 BYTE "Total for cake type 2 is: ",0
msg6 BYTE "Total for cake type 3 is: ",0
```

```
.code
main PROC
mov ebx, 0
mov ecx, 4
input_loop:
    mov edx, OFFSET msg1
    call WriteString
    call Crlf
    call ReadInt
    mov [type1+ebx], eax

    mov edx, OFFSET msg2
    call WriteString
    call Crlf
    call ReadInt
    mov [type2+ebx], eax
```

```
mov edx, OFFSET msg3
call WriteString
call Crlf
call ReadInt
mov [type3+ebx], eax

add ebx, 4
loop input_loop
```

```
mov eax, 0
mov esi, OFFSET type1
mov ecx, 4
L1:
add eax, [esi]
add esi,4
loop L1
mov [total1], eax
mov edx, OFFSET msg4
call WriteString
mov eax, [total1]
call WriteDec
call Crlf
```

```
mov eax, 0
mov esi, OFFSET type2
mov ecx, 4
L2:
add eax, [esi]
add esi,4
loop L2
mov [total2], eax
mov edx, OFFSET msg5
call WriteString
mov eax, [total2]
call WriteDec
call Crlf
```

```
mov eax, 0
mov esi, OFFSET type3
mov ecx, 4
L3:
add eax, [esi]
add esi,4
loop L3
mov [total3], eax
mov edx, OFFSET msg6
call WriteString
mov eax, [total3]
call WriteDec
call Crlf
```

```
    call Crlf
call DumpRegs
exit
main ENDP
END main
```

```
10
Enter sales count for cake type 2
20
Enter sales count for cake type 3
30
Enter sales count for cake type 1
10
Enter sales count for cake type 2
20
Enter sales count for cake type 3
30
Enter sales count for cake type 1
10
Enter sales count for cake type 2
20
Enter sales count for cake type 3
30
Total for cake type 1 is: 40
Total for cake type 2 is: 80
Total for cake type 3 is: 120
```

Unsigned	ZF	CF
destination < source	0	1
destination > source	0	0
destination = source	1	0

Signed	FLAGS
destination < source	SF != OF
destination > source	SF == OF
destination = source	ZF = 1

Q) Determine whether destination is less/greater or equal to source.

```
TITLE UNSIGNED
Include Irvine32.inc
.data
msg1 BYTE "Destination is equal to the source",0
msg2 BYTE "Destination is less than the source",0
msg3 BYTE "Destination is greater than the source",0
source DWORD 10
_dest DWORD 20
.code
main PROC
mov eax, 0
mov eax, _dest
mov ebx, source
cmp eax, ebx
jz Lequal
jc Lless
jmp LGreater

Lless:
mov edx, OFFSET msg2
call WriteString
jmp _exit
Lequal:
mov edx, OFFSET msg1
call WriteString
jmp _exit
LGreater:
mov edx, OFFSET msg3
call WriteString
_exit:
call DumpRegs
exit
main ENDP
END main
```

```
Destination is greater than the source
  EAX=00000014  EBX=0000000A  ECX=00AD10AA  EDX=00AD6047
  ESI=00AD10AA  EDI=00AD10AA  EBP=012FFE10  ESP=012FFE04
  EIP=00AD369F  EFL=00000202  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=0
```

Q) Signed integers

```
TITLE SIGNED INTEGER COMPARISON
Include Irvine32.inc
.data
msg1 BYTE "Destination is equal to the source", 0
msg2 BYTE "Destination is less than the source", 0
msg3 BYTE "Destination is greater than the source", 0
source DWORD -10          ; Source is -10
_dest DWORD -20           ; Destination is -20
.code
main PROC
    mov eax, 0
    mov eax, _dest        ; Move destination to EAX
    mov ebx, source       ; Move source to EBX
    cmp eax, ebx          ; Compare signed integers
    jz Lequal             ; Jump if equal

    ; Check signed comparison
    js Lcond1             ; Jump if the sign flag is set (indicates negative result)

    ; No sign flag set, check if overflow occurred
    jo LGreater           ; Jump if overflow occurred
    jl Lless              ; Jump if destination is less (signed comparison)
    jg LGreater           ; Jump if destination is greater (signed comparison)
```

```
Lcond1:
    jo LGreater           ; Jump to greater if overflow occurred
    jl Lless              ; Jump if less
    jg LGreater           ; Jump if greater

Lequal:
    mov edx, OFFSET msg1  ; Load "equal" message
    call WriteString
    jmp _exit

Lless:
    mov edx, OFFSET msg2  ; Load "less than" message
    call WriteString
    jmp _exit

LGreater:
    mov edx, OFFSET msg3  ; Load "greater than" message
    call WriteString
    jmp _exit

_exit:
    call DumpRegs
    exit
main ENDP
END main
```

```
Destination is less than the source
  EAX=FFFFFFEC  EBX=FFFFFFF6  ECX=00D510AA  EDX=00D56023
  ESI=00D510AA  EDI=00D510AA  EBP=010FFC80  ESP=010FFC74
  EIP=00D536AB  EFL=00000202  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=0
```

Q) MUL instruction:

```
TITLE MUL Instruction with JC (Jump if Carry)
Include Irvine32.inc

.data
byteVal BYTE 5            ; 8-bit multiplicand
wordVal WORD 1000         ; 16-bit multiplicand
dwordVal DWORD 100000     ; 32-bit multiplicand
msgNoCarry BYTE "Result fits in destination register.", 0
msgCarry BYTE "Carry flag is set, result overflows into higher registers.", 0

.code
main PROC
    ; Case 1: 8-bit multiplication (AL)
    mov al, 10           ; Load 8-bit multiplicand into AL
    mov bl, byteVal      ; Load 8-bit value into BL
    mul bl               ; AL = AL * BL, result is stored in AX
    jc carry_case        ; Jump if Carry Flag is set
    jmp NoCarry         ; No carry, result fits in AX
    call DumpRegs        ; Display register values

    ; Case 2: 16-bit multiplication (AX)
    mov ax, 200          ; Load 16-bit multiplicand into AX
    mov bx, wordVal      ; Load 16-bit value into BX
    mul bx               ; AX = AX * BX, result is stored in DX:AX
    jc carry_case        ; Jump if Carry Flag is set
    jmp NoCarry         ; No carry, result fits in DX:AX
    call DumpRegs        ; Display register values
```

```
; Case 3: 32-bit multiplication (EAX)
mov eax, 50000        ; Load 32-bit multiplicand into EAX
mov ebx, dwordVal     ; Load 32-bit value into EBX
mul ebx               ; EAX = EAX * EBX, result is stored in EDX:EAX
jc carry_case         ; Jump if Carry Flag is set
jmp NoCarry          ; No carry, result fits in EDX:EAX
call DumpRegs         ; Display register values

jmp _exit
```

```
carry_case:
    mov edx, OFFSET msgCarry
    call WriteString
    jmp _exit

NoCarry:
    mov edx, OFFSET msgNoCarry
    call WriteString

_exit:
    call DumpRegs
    exit
main ENDP
END main
```

DIV AND MUL INSTRUCTION:

Dividend	Divisor	Quotient	Remainder
AX	reg/mem8	AL	AH
DX:AX	reg/mem16	AX	DX
EDX:EAX	reg/mem32	EAX	EDX

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

Q) Tell whether number is even or odd using DIV instruction.

```
TITLE Even or Odd Check using DIV and JC (Jump if Carry)
INCLUDE Irvine32.inc
.data
number DWORD 10           ; Number to check
msgEven BYTE "The number is even.", 0
msgOdd BYTE "The number is odd.", 0
.code
main PROC
    ; Load the number to check
    mov eax, number       ; Load the number into EAX register
    ; Clear EDX before division (required for division in x86)
    xor edx, edx          ; Clear EDX to ensure no leftover data affects the division
    ; Divide by 2
    mov ecx, 2            ; Divisor is 2
    div ecx           ; EAX / ECX; Quotient in EAX, remainder in EDX
```

```
    ; Check remainder in EDX
    cmp edx, 0            ; If remainder is 0, the number is even
    call DumpRegs
    je even_case          ; Jump to even_case if zero
    jmp odd_case          ; Otherwise, jump to odd_case

even_case:
    mov edx, OFFSET msgEven
    call WriteString
    jmp end_program       ; Skip the rest and go to end_program

odd_case:
    mov edx, OFFSET msgOdd
    call WriteString

end_program:
    exit
main ENDP
END main
```

```
EAX=00000005  EBX=00D08000  ECX=00000002  EDX=00000000
ESI=007010AA  EDI=007010AA  EBP=00EFFC24  ESP=00EFFC18
EIP=00703676  EFL=00000246  CF=0  SF=0  ZF=1  OF=0  AF=0  PF=1

The number is even.
```

See the remainder in EDX is 0.

Question No 2(b): Convert the following Code in x86 assembly. **Points]**

```
int myarray[100];
while(j>=0 && j<=100)
{
     myarray[j+1] = myarray[j];
     j = j-1;
}
```

```
TITLE WHILELOOP
INCLUDE Irvine32.inc
.data
    myarray DWORD 100 DUP(0)       ; Reserve space for 100 integers initialized to 0
    j DWORD 100                     ; Initialize `j` with 100 as in C code
.code
main PROC
    mov ecx, j                      ; Load `j` value into ECX
while_loop:
    cmp ecx, 0                      ; Check if j >= 0
    jl end_while                    ; If j < 0, exit loop
    cmp ecx, 100                    ; Check if j <= 100
    jg end_while                    ; If j > 100, exit loopS
    ; Copy myarray[j] to myarray[j+1]
    mov eax, ecx                    ; EAX = j (index)
    mov ebx, OFFSET myarray         ; EBX = base address of myarray
    mov edx, [ebx + eax*4]          ; Load myarray[j] into EDX (4 bytes per integer)
    inc eax
    mov [ebx + eax*4], edx      ; Store EDX in myarray[j+1]
    dec eax
    ; Decrement j
    dec ecx                         ; j = j - 1
    jmp while_loop                  ; Repeat loop

end_while:
    call DumpRegs                   ; Display register values for verification
    exit
main ENDP
END main
```

Reverse a string using stack:

```
INCLUDE Irvine32.inc
.data
aName BYTE "Computer Architecture",0
nameSize = ($ - aName) - 1
.code
main PROC
; Push the name on the stack.
mov ecx,nameSize
mov esi,0
L1: movzx eax,aName[esi] ; get character
push eax ; push on stack
inc esi
loop L1
; Pop the name from the stack, in reverse,
; and store in the aName array.
mov ecx,nameSize
mov esi,0
L2: pop eax ; get character
mov aName[esi],al ; store in string
inc esi
loop L2
; Display the name.
mov edx,OFFSET aName
call WriteString
call Crlf
exit
main ENDP
END main
```

Algorithm for Checking Valid Parentheses in Assembly:

1. Initialization:
    - Calculate the length of the `expression` string.
    - Initialize `ECX` with the length of `expression` to serve as the loop counter.
    - Set `ESI` to `0` to use as an index for accessing each character in `expression`.
    - Set `EDI` to `0` to use as a counter to track the balance of parentheses.
2. Loop Through Each Character in the Expression:
    - Load the character at the `ESI` index from `expression` into `AL`.
    - Check if the character is an opening parenthesis `'('`:
        - If true, jump to the `save` section.
        - In `save`, push `EAX` (which contains `'('`) onto the stack, and increment `EDI` (balance counter) by `1`. Then, jump to `next` to continue to the next character.
    - If the character is a closing parenthesis `')'`:
        - If true, jump to `_test`.
        - In `_test`, check if `EDI` (balance counter) is `0` (indicating no matching opening parenthesis).
            - If `EDI` is `0`, jump to the `error` section.
            - Otherwise, pop the stack to retrieve the last pushed `'('`, decrement `EDI` by `1` (indicating one matching pair is closed). Then, jump to `next` to continue to the next character.
    - If the character is neither `'('` nor `')'`, jump to `next` to skip it.
3. Continue to the Next Character:
    - Increment `ESI` to move to the next character in `expression`.
    - Decrement `ECX` (loop counter) and repeat the loop until `ECX` reaches `0` (all characters processed).
4. Final Balance Check:
    - After the loop, check if `EDI` is `0`.
        - If `EDI` is not `0`, jump to the `error` section because there are unmatched opening parentheses.

        - If `EDI` is `0`, all parentheses are balanced, and the program can proceed to display a valid message.
5. Display Result:
    - If all parentheses are balanced, load the `validMessage` into `EDX` and call `WriteString` to print "Parentheses are balanced".
    - If there is an imbalance (unmatched parentheses), load the `errorMessage` into `EDX` and call `WriteString` to print "Parentheses are not balanced".
6. End the Program:
    - Print a newline using `Crlf` and exit the program.

```
TITLE Valid Parentheses Check (ValidParentheses.asm)
INCLUDE Irvine32.inc

.data
    expression BYTE "((2+3)-(4+4))", 0  ; Expression to check
    _length = ($ - expression) - 1       ; Length of the expression string
    validMessage BYTE "Parentheses are balanced", 0
    errorMessage BYTE "Parentheses are not balanced", 0

.code
main PROC
    ; Initialize
    mov ecx, _length       ; Set loop counter to length of expression
    mov esi, 0             ; Index for accessing each character in expression
    mov edi, 0             ; Initialize stack balance counter
```

```
L1:
    mov al, expression[esi]      ; Load character from expression

    cmp al, '('        ; Check if character is '('
    je save            ; If '(', jump to save

    cmp al, ')'       ; Check if character is ')'
    je _test          ; If ')', jump to test

    jmp next        ; Skip other characters

save:
    push eax        ; Push '(' onto the stack
    inc edi         ; Increase balance counter
    jmp next

_test:
    cmp edi, 0      ; Check if there's an unmatched ')'
    je error        ; If balance counter is zero, report error

    pop eax         ; Pop the top of the stack
    dec edi         ; Decrease balance counter
    cmp al, '('     ; Check if matching '(' was popped
    je next         ; If match, continue
    jmp error       ; Otherwise, error

next:
    inc esi      ; Move to the next character
    loop L1      ; Repeat until end of expression
```

```
    ; Final check to see if all parentheses were matched
    cmp edi, 0   ; If counter is not zero, report error
    jne error

    ; Display valid message
    mov edx, OFFSET validMessage
    call WriteString
    jmp endProg

error:
    ; Display error message
    mov edx, OFFSET errorMessage
    call WriteString

endProg:
    call Crlf
    exit
main ENDP
END main
```

```
Parentheses are balanced
```

Swap:

```
TITLE Swap Procedure using Stack (SwapValues.asm)
INCLUDE Irvine32.inc

.data
    var1 DWORD 10          ; First variable
    var2 DWORD 20          ; Second variable
    resultMessage BYTE "After swapping:", 0

.code
main PROC
    ; Display initial values
    mov eax, var1
    call WriteInt
    call Crlf
    mov eax, var2
    call WriteInt
    call Crlf
    ; Call the swap procedure
    call SwapValues
    ; Display result message
    mov edx, OFFSET resultMessage
    call WriteString
    call Crlf

    ; Display swapped values
    mov eax, var1
    call WriteInt
    call Crlf
    mov eax, var2
    call WriteInt
    call Crlf

    exit
main ENDP
```

```
;---------------------------------------
; SwapValues Procedure
; Swaps the values of var1 and var2 using
; the stack.
;---------------------------------------
SwapValues PROC
    ; Push var1 and var2 onto the stack
    mov eax, var1        ; Load var1 into EAX
    push eax             ; Push var1 onto the stack

    mov eax, var2        ; Load var2 into EAX
    push eax             ; Push var2 onto the stack

    ; Pop them in reverse order to swap
    pop eax              ; Pop the top of the stack into EAX (var2's value)
    mov var1, eax        ; Store the value of var2 into var1

    pop eax              ; Pop the next value (var1's original value) into EAX
    mov var2, eax        ; Store the value of var1 into var2

    ret
SwapValues ENDP

END main
```

WHEN NOT TO PUSH REGISTERS:

Purpose of Pushing and Popping Registers:

- The push instruction saves the current value of a register onto the stack, while pop restores it from the stack. This is useful when a register needs to be used temporarily without losing its original value, typically in subroutines where preserving the state is important.

When Not to Push and Pop Registers:

- If you are not concerned with preserving the original value of a register (like eax here), then there is no need to push and pop it. In this example, the eax register is used to hold the result of the sum, so its original value isn't needed after the calculations.
- Pushing and popping unnecessarily adds overhead, which can slow down the execution slightly, especially in performance-critical code.

```
SumOf PROC    ; sum of three integers
    push eax  ; 1
    add eax,ebx   ; 2
    add eax,ecx   ; 3
    pop eax   ; 4
    ret
SumOf ENDP
```

ArraySum:

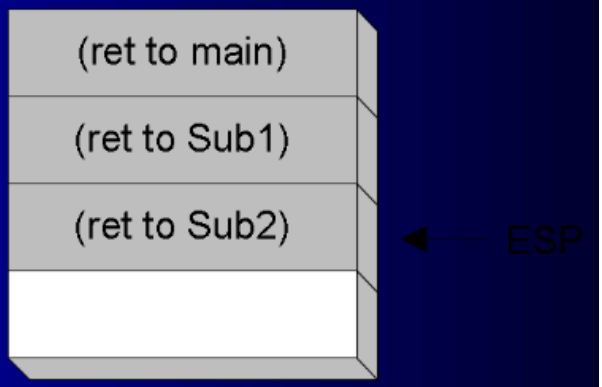
This version of ArraySum returns the sum of any doubleword array whose address is in SI. The sum is returned in EAX:

```
ArraySum PROC
; Receives: SI points to an array of doublewords,
;   ECX = number of array elements.
; Returns: EAX = sum
;-----------------------------------------------------
    mov eax,0   ; set the sum to zero

L1: add eax,[si]    ; add each integer to sum
    add si,4    ; point to next integer
    loop L1 ; repeat for array size

    ret
ArraySum ENDP
```

Nested Procedure Calls

```
main PROC
    .
    .
    call Sub1
    exit
main ENDP

Sub1 PROC
    .
    .
    call Sub2
    ret
Sub1 ENDP

Sub2 PROC
    .
    .
    call Sub3
    ret
Sub2 ENDP

Sub3 PROC
    .
    .
    ret
Sub3 ENDP
```

By the time Sub3 is called, the stack contains all three return addresses:



Q) Write a program to count the digits of a given number lets say n = 500 using recursion in MASM.

```
TITLE Count Digits
INCLUDE Irvine32.inc
.data
msg BYTE "The digits in n are: ",0
n DWORD 5000
.code
main PROC
mov edx, 0
mov eax, n
mov ecx, 0
call Countdigits
mov eax, ecx
mov edx, OFFSET msg
call WriteString
call WriteDec
call Crlf
exit
main ENDP
```

```
Countdigits PROC
cmp eax, 0
je basecase
mov ebx, 10
div ebx ; eax = eax/10
inc ecx
call Countdigits
basecase:
ret
Countdigits ENDP
END main
```

```
The digits in n are: 4
```

Q) Convert from decimal to binary using recursion.

```
TITLE Decimal to Binary Conversion
INCLUDE Irvine32.inc

.data
msg BYTE "Binary : ", 0
n DWORD 23
binarynum DWORD 0

.code
main PROC
mov eax, n
mov binarynum, 0
call DecimaltoBinary

mov edx, OFFSET msg
call WriteString
mov eax, binarynum
call WriteBin
call Crlf
exit
main ENDP
```

```
DecimaltoBinary PROC
cmp eax, 0
je Basecase

mov ebx, 2                  ; Divisor (for binary conversion)
mov edx, 0                  ; Clear EDX for the division
div ebx                     ; EAX = EAX / 2, EDX = remainder (binary digit)
shl binarynum, 1            ; Shift the binarynum left by 1 (make space for the new bit)
or binarynum, edx           ; Set the least significant bit (LSB) to the remainder (0 or 1)

call DecimaltoBinary        ; Recursive call to divide by 2 again

Basecase:
ret                         ; Return from the recursion
DecimaltoBinary ENDP

END main
```

```
Binary : 0000 0000 0000 0000 0000 0000 0001 1101
```

Logic behind using shl and or instructions:

Initial Setup:

- `n = 23` (The decimal number to convert to binary)
- `binarynum = 0` (Initial binary number is set to zero)

Step 1: First Division (`23 / 2`)

- Perform the division:
  `23 / 2 = 11` (quotient) with a remainder of `1`.
  After division, we have:
  - `eax = 11` (quotient)
  - `edx = 1` (remainder)
- Now, let's perform the bitwise operations:
  `shl binarynum, 1`:
  - `binarynum = 0` (initial value)
  - Shift left by one bit: `binarynum << 1` results in `binarynum = 0`.
    After shifting, the position for the new bit is prepared, but the value of `binarynum` is still `0`.
- `or binarynum, edx`:
  - `edx = 1` (the remainder of the division).
  - Perform the bitwise OR: `binarynum = 0 OR 1 = 1`.
    Now, `binarynum = 1`.
- At this point, the binary representation of the first division remainder is stored as `binarynum = 1`.

Step 2: Second Division (`11 / 2`)

- Perform the division:
  `11 / 2 = 5` (quotient) with a remainder of `1`.
  After division, we have:

    - `eax = 5` (quotient)
    - `edx = 1` (remainder)
- Now, let's perform the bitwise operations again:
  `shl binarynum, 1`:
    - `binarynum = 1` (current value).
    - Shift left by one bit: `binarynum = 1 << 1 = 10` (binary).
      So, `binarynum = 2` (in decimal).
- `or binarynum, edx`:
    - `edx = 1` (the remainder of the division).
    - Perform the bitwise OR: `binarynum = 10 OR 1 = 11`.
      Now, `binarynum = 11`.
- At this point, the binary representation of the first two remainders is stored as `binarynum = 11`.

Step 3: Third Division (`5 / 2`)

- Perform the division:
  `5 / 2 = 2` (quotient) with a remainder of `1`.
  After division, we have:
    - `eax = 2` (quotient)
    - `edx = 1` (remainder)
- Let's perform the bitwise operations again:
  `shl binarynum, 1`:
    - `binarynum = 11` (current value).
    - Shift left by one bit: `binarynum = 11 << 1 = 110` (binary).
      So, `binarynum = 6` (in decimal).
- `or binarynum, edx`:
    - `edx = 1` (the remainder of the division).
    - Perform the bitwise OR: `binarynum = 110 OR 1 = 111`.
      Now, `binarynum = 7`.
- At this point, the binary representation of the first three remainders is stored as `binarynum = 111`.

Step 4: Fourth Division (2 / 2)

- Perform the division:
  `2 / 2 = 1` (quotient) with a remainder of `0`.
  After division, we have:
  - `eax = 1` (quotient)
  - `edx = 0` (remainder)
- Let's perform the bitwise operations again:
  `shl binarynum, 1`:
  - `binarynum = 111` (current value).
  - Shift left by one bit: `binarynum = 111 << 1 = 1110` (binary).
    So, `binarynum = 14` (in decimal).
- `or binarynum, edx`:
  - `edx = 0` (the remainder of the division).
  - Perform the bitwise OR: `binarynum = 1110 OR 0 = 1110`.
    Now, `binarynum = 14`.
- At this point, the binary representation of the first four remainders is stored as `binarynum = 1110`.

Step 5: Fifth Division (1 / 2)

- Perform the division:
  `1 / 2 = 0` (quotient) with a remainder of `1`.
  After division, we have:
  - `eax = 0` (quotient)
  - `edx = 1` (remainder)
- Let's perform the bitwise operations again:
  `shl binarynum, 1`:
  - `binarynum = 1110` (current value).

    - Shift left by one bit: `binarynum = 1110 << 1 = 11100` (binary).
      So, `binarynum = 28` (in decimal).
- `or binarynum, edx`:
    - `edx = 1` (the remainder of the division).
    - Perform the bitwise OR: `binarynum = 11100 OR 1 = 11101`.
      Now, `binarynum = 29`.
- At this point, the binary representation of the first five remainders is stored as `binarynum = 11101`.

Final Step: Termination

- At this point, `eax = 0` (since the quotient is now 0), so the recursion terminates.

Final Binary Number:

- The binary representation of `23` is stored in `binarynum = 11101`, which is the correct binary representation of the decimal number `23`.

```
;-----------------------------------------------------
Str_trim PROC USES eax ecx edi,
     pString:PTR BYTE,          ; points to string
     char:BYTE                  ; char to remove
;
; Remove all occurrences of a given delimiter character from
; the end of a string.
; Returns: nothing
;-----------------------------------------------------
     mov  edi,pString           ; prepare to call Str_length
     INVOKE Str_length,edi      ; returns the length in EAX
     cmp  eax,0                 ; is the length equal to zero?
     je   L3                    ; yes: exit now
     mov  ecx,eax               ; no: ECX = string length
     dec  eax
     add  edi,eax               ; point to last character


L1:  mov  al,[edi]              ; get a character
     cmp  al,char               ; is it the delimiter?
     jne  L2                    ; no: insert null byte
     dec  edi                   ; yes: keep backing up
     loop L1                    ; until beginning reached

L2:  mov  BYTE PTR [edi+1],0    ; insert a null byte
L3:  ret
Stmr_trim ENDP
```

Table 9-4 Testing the Str_trim Procedure with a # Delimiter Character

Input String	Expected Modified String
"Hello##"	"Hello"
"#"	"" (empty string)
"Hello"	"Hello"
"H"	"H"
"#H"	"#H"

Detailed Description

Let us carefully examine *Str_trim*. The algorithm starts at the end of the string and scans backwards, looking for the first nondelimiter character. When it finds one, a null byte is inserted into the string just after the character position:

```
ecx = length(str)
if length(str) > 0 then
     edi = length - 1
     do while ecx > 0
       if str[edi] ≠ delimiter then
           str[edi+1] = null
           break
       else
           edi = edi - 1
       end if
       ecx = ecx - 1
     end do
else
     str[edi+1] = null
```

```
INCLUDE Irvine32.inc
.data
target BYTE "AAEBDCFBBC", 0
freqTable DWORD 256 DUP(0)
alphabet BYTE "ABCDEFGHIJKLMNOPQRSTUVWXYZ",0

.code
Get_frequencies PROC targ:DWORD, freq :DWORD
    cld
    mov esi, targ
    mov edi, freq
    mov ecx,0

count:
    mov al, [esi + ecx]
    test al, al
    jz done

    movzx eax, al       ;ascii val
    inc dword ptr [edi + eax*4]

    inc ecx
    jmp count
```

```
done:
    ret
Get_frequencies ENDP
```

```
main PROC
    INVOKE Get_frequencies, ADDR target, ADDR freqTable
    mov edx,offset target
    call writestring
    call crlf
    mov edi ,offset target
    mov ecx, 0
    mov esi, offset alphabet
    print:
        mov eax, [freqTable + ecx*4]
        cmp eax, 0
        je skip
        mov al, [esi]
        call WriteChar
        inc esi
        mov eax,':'
        call Writechar
        mov eax, [freqTable + ecx*4]
        call WriteDec
        mov al,' '
        call writechar
```

```
    skip:
        inc ecx
        inc edi
        cmp ecx, 256
        jl print

    exit
main ENDP
END main
```

```
AAEBDCFBBC
A:2 B:3 C:2 D:1 E:1 F:1
```

9.3.5 Str_ucase Procedure

The **Str_ucase** procedure converts a string to all uppercase characters. It returns no value. When you call it, pass the offset of a string:

```
INVOKE Str_ucase, ADDR myString
```

Here is the procedure implementation:

```
Str_ucase PROC USES eax esi,
    pString:PTR BYTE
; Converts a null-terminated string to uppercase.
; Returns: nothing
;-----------------------------------------
    mov   esi,pString

L1:
    mov   al,[esi]                    ; get char
    cmp   al,0                        ; end of string?
    je    L3                          ; yes: quit
    cmp   al,'a'                      ; below "a"?
    jb    L2
    cmp   al,'z'                      ; above "z"?
    ja    L2
    and   BYTE PTR [esi],11011111b    ; convert the char
L2: inc   esi                         ; next char
    jmp   L1
L3: ret
Str_ucase ENDP
```