National University of Computer & Emerging Sciences, Karachi.
FAST School of Computing,
Mid Term I, Fall 2021.
October 13, 2021, 9:00 am – 10:00 am.

Course Code: EE 2003	**Course Name:** Computer Organization and Assembly Language
Instructors: Dr. Nouman M Durrani, Shoaib Rauf, Aashir Mahboob, Aamir Ali, and Qurat ul Ain	
Student's Roll No:	**Section:**

Instructions:

- Except for your Roll No and Section, DO NOT SOLVE anything on this paper.
- Return the question paper.
- Read each question completely before answering it. There are **3 questions on 2 pages (1 Sheet).**
- In case of any ambiguity, you may make an assumption. But your assumption should not contradict any statement in the question paper.
- All the answers must be solved according to the SEQUENCE given in the question paper, otherwise, points will be deducted.
- This paper is subjective.
- Where asked for values, only provide the **hex-decimal** values.
- Problems needing iterations should be coded using iterative instructions. No points will be awarded otherwise.

Time Allowed: 60 minutes. **Maximum Points**: 32 points

==========

Q. No. 1 Briefly answer each of the following: [8 x 2 = 16 points]

(i) Explain the difference between the direct-offset and indexed operands with examples.

4.1.8 Direct-Offset Operands

You can add a displacement to the name of a variable, creating a direct-offset operand. This lets you access memory locations that may not have explicit labels. Let's begin with an array of bytes named **arrayB**:

```
arrayB  BYTE 10h,20h,30h,40h,50h
```

If we use MOV with **arrayB** as the source operand, we automatically move the first byte in the array:

```
mov  al,arrayB                ; AL = 10h
```

We can access the second byte in the array by adding 1 to the offset of **arrayB**:

```
mov  al,[arrayB+1]            ; AL = 20h
```

The third byte is accessed by adding 2:

```
mov  al,[arrayB+2]            ; AL = 30h
```

4.4.3 Indexed Operands

An *indexed operand* adds a constant to a register to generate an effective address. Any of the 32-bit general-purpose registers may be used as index registers. There are different notational forms permitted by MASM (the brackets are part of the notation):

```
constant[reg]
[constant + reg]
```

The first notational form combines the name of a variable with a register. The variable name is translated by the assembler into a constant that represents the variable's offset. Here are examples that show both notational forms:

arrayB[esi]	[arrayB + esi]
arrayD[ebx]	[arrayD + ebx]

Indexed operands are ideally suited to array processing. The index register should be initialized to zero before accessing the first array element:

```
.data
arrayB BYTE 10h,20h,30h
.code
mov  esi,0
mov  al,[arrayB + esi]          ; AL = 10h
```

(ii) In which case, we use MOVSX instruction? Explain one example of MOVSX in which a smaller value is moved into a larger destination.

MOVSX Instruction

The MOVSX instruction (move with sign-extend) copies the contents of a source operand into a destination operand and sign-extends the value to 16 or 32 bits. This instruction is only used with signed integers. There are three variants:

```
MOVSX  reg32,reg/mem8
MOVSX  reg32,reg/mem16
MOVSX  reg16,reg/mem8
```

```
.data
byteVal BYTE 10001111b
.code
movsx  ax,byteVal          ; AX = 1111111110001111b
```

(iii) Discuss the Signed and Unsigned integers with reference to the hardware viewpoint.

Signed/Unsigned Integers: Hardware Viewpoint

- All CPU instructions operate exactly the same on signed and unsigned integers
- The CPU cannot distinguish between signed and unsigned integers
- YOU, the programmer, are solely responsible for using the correct data type with each instruction

(iv) Discuss the term label as an identifier and directive with examples.

Label

A *label* is an identifier that acts as a place marker for instructions and data. A label placed just before an instruction implies the instruction's address. Similarly, a label placed just before a variable implies the variable's address.

Data Labels A data label identifies the location of a variable, providing a convenient way to reference the variable in code. The following, for example, defines a variable named count:

```
count  DWORD 100
```

The assembler assigns a numeric address to each label. It is possible to define multiple data items following a label. In the following example, array defines the location of the first number (1024). The other numbers following in memory immediately afterward:

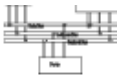
```
array  DWORD 1024, 2048
       DWORD 4096, 8192
```

Code Labels A label in the code area of a program (where instructions are located) must end with a colon (:) character. Code labels are used as targets of jumping and looping instructions. For example, the following JMP (jump) instruction transfers control to the location marked by the label named **target**, creating a loop:

```
target:
    mov   ax,bx
    ...
    jmp   target
```

(v) The LOOP instruction creates a counting loop. What happens when a loop instruction is encountered?

LOOP Instruction

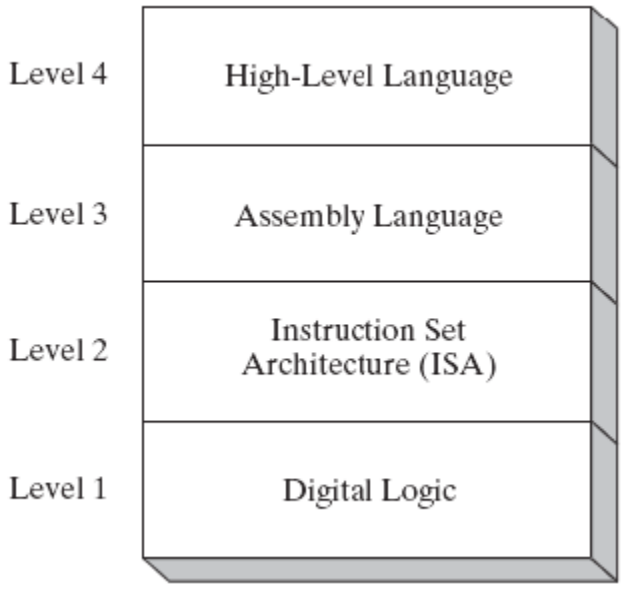


- The LOOP instruction creates a counting loop
- Syntax: LOOP *target*
- Logic:
  - ECX ← ECX - 1
  - if ECX != 0, jump to *target*

(vi) Why the concept of virtual machines is important in computer organization? At which level does assembly language appear at the virtual machine level? Consider L1 as the lowest level.

- Virtual Machine Concept:

  – An effective way to explain how a computer's hardware and software are related is called the *virtual machine concept.*



(vii) How do we load and store the status flag bits? Give example instructions.

4.1.6 LAHF and SAHF Instructions

The LAHF (load status flags into AH) instruction copies the low byte of the EFLAGS register into AH. The following flags are copied: Sign, Zero, Auxiliary Carry, Parity, and Carry. Using this instruction, you can easily save a copy of the flags in a variable for safekeeping:

```
.data
saveflags BYTE ?
.code
lahf                          ; load flags into AH
mov saveflags,ah              ; save them in a variable
```

The SAHF (store AH into status flags) instruction copies AH into the low byte of the EFLAGS register. For example, you can retrieve the values of flags saved earlier in a variable:

```
mov ah,saveflags              ; load saved flags into AH
sahf                          ; copy into Flags register
```

(viii) Compare the following x86 processors' mode of operations:

(a) Real address mode

(b) Protected address mode

- Real-Address mode (original mode provided by 8086)
  - Only 1 MB of memory can be addressed, from 0 to FFFFF (hex)
  - Programs can access any part of main memory
  - MS-DOS runs in real-address mode
- Implements the programming environment of the Intel 8086 processor
- This mode is available in Windows 98, and can be used to run an MS-DOS program that requires direct access to system memory and hardware devices
- Programs running in real-address mode can cause the operating system to crash (stop responding to commands)
- Protected mode (introduced with the 80386 processor)
  - Each program can address a maximum of 4 GB of memory
  - The operating system assigns memory to each running program
  - Programs are prevented from accessing each other's memory *(segments)*
  - Native mode used by Windows NT, 2000, XP, and Linux

Q. No. 2 Consider the following initialization: [3 + 3 + 2 = 8 points]

```
X1   WORD  0E342H, 4 DUP(0Eh)
```

(i) Assign proper physical addresses (using a real address mode) to each byte stored in the data segment, and draw a memory map. (Assume DS= 2FF0h, and the starting offset is 2304h).

(ii) For the above data definition directives, give the content of the destination register after execution of each of the following instructions:

Address	value
32204	42h
32205	E3h
32206	0Eh
32207	00
32208	0Eh
32209	00
3220A	0E
3220B	00
3220C	0E
3220D	00

```
MOV  EAX, DWORD PTR X1          ;(a) EAX = 000EE342h
MOV  BL, SIZEOF X1              ;(b) BL = 0Ah
MOV ESI, 4
MOV   BX, [X1+ESI]              ;(c) BX = 000Eh
```

(iii) Where indicated, write down the values of the Carry, Sign, Zero, and Overflow flags after the execution of each instruction below:

```
MOV AX,7FF0H
ADD AL,10H      ; (a) CF = 1      SF = 0      ZF = 1 OF =0
```

```
ADD AH,1          ; (b) CF = 0 SF = 1    ZF = 0     OF =1
```

Q. No. 3 Write assembly language programs for the following problems: [4 + 4 = 8 points]

(i) Declare two variables "val1" of type BYTE and "val2" of type WORD, initialized with hexadecimal values 79h and 100h respectively. Find the multiplication of these variables using a loop instruction and store the result into a third variable "val3" of type DWORD.

```
INCLUDE Irvine32.inc
.data
val1 BYTE 79h
val2 WORD 100h
val3 DWORD ?
.code
main PROC
      MOVZX EBX, val2
      MOVZX ECX, val1
      MOV EAX, 0
      L1:
            ADD EAX,EBX
      LOOP L1
      MOV val3,EAX
      call dumpregs
exit
main ENDP
END main
```

(ii) The Lucas sequence has the same recursive relationship as the Fibonacci sequence, where each term is the sum of the previous two terms, but with different starting values. If the starting values are 2 and 1, write an assembly language program that finds and stores the missing elements in the following series into a WORD type variable X1:

2, 1, 3, 4, 7,__, __, __, __, __ .

Hint: X1 word 2, 1, 3, 4, 7, 5 DUP(?)

```
INCLUDE Irvine32.inc
.code
main PROC
      MOV EBX, 2
      MOV EDX, 1
      MOV EAX, 2
      CALL WRITEDEC
      MOV ECX, 10 - 1
      L1:
            MOV AL, ','
            CALL WRITECHAR
```

```
            MOV EAX,EDX
            CALL WRITEDEC
            ADD EAX, EBX
            MOV EBX, EDX
            MOV EDX, EAX

      LOOP L1
exit
main ENDP
END main
```

Best of LUCK