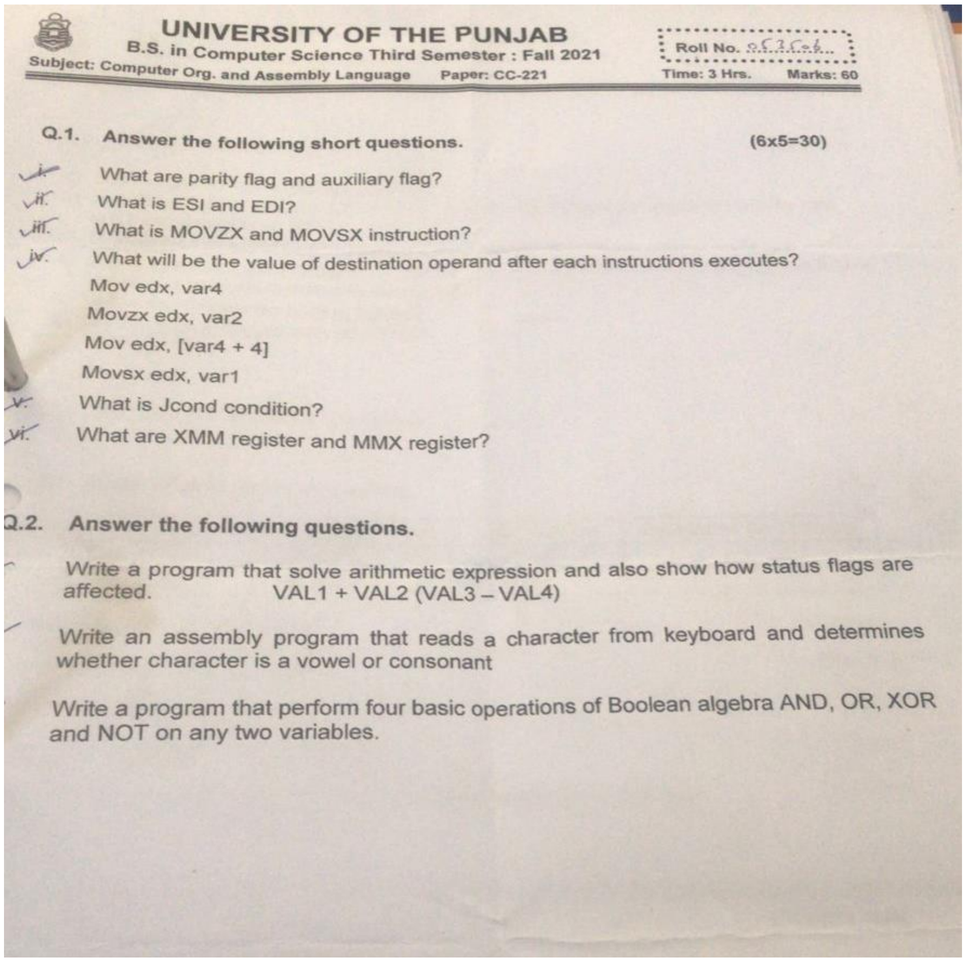

UNIVERSITY OF THE PUNJAB

B.S. in Computer Science Third Semester : Fall 2021

Subject: Computer Org. and Assembly Language **Paper: CC-221** **Roll No.** [illegible] **Time: 3 Hrs.** **Marks: 60**

Q.1. Answer the following short questions. **(6x5=30)**

i. What are parity flag and auxiliary flag?
ii. What is ESI and EDI?
iii. What is MOVZX and MOVSX instruction?
iv. What will be the value of destination operand after each instructions executes?
Mov edx, var4
Movzx edx, var2
Mov edx, [var4 + 4]
Movsx edx, var1
v. What is Jcond condition?
vi. What are XMM register and MMX register?

Q.2. Answer the following questions.

Write a program that solve arithmetic expression and also show how status flags are affected. VAL1 + VAL2 (VAL3 – VAL4)

Write an assembly program that reads a character from keyboard and determines whether character is a vowel or consonant

Write a program that perform four basic operations of Boolean algebra AND, OR, XOR and NOT on any two variables.

Solution 3rd Semester

Short Questions

Q1 (i) What are parity and auxiliary flags?

The parity flag (PF) and auxiliary carry flag (AF) are two flags in the flags register (also known as the status register) of a processor that are used in assembly language programming.

The parity flag (PF):
The parity flag is set to 1 if the least significant byte (LSB) of the result of an arithmetic or logical operation contains an even number of 1 bits; otherwise, it is set to 0. The PF is used for parity checks in error detection algorithms.

For example, if the result of an operation is 10100110 in binary, which contains four 1 bits, then the PF is set to 0 because four is an even number. If the result were 11000111, which contains five 1 bits, then the PF would be set to 1.

The auxiliary carry flag (AF):
The auxiliary carry flag is used in binary-coded decimal (BCD) arithmetic operations. It indicates whether there was a carry from the least significant 4 bits (the lower nibble) to the most significant 4 bits (the upper nibble) during an addition or subtraction operation.

For example, if you add two BCD numbers 0x12 and 0x34, the result is 0x46. Since there is no carry from the lower nibble to the upper nibble, the AF is set to 0. However, if you add 0x29 and 0x34, the result is 0x5D, and there is a carry from the lower nibble to the upper nibble, so the AF is set to 1.

In summary, the parity flag is used for parity checks in error detection algorithms, while the auxiliary carry flag is used in BCD arithmetic operations to indicate a carry between the lower and upper nibbles.

Q 1 (ii). What is ESI and EDI ?

ESI and EDI are commonly used registers in assembly language programming for manipulating data in memory, particularly for string operations and memory copies.

ESI (Extended Source Index) register:

ESI is a 32-bit register used as a pointer to the source data in memory during operations such as data movement, string operations, and memory copies. It is commonly used in conjunction with the ECX (Extended Counter) register for loop control in string operations.

EDI (Extended Destination Index) register:

EDI is a 32-bit register used as a pointer to the destination data in memory during operations such as data movement, string operations, and memory copies. It is commonly used in conjunction with the ECX (Extended Counter) register for loop control in string operations

Q 1 (iii) What is MOVZX and MOVSX instructions?

Answer: MOVZX and MOVSX are instructions used to move values between registers or memory locations of different sizes, while also extending or truncating the values. MOVZX zero-extends the value to fill the upper bits with zeros, while MOVSX sign-extends the value to preserve the sign bit.

MOVZX (Move with Zero-Extend) instruction:

MOVZX is an instruction that copies a value from a source operand to a destination operand while zero-extending the value to a larger size. It is typically used when moving values between registers or memory locations of different sizes.

For example, the instruction "movzx ebx, byte ptr [eax]" will move a byte from the memory location pointed to by EAX into the lower 8 bits of EBX, while zero-extending the value to fill the upper 24 bits of EBX with zeros.

MOVSX (Move with Sign-Extend) instruction:

MOVSX is an instruction that copies a value from a source operand to a destination operand while sign-extending the value to a larger size. It is typically used when moving values between registers or memory locations of different sizes, and when preserving the sign of a value.

For example, the instruction "movsx ebx, byte ptr [eax]" will move a byte from the memory location pointed to by EAX into the lower 8 bits of EBX, while sign-extending the value to fill the upper 24 bits of EBX with the sign bit of the original value.

Q 1 (iv) what will be value in destination operand after each instruction executes
MOV edx, var4
MOVZX edx, var2
MOV edx, var2
MOVSX edx, var1

Answer:
MOV edx, var4:
This instruction moves the value of var4 into the EDX register. The size of var4 is not specified, but assuming it is a 32-bit variable, the value of EDX will be the value of var4.

MOVZX edx, var2:
This instruction moves the value of var2 into the EDX register, while zero-extending the value to a 32-bit size. Assuming var2 is an 8-bit variable with a value of 0x0A, the value of EDX will be 0x0000000A.

MOV edx, var2:
This instruction moves the value of var2 into the EDX register, truncating the value to a 32-bit size. Assuming var2 is a 16-bit variable with a value of 0xABCD, the value of EDX will be 0x0000ABCD.

MOVSX edx, var1:
This instruction moves the value of var1 into the EDX register, while sign-extending the value to a 32-bit size. Assuming var1 is an 8-bit variable with a value of 0xFF (which represents -1 in two's complement notation), the value of EDX will be 0xFFFFFFFF.

Q 1 (v) What is JCond condition
J(cond) stands for Jump and "cond" replace different letter depending upon the jump condition following are all variants of jump conditions
JE: Jump if equal (jump if the two operands being compared are equal)
JNE: Jump if not equal (jump if the two operands being compared are not equal)
JG/JNLE: Jump if greater (jump if the first operand is greater than the second operand)
JGE/JNL: Jump if greater than or equal (jump if the first operand is greater than or equal to the second operand)
JL/JNGE: Jump if less (jump if the first operand is less than the second operand)
JLE/JNG: Jump if less than or equal (jump if the first operand is less than or equal to the second operand)
JC: Jump if carry (jump if the last arithmetic operation resulted in a carry out of the most significant bit)
JNC: Jump if not carry (jump if the last arithmetic operation did not result in a carry out of the most significant bit)
JO: Jump if overflow (jump if the last arithmetic operation resulted in an overflow)
JNO: Jump if not overflow (jump if the last arithmetic operation did not result in an overflow)
JS: Jump if sign (jump if the sign bit of the result is set)
JNS: Jump if not sign (jump if the sign bit of the result is not set)
JP/JPE: Jump if parity (jump if the result has even parity)
JNP/JPO: Jump if not parity (jump if the result has odd parity)

Q 1 (vi) what are XMM and MMX registers?
XMM and MMX registers are both types of registers used in x86-based processors for performing SIMD (Single Instruction Multiple Data) operations. SIMD allows a processor to perform the same operation on multiple pieces of data at the same time, which can greatly accelerate certain types of computations.

MMX (Multimedia Extensions) registers were introduced by Intel in 1996 and were originally intended for accelerating multimedia and graphics processing tasks. MMX registers are 64 bits wide and are used for performing integer-based SIMD operations.

XMM registers, on the other hand, were introduced with the introduction of the SSE (Streaming SIMD Extensions) instruction set in 1999. XMM registers are 128 bits wide and are used for performing SIMD operations on both integer and floating-point data types. SSE also introduced additional instructions for performing more complex operations on SIMD data, such as vector addition, multiplication, and shuffle operations.

Both MMX and XMM registers are used by compilers and applications to optimize performance in a variety of areas, including multimedia, scientific computing, and gaming.

Long Questions

Q 2 (i) write a program that solves arithmetic expressions and also show how status flags are affected? Val1+val2(val3-val4)

```
section .data
  Val1 dd 10
  Val2 dd 5
  Val3 dd 7
  Val4 dd 3
  Result dd 0

section .text
  global main
  extern printf, scanf

main:
  ; Calculate Val3 - Val4 and store the result in EBX
  mov eax, [Val3]
  sub eax, [Val4]
  mov ebx, eax

  ; Multiply Val2 by EBX and store the result in EAX
```

```
    mov eax, [Val2]
    imul ebx

    ; Add Val1 to EAX and store the result in Result
    add eax, [Val1]
    mov [Result], eax

    ; Display the result
    push dword [Result]
    push format
    call printf
    add esp, 8

    ; Show how the status flags are affected
    cmp eax, 0
    jz zero
    ja positive
    js negative

zero:
    mov ebx, 0
    mov ecx, 0
    jmp end

positive:
    mov ebx, 1
    mov ecx, 0
    jmp end

negative:
    mov ebx, 0
    mov ecx, 1

end:
    ; Exit the program
    xor eax, eax
    ret

section .data
format db "%d", 10, 0
```

The program first defines the variables Val1, Val2, Val3, Val4, and Result in the .data section of the program. It then moves Val3 and Val4 into the EAX and EBX registers, respectively, subtracts EBX from EAX, and stores the result back in EBX. This calculates the value of (Val3 - Val4).

Next, the program multiplies Val2 by EBX using the imul instruction, adds Val1 to the result using the add instruction, and stores the final result in the Result variable.

The program then displays the result using the printf function, and uses the cmp instruction to compare the result with zero. Depending on the result of the comparison, the program sets the EBX and ECX registers to different values to indicate whether the result is positive, negative, or zero.

Finally, the program exits using the ret instruction.

Note that the status flags are affected by the cmp, sub, add, and imul instructions used in the program. These instructions modify the CF (carry flag), ZF (zero flag), SF (sign flag), OF (overflow flag), and AF (auxiliary carry flag) flags, which can be used by other instructions to make decisions based on the outcome of these operations.

Q 1 (ii) write an assembly program that reads a character from keyboard and determines whether character is a vowel or consonents?

```
section .data
   prompt db "Enter a character: ", 0
   vowel db "Vowel", 0
   consonant db "Consonant", 0

section .bss
   char resb 1

section .text
   global main
   extern printf, scanf

main:
   ; Display prompt and read character from keyboard
   push prompt
   call printf
   add esp, 4
   push char
   push dword 1
   push dword 0
   call scanf
   add esp, 12

   ; Check if character is a vowel or consonant
   mov al, [char]
   cmp al, 'a'
   jb consonant
   cmp al, 'z'
   ja consonant
   cmp al, 'e'
   je vowel
   cmp al, 'i'
   je vowel
   cmp al, 'o'
```

```
    je vowel
    cmp al, 'u'
    je vowel
    jmp consonant

vowel:
    ; Display "Vowel"
    push vowel
    call printf
    add esp, 4
    jmp end

consonant:
    ; Display "Consonant"
    push consonant
    call printf
    add esp, 4

end:
    ; Exit the program
    xor eax, eax
    ret
```

The program first defines the variables prompt, vowel, and consonant in the .data section of the program, and the variable char in the .bss section of the program. The prompt variable contains a message asking the user to enter a character, while the vowel and consonant variables contain the strings "Vowel" and "Consonant", respectively.

The program then displays the prompt message using the printf function, reads a single character from the keyboard using the scanf function, and stores the result in the char variable.

Next, the program checks if the character is a vowel or a consonant by comparing it to the letters 'a', 'e', 'i', 'o', and 'u' using the cmp instruction. If the character matches one of these vowels, the program jumps to the vowel label and displays the "Vowel" message. Otherwise, the program jumps to the consonant label and displays the "Consonant" message.

Finally, the program exits using the ret instruction.

Q 2 (iii) write a program that performs four basic operations of Boolean algebra AND, OR, XOR and NOT on any two variables

```
section .data
    var1 dd 10110110b
    var2 dd 01101001b

section .text
    global main

main:
    ; Perform bitwise AND on var1 and var2
    mov eax, [var1]
    and eax, [var2]
    ; Result is now in eax

    ; Perform bitwise OR on var1 and var2
    mov ebx, [var1]
    or ebx, [var2]
    ; Result is now in ebx

    ; Perform bitwise XOR on var1 and var2
    mov ecx, [var1]
    xor ecx, [var2]
    ; Result is now in ecx

    ; Perform bitwise NOT on var1
    mov edx, [var1]
    not edx
    ; Result is now in edx

    ; Exit the program
    xor eax, eax
    ret
```

The program first defines the variables var1 and var2 in the .data section.

In the main function, the program performs the AND, OR, and XOR operations on var1 and var2 using the and, or, and xor instructions, respectively. The results are stored in the eax, ebx, and ecx registers, respectively.

The program then performs the NOT operation on var1 using the not instruction, and stores the result in the edx register.

Finally, the program exits using the ret instruction.