PAMANTASAN NG LUNGSOD NG MAYNILA
(University of the City of Manila)
Intramuros, Manila

MICROPROCESSOR (LECTURE)

Activity No. 2
Assembly Language

Score

Submitted by:
Laurente, Queenie D.
S 1:00-7:00 / CPE 0412-2

Date Submitted
21-10-2023

Submitted to:

Engr. Maria Rizette H. Sayo

Note: Answers are in green font and explanations are provided below the code.

Screenshot 1: Explain why each of the following MOV statements are invalid:

```
.data
bVal   BYTE   100
bVal2  BYTE    ?
wVal   WORD    2
dVal   DWORD   5
.code
   mov ds, 45         ; a. it is not allowed to set DS with immediate value
   mov esi, wVal      ; b. sizes of source register and memory don't match
   mov eip, dVal      ; c. EIP value cannot be controlled
   mov 25, bVal       ; d. immediate value cannot be a destination
   mov bVal2, bVal    ; e. moving data directly from one memory location to
                           another is not allowed
```

Explanation:

In assembly language, the MOV statement stands for move. It is primarily used to transfer data from one location to another within the computer's memory or registers. Its general syntax goes by: MOV <destination>, <value> ; Each examples above are invalid because of the following reasons:

a. In this instruction, the destination is DS or the data segment. DS is a segment register that stores the base address of the data segment to determine the location of data in memory. The value in DS is set by a program loader so it cannot be directly manipulated by the user by providing an immediate value and using the MOV instruction since it is not a normal case.
b. In this instruction, the wVal variable is 16-bit, while the ESI destination is 32-bit. Since there is a size mismatch, the value cannot be transferred in the desired destination. However, sign-extend or zero-extend instruction can be used to handle size mismatch.
c. In this instruction, the assigned destination is EIP or the extended instruction pointer. EIP is a special register that points to the memory address of the next instruction to be executed. It is not possible to set its value manually by the user.
d. In this instruction, the destination is 25 which is an immediate value. Immediate values cannot be destinations because the MOV instruction only moves data between registers and memory location only. If the instruction is reversed, 25 may be stored in bVal instead.
e. In this instruction, the destination is bVal2 which is a memory location, whereas, the value is from bVal, which is also a memory location. As mentioned, the MOV instruction can only transfer data from memory to register or vice versa. It cannot transmit data from one memory location to another memory location. To be able to do that, a temporary register is necessary.

Screenshot 2: Show the value of the destination operand after each of the following instruction executes:

```
.data
myByte  BYTE  0FFh,  0
.code
   mov al, myByte       ; AL = FFh
   mov ah, [myByte+1]   ; AH = 00h
   dec ah               ; AH = FFh
   inc al               ; AL = 00h
   dec ax               ; AX = FEFF
```

Explanation:

In the .data, the variable myByte stores a value of 0FFH and 0 accordingly. The first instruction moves the value of myByte (0FFH) in the AL register, therefore, AL = 0FFH. Then, the value of myByte + 1 or the second element of myByte which is 0 is stored in AH, hence, AH = 0. Afterwards, the value in AH was decremented (dec) from 0 to 0FFh. Contrastingly, the value in AL was incremented (inc) from 0FFh to 00h. Lastly, the AX register which is a combination of AL and AH registers, decrements from 0FF00h to 0FEFFh.

Screenshot 3: For each of the following market entries, show the value of the destination operand and the sign, zero, and carry flags:

```
mov ax, 00FFh
add ax, 1      ; AX = 0100h   SF = 0 ZF = 0 CF = 0
sub ax, 1      ; AX = 00FFh   SF = 0 ZF = 0 CF = 0
add al, 1      ; AL = 00h     SF = 0 ZF = 1 CF = 1
mov bh, 6Ch
```

```
add bh, 95h     ; BH = 01h      SF = 0 ZF = 0 CF = 1

mov al, 2
sub al, 3       ; AL = FFh      SF = 1 ZF = 0 CF = 1
```

Explanation:

The first line in this code simply loads the register AX with an immediate value of 00FFh. In this instruction, the AX = 00FFh since it is the loaded value, the sign flag or SF = 0 because the most significant bit of AX is 0, the zero flag ZF = 0 since the value of AX is not 0, and the carry flag CF = 0 because there is no carry from the previous operation. The second instruction simply adds 1 to the AX register, hence, AX = 00FFh + 1 = 0100h. Its SF, ZF, and CF still remains 0 because of the same reason with the first instruction. Subsequently, 1 is subtracted from the AX register, hence AX goes back to its original value, AX = 00FFh. All flags SF, ZF, and CF still remains 0 because of the same reason with the first instruction. Afterwards, 1 is added to the AL register or the lower byte of the AX register. AL = 00h, SF = 0 since the most significant bit of AL value is 0, ZF = 1 because AL is now 0, and CF = 1 since there is a carry from the addition operation. The fifth line of instruction moves the immediate value 6Ch in the BH register. Then, an immediate value 95h was added in the BH register making BH = 01h, SF = 0, ZF = 0, and CF = 1 since there is a carry upon adding the mentioned values. Lastly, an immediate value of 2 is moved in the AL register, so AL = 2. Next, 3 is subtracted from the same register, giving the final value of AL = FFh, SF = 1 since the most significant bit is 1, ZF = 0 since the value is not 0, and CF = 1 since there is a carry upon subtracting the two values.

Screenshot 4: What will be the values of the Overflow flag?

```
mov al, 80h
add al, 92h     ; OF = 1

mov al, -2
add al, +127    ; OF = 0
```

Explanation:

The overflow flag (OF) is used to indicate when an arithmetic operation results in an overflow or underflow, specifically in signed integer arithmetic. An overflow occurs when the result is too large to be represented within the specified number of bits. In the code provided, the first line of code simply moves an intermediate value of 80h in the AL register. The second instruction adds 92h in the AL register. Since there is an overflow when 80h is added with 92h because the result exceeds the maximum value that can be represented by an 8-bit signed integer, OF becomes 1 or OF = 1. Similarly, the immediate value of -2 was moved in AL register. Then, the immediate value, 127, is added in the AL register. The result becomes 125, and there is no overflow since it is within the valid range of signed 8-bit integers. Therefore, OF = 0.

Screenshot 5: What will be the values of the carry and overflow flags after each operation?

```
mov al, -128
neg al          ; CF = 1 OF = 1

mov ax, 8000h
add ax, 2       ; CF = 0 OF = 0

mov ax, 0
sub ax, 2       ; CF = 1 OF = 0

mov al, -5
sub al, +125    ; CF = 1
```

Explanation:

First, an immediate value of -128 is moved in the AL register, so AL = -128. This value was then negated (neg) in the second instruction, hence AL = -(-128) = 128. The carry flag CF = 1 because there is a borrow in negating -128, while the overflow flag OF = 1 because the negation resulted in an overflow of values. Next, an immediate value of 8000h was moved in the AX register, so AX = 8000h. 2 is then added in the AX register per instruction. The CF = 0 because the addition process did not need carry and OF = 0 since the result is within the valid range of a signed 16-bit integer. Then, 0 is moved in the AX register, so AX = 0. A value of 2 is subtracted from the AX register, making the CF = 1 since it needed to borrow value, while OF = 0 because the result is still within the valid range of a signed 16-bit integer. Lastly, an immediate value of -5 is moved in the AL register, so AL = -5. This was followed by a subtraction of 125 in the same

register, which makes CF = 1 since the subtraction operation needed to borrow and OF = 0 for the same reason as above.

Screenshot 6: (1) What will be the final value of ax? **10**
(2) How many times will the loop execute? **4294967296**

```
   mov ax, 6
   mov ecx,4
L1:
   inc ax
   loop L1
```

```
   mov ecx, 0
X2:
   inc ax
   loop X2
```

Explanation:

For the first block of code, its first line of instruction moves the immediate value of 6 in the AX register. The second line of code almost bears the same function. It also moves an immediate value of 4 in the ECX register. Therefore, AX = 6 and EXC = 4. The loop will only stop once the value in the ECX register becomes 0. The first iteration begins with incrementing AX, so AX = 6 + 1 = 7 and decrementing ECX, so ECX = 4 – 1 = 3. The loop continues: AX = 8 while ECX = 2, AX = 9 while ECX = 1, and finally, AX = 10 while ECX = 0. The loop will now end since the value in the ECX register is 0. Therefore, the final value of AX is 10. For the second block of code, its first line of instruction moves an immediate value of 0 in the ECX register, hence ECX = 0. The loop x2 will only stop once the value in the ECX register becomes 0. The first iteration begins with incrementing the AX register while decrementing the ECX, so AX = 1 while ECX = -1. In assembly language, when a loop counter is set to -1, it is treated as the unsigned integer 4,294,967,295, which is equivalent to 2^32 - 1 or the two's complement representation of -1. As a result, the loop will run for a total of 4,294,967,296 times.