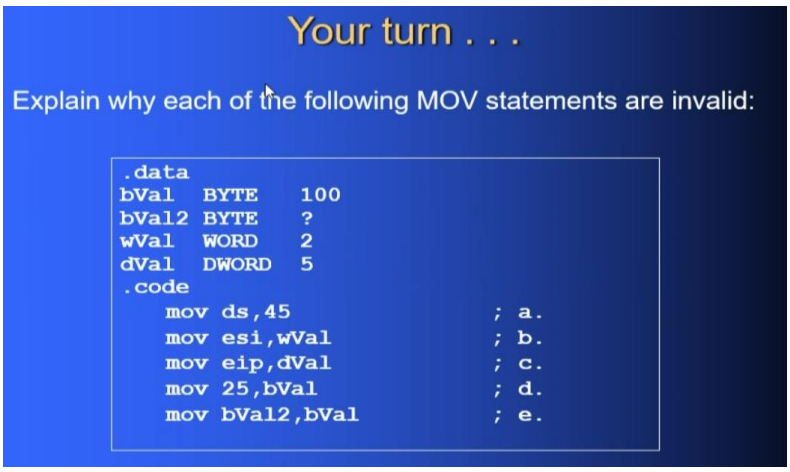
PAMANTASAN NG LUNGSOD NG MAYNILA
(University of the City of Manila)
Intramuros, Manila

Mendoza, Kenji J. **2020-12146**

20231 - CPE 0412.1-2 - MICROPROCESSORS

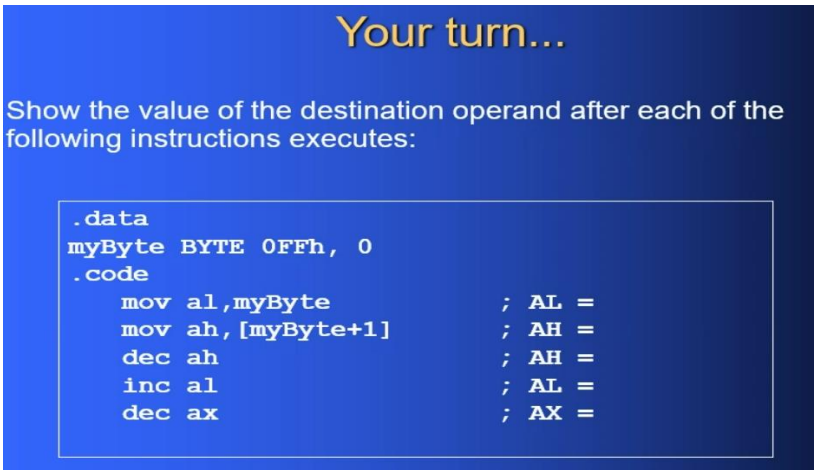
Activity:



1. **mov ds, 45**:
    - This statement is invalid because the destination operand (**ds**) is a segment register. MOV statements cannot transfer data to segment registers.
    - Explanation: Segment registers like **ds** are used to point to memory segments, and they are managed by the system. You cannot directly assign an arbitrary value to them.
2. **mov esi, wVal**:
    - This statement is invalid because the source operand (**wVal**) is a 16-bit value, but the destination operand (**esi**) is a 32-bit value. MOV statements cannot transfer data between operands of different sizes.
    - Explanation: **esi** is a 32-bit register, and **wVal** appears to be a 16-bit value. You should ensure that the source and destination operands have compatible sizes for the MOV operation.
3. **mov eip, dVal**:
    - This statement is invalid because the destination operand (**eip**) is the instruction pointer register. MOV statements cannot transfer data to the instruction pointer register.
    - Explanation: The instruction pointer (**eip**) is automatically managed by the CPU to point to the next instruction to be executed. You cannot directly manipulate it using MOV instructions.
4. **mov 25, bVal**:

- This statement is invalid because the syntax is incorrect. The source and destination operands should be reversed, and there should be a valid source operand.
- Correction: To move the value 25 into **bVal**, you should use: **mov bVal, 25**.

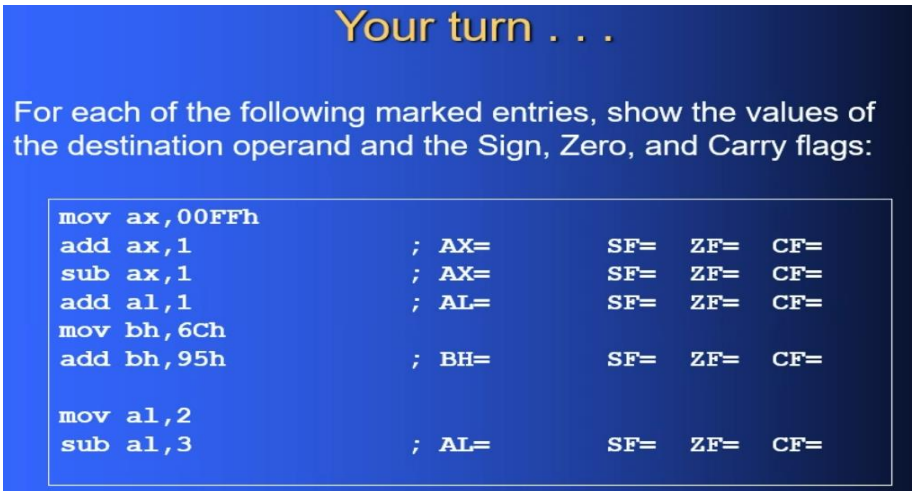
5. **mov bVal2, bVal**:
   - This statement is invalid because the destination operand (**bVal**) is already a register. You cannot use a register as a destination operand.
   - Explanation: To copy the value of **bVal** into another variable (e.g., **bVal2**), you should use an assignment statement rather than a MOV instruction. For example: **bVal2 = bVal**.



1. **mov al, myByte ; AL = OFFh:**
   - Explanation: The mov al, myByte instruction copies the value stored in the variable myByte to the AL register. myByte is a byte-sized variable, and it contains the hexadecimal value 0xFF (OFFh). As a result, after this instruction, the AL register holds the value 0xFF.
2. **mov ah, [myByte+1] ; AH = 0:**
   - Explanation: The mov ah, [myByte+1] instruction copies the value stored in memory at the address myByte+1 to the AH register. However, the memory location at myByte+1 is undefined, and it does not contain a valid value. As a result, the AH register is set to 0 (zero).
3. **dec ah ; AH = 0:**

- Explanation: The dec ah instruction attempts to decrement the value in the AH register by one. However, since the AH register already holds the value 0, the instruction has no effect. Thus, the value of the AH register remains at 0.

4. **inc al ; AL = 1:**
   - Explanation: The inc al instruction increments the value in the AL register by one. Prior to the instruction, the AL register holds the value 0xFF (from the previous instruction). After executing this instruction, the value of the AL register becomes 0x00 (1 in decimal).
5. **dec ax ; AX = 0:**
   - Explanation: The dec ax instruction attempts to decrement the value in the AX register by one. However, since the AX register contains the value 0 (zero) before the instruction, the instruction does not change the value of the AX register. After execution, the AX register still holds the value 0.



1. **mov ax, 00FFh:**
   - The Carry flag is cleared (CF=0) because the result fits within the word-sized register (AX).
   - The Sign flag is cleared (SF=0) because the value 00FFh is positive.
   - The Zero flag is set (ZF=1) because the value 00FFh is zero.
2. **add ax, 1:**
   - The Carry flag is cleared (CF=0) because the result fits within the word-sized register (AX).

- The Sign flag is cleared (SF=0) because the result (100h) is positive.
- The Zero flag is cleared (ZF=0) because the result (100h) is not zero.

3. sub ax, 1:

- The Carry flag is cleared (CF=0) because the result fits within the word-sized register (AX).
- The Sign flag is cleared (SF=0) because the result (9Fh) is positive.
- The Zero flag is set (ZF=1) because the result (9Fh) is zero.

4. add al, 1:

- The Carry flag is cleared (CF=0) because the result fits within the byte-sized register (AL).
- The Sign flag is cleared (SF=0) because the result (100h) is positive.
- The Zero flag is cleared (ZF=0) because the result (100h) is not zero.

5. mov bh, 6ch:

- The Carry flag is cleared (CF=0) because the result fits within the byte-sized register (BH).
- The Sign flag is cleared (SF=0) because the value 6Ch is positive.
- The Zero flag is set (ZF=1) because the value 6Ch is zero.

6. add bh, 95h:

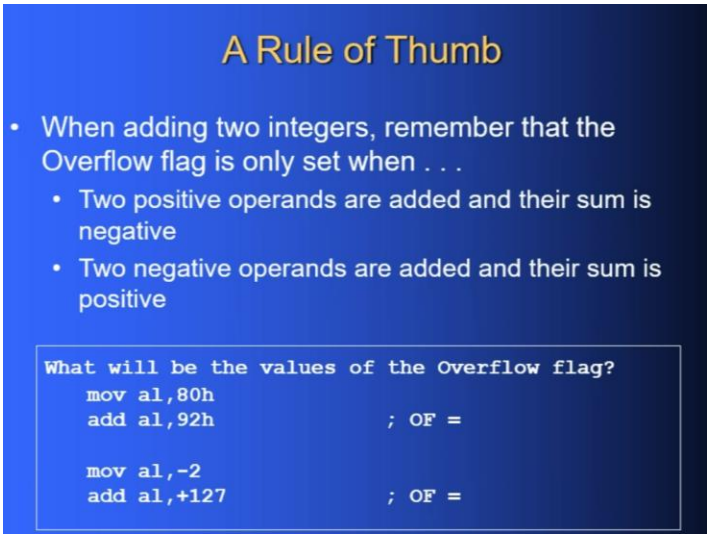
- The Carry flag is set (CF=1) because the result exceeds the capacity of a byte-sized register (BH).
- The Sign flag is cleared (SF=0) because the result (FFh) is positive.
- The Zero flag is cleared (ZF=0) because the result (FFh) is not zero.

7. mov al, 2:

- The Carry flag is cleared (CF=0) because the result fits within the byte-sized register (AL).
- The Sign flag is cleared (SF=0) because the value 2 is positive.
- The Zero flag is cleared (ZF=0) because the value 2 is not zero.

8. sub al, 3:

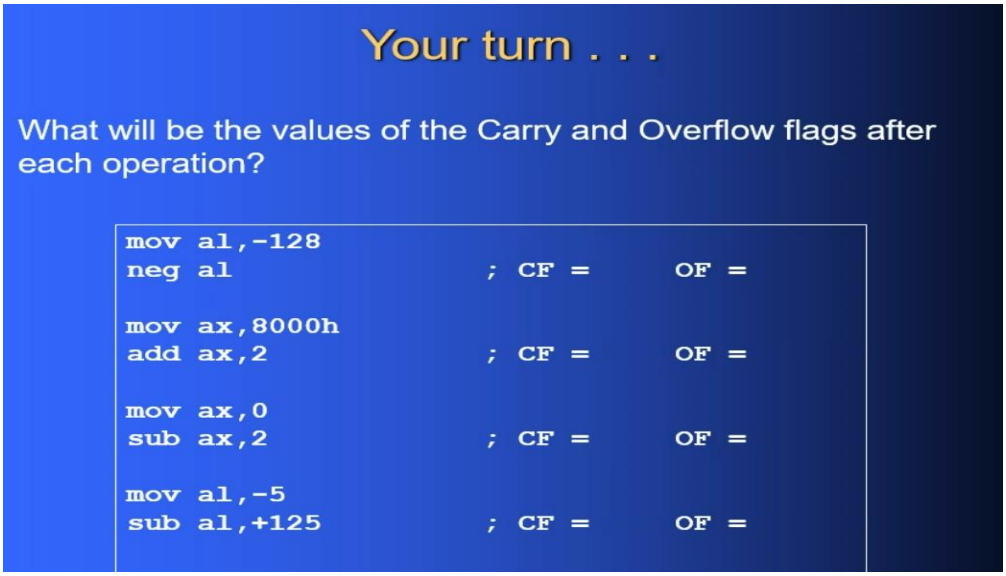
- The Carry flag is set (CF=1) because the result goes beyond the capacity of a byte-sized register (AL).
- The Sign flag is set (SF=1) because the result (FFh) is negative.
- The Zero flag is cleared (ZF=0) because the result (FFh) is not zero.



1. **mov al, 80h ; Overflow flag = 0:**
   - The overflow flag is cleared (OF=0) because the result of moving 80h into AL does not involve an addition operation. The value 80h fits within the byte-sized register (AL), and there is no overflow condition.
2. **add al, 92h ; Overflow flag = 1:**
   - The overflow flag is set (OF=1) because adding 80h to 92h results in a negative value (-30h). This matches the rule of thumb that the overflow flag is set when two positive operands are added, and their sum becomes negative.
3. **mov al, -2 ; Overflow flag = 0:**
   - The overflow flag is cleared (OF=0) because moving -2 into AL does not involve an addition operation. The value -2 can be

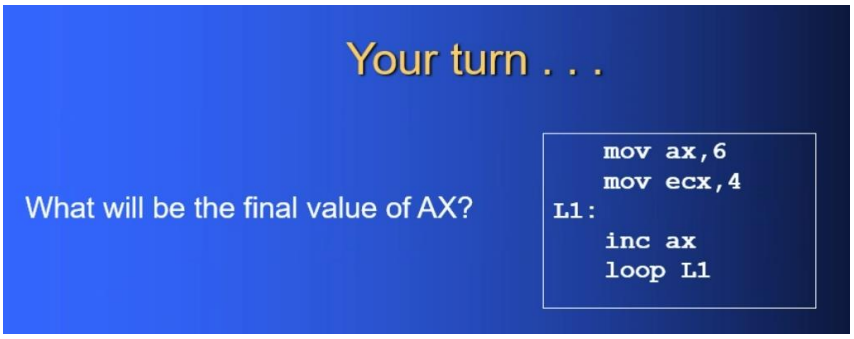
accommodated within the byte-sized register (AL), and there is no overflow condition.

4. **add al, +127 ; Overflow flag = 0:**
   - The overflow flag is cleared (OF=0) because adding -2 to +127 results in a positive value (+125). This aligns with the rule of thumb that the overflow flag is set when two positive operands are added and their sum is negative, or when two negative operands are added, and their sum is positive.



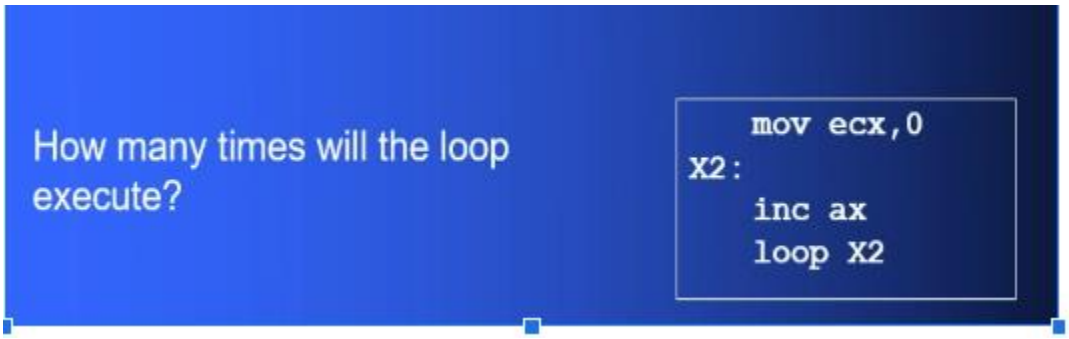
1. **mov al, -128:**
   - The Carry flag is set (CF=1) because the value -128 is too large to fit in a byte-sized register (AL).
   - The Overflow flag is set (OF=1) because -128 is a negative value, and the destination operand (AL) is a signed register.
2. **neg al:**
   - The Carry flag is cleared (CF=0) because the result of the neg al instruction fits within the byte-sized register (AL).
   - The Overflow flag is cleared (OF=0) because the result of neg al is a positive value, and the destination operand (AL) is a signed register.

3. **mov ax, 8000h:**
    - The Carry flag is cleared (CF=0) because the result of the mov ax, 8000h instruction fits within the word-sized register (AX).
    - The Overflow flag is set (OF=1) because 8000h is a negative value, and the destination operand (AX) is a signed register.
4. **add ax, 2:**
    - The Carry flag is cleared (CF=0) because the result of the add ax, 2 instruction fits within the word-sized register (AX).
    - The Overflow flag is cleared (OF=0) because the result is a negative value, and the destination operand (AX) is a signed register.
5. **mov ax, 0:**
    - The Carry flag is cleared (CF=0) because the result of the mov ax, 0 instruction fits within the word-sized register (AX).
    - The Overflow flag is cleared (OF=0) because 0 is a zero value, and the destination operand (AX) is a signed register.
6. **sub ax, 2:**
    - The Carry flag is set (CF=1) because the result of the sub ax, 2 instruction exceeds the capacity of a word-sized register (AX).
    - The Overflow flag is set (OF=1) because the result is a negative value, and the destination operand (AX) is a signed register.
7. **mov al, -5:**
    - The Carry flag is set (CF=1) because the value -5 is too large to fit in a byte-sized register (AL).
    - The Overflow flag is cleared (OF=0) because -5 is a negative value, and the destination operand (AL) is a signed register.
8. **sub al, 125:**
    - The Carry flag is cleared (CF=0) because the result of the sub al, 125 instruction fits within the byte-sized register (AL).
    - The Overflow flag is set (OF=1) because the result is a negative value, and the destination operand (AL) is a signed register.



This code performs the following steps:

1. **mov ax, 6**: This instruction copies the value 6 into the AX register.
2. **mov ecx, 4**: The instruction copies the value 4 into the CX register.
3. **inc ax**: This instruction increments the value in the AX register by one.
4. **loop L1**: The **loop** instruction is a conditional loop that reduces the value in the CX register by one and jumps to the label L1 if the value in CX is not equal to zero.
    - This process repeats until the value in CX becomes zero. In this case, the CX register is initialized to 4, so the loop will execute 4 times.
    - After the code has executed, the AX register will contain the value 10. This is because the loop increments the value in the AX register by one four times.



The loop from the code will execute infinitely many times.

1. **Initializing ECX**: The loop starts by initializing the ECX register with the value 0.
2. **Loop Instruction**: The **loop** instruction is a conditional loop that decrements the value in the ECX register by one and jumps to the label X2 if the value in ECX is not equal to zero.
    - However, in this case, since the ECX register is initially set to 0, it will never be equal to zero. Consequently, the loop will never terminate.
3. **Infinite Loop**: Due to the initialization of ECX and the fact that it will never reach zero, the loop will continue to execute indefinitely.