Overlapping Values:

In previous lab, you have seen that MOV procedure require both operands to be of same size. That's mean, we can't copy a BYTE into AX or EAX but only in AL. When **oneByte** is moved to AL, it overwrites the existing value of AL but all values of other bytes remain same. When **oneDword** is moved to EAX, it overwrites AX. If 0 is moved to AX, all values of lower 16 bits of EAX will become 0. Try debugging and viewing register values with following program.

*"**DumpRegs** print the current value of all registers and Flags.*
Values printed by DumpRegs are in Hexdecimal, from now and on, we will always use hexadecimal values prefixed with 'h' to avoid confusion."

Example - OVERLAPPING

```
INCLUDE Irvine32.inc
.data

oneByte BYTE 78h
oneWord WORD 1234h
oneDword DWORD 12345678h


.code
main PROC

      mov eax,99999999h          ; EAX = 99999999h
      call DumpRegs
      mov al,oneByte             ; EAX = 00000078h
      call DumpRegs
      mov ax,oneWord             ; EAX = 00001234h
      call DumpRegs
      mov eax,oneDword           ; EAX = 12345678h
      call DumpRegs
      mov ax,0                   ; EAX = 12340000h
      call DumpRegs
      exit

main ENDP
END main
```

Copying Smaller Values to Larger Ones

Although MOV cannot directly copy data from a smaller operand to a larger one, programmers can create workarounds. Suppose **count** (unsigned, 16 bits) must be moved to ECX (32 bits). We can set ECX to zero and move **count** to CX:

```
.data
count WORD 1
.code
mov ecx,0
mov cx,count
```

What happens if we try the same approach with a signed integer equal to 16?

```
.data
signedVal SWORD -16 ; FFF0h (-16)
.code
mov ecx,0
mov cx,signedVal ; ECX = 0000FFF0h (+65,520)
```

The value in ECX (65,520) is completely different from 16. On the other hand, if we had filled ECX first with FFFFFFFFh and then copied **signedVal** to CX, the final value would have been correct:

```
mov ecx,0FFFFFFFF
mov cx,signedVal ; ECX = FFFFFFF0h (-16)
```

The effective result of this example was to use the highest bit of the source operand (1) to fill the upper 16 bits of the destination operand, ECX. This technique is called *sign extension*. Of course, we cannot always assume that the highest bit of the source is a 1. Fortunately, the engineers at Intel anticipated this problem when designing the Intel386 processor and introduced the MOVZX and MOVSX instructions to deal with both unsigned and signed integers.

MOVZX Instruction

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

Format

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

*reg/mem8    =  8-bit operand, which can be an 8-bit general register or memory byte
```

```
*reg/mem16   =  16-bit operand, which can be a 16-bit general register or memory word
```

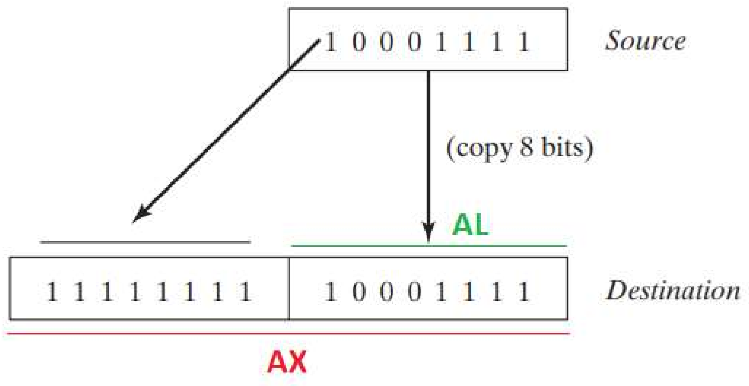
32 bit-operand can’t be copied to any destination using MOVZX. Because it is the job of MOV. In each of the three variants, the first operand (a register) is the destination and the second is the source. In following example, a BYTE (111) is moved in EAX. AL part of EAX will have value of that moved BYTE (111) and rest part of the EAX will automatically become zero.

Example - MOVZX

```
INCLUDE Irvine32.inc
.data
oneDword Dword 12345678h
oneByte Byte 11h
.code
main PROC

       mov eax, oneDword          ; EAX = 123456h
       call DumpRegs
       movzx eax, oneByte         ; EAX = 000011h
       call DumpRegs
       call ReadInt
       exit

main ENDP
END main
```

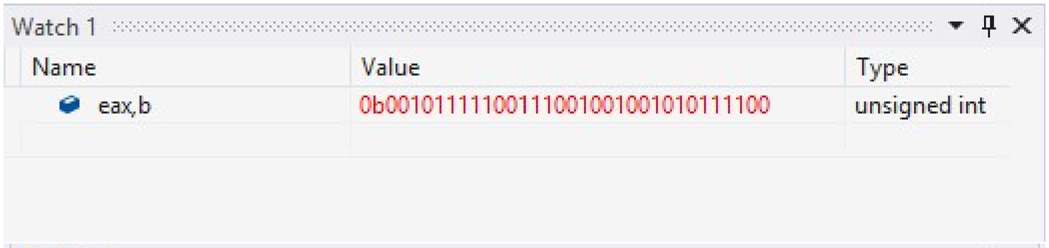
MOVSX Instruction

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 just work like MOVZX, instead of copying 0, it copy (1 for negative 0 for positive) in all other bytes. In singed integers, Left most bit is reserved for sign. SBYTE (Signed-Byte) have left most bit of sign and rest 7 bits for storing data.
So, SBYTE can store -128 to +127. While unsinged (BYTE) can store 0 to 255.
If we copy 10001111 from BYTE to AX (16-bit register). AL will contain 10001111 and the rest part of AX will have 1 in it.

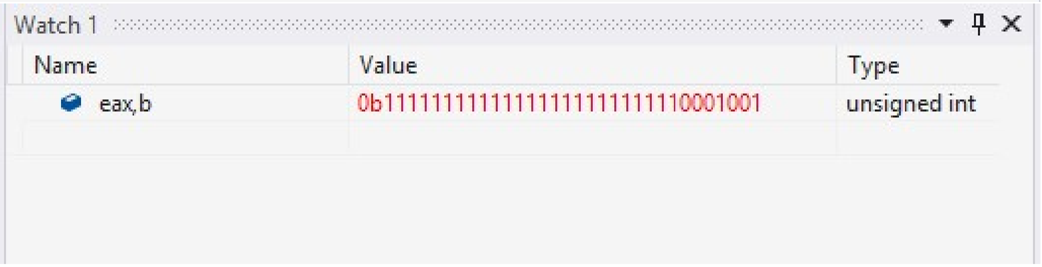


Example - MOVSX

```
INCLUDE Irvine32.inc
.data

oneDword SDword 00101111100111001001001010111100b
oneByte SBYTE 10001001b

.code
main PROC

     mov eax, oneDword          ; EAX = 00101111100111001001001010111100
     movsx eax, oneByte         ; EAX = 11111111111111111111111110001001
     call ReadInt
     exit

main ENDP
END main
```



Name	Value	Type
eax,b	0b00101111100111001001001010111100	unsigned int



Name	Value	Type
eax,b	0b11111111111111111111111110001001	unsigned int

XCHG Instruction

The XCHG (exchange data) instruction exchanges the contents of two operands. There are three variants:

Format

```
XCHG reg,reg
XCHG reg,mem
XCHG mem,reg
```

The rules for operands in the XCHG instruction are the same as those for the MOV instruction except that XCHG does not accept immediate operands. In array sorting applications, XCHG provides a simple way to exchange two array elements. XCHG doesn't work either if both operands are variables.

Example – Operand Exchange

```
INCLUDE Irvine32.inc
.data
a dword 1111h
b dword 2222h
.code
main PROC
      Mov Eax,a
      Mov Ebx,b
      Call DumpRegs
      Xchg Eax,Ebx
      Call DumpRegs
      Call ReadInt
      Exit
main ENDP
END main
```

INC and DEC Instructions

The INC (increment) and DEC (decrement) instructions, respectively, add 1 and subtract 1 from a single operand.

Format:

```
INC reg/mem
DEC reg/mem
```

Following Example shows how a value can be incremented and decremented.

Example – Increment Decrement

```
INCLUDE Irvine32.inc
.data
a dword 1111h
.code
main PROC
       Mov Eax,a
       Call DumpRegs
       Inc Eax
       Call DumpRegs
       Dec Eax
       Call DumpRegs

       Call ReadInt
       Exit
main ENDP
END main
```

Flags

Flags Affected by Addition and Subtraction

When executing arithmetic instructions, we often want to know something about the result. Is it negative, positive, or zero? Is it too large or too small to fit into the destination operand? Answers to such questions can help us detect calculation errors that might otherwise cause erratic program behavior. We use the values of CPU status flags to check the outcome of arithmetic operations.
We also use status flag values to activate conditional branching instructions, the basic tools of program logic. Here's a quick overview of the status flags.

• The **Carry flag** indicates unsigned integer overflow. For example, if an instruction has an 8-bit destination operand but the instruction generates a result larger than 11111111 binary, the
Carry flag is set.

• The **Overflow flag** indicates signed integer overflow. For example, if an instruction has a 16- bit destination operand but it generates a negative result smaller than -32,768 decimal, the
Overflow flag is set.

• The **Zero flag** indicates that an operation produced zero. For example, if an operand is subtracted from another of equal value, the Zero flag is set.

• The **Sign flag** indicates that an operation produced a negative result. If the most significant bit
(MSB) of the destination operand is set, the Sign flag is set.

• The **Parity flag** indicates whether or not an even number of 1 bit occurs in the least significant byte of the destination operand, immediately after an arithmetic or Boolean instruction has executed.

Zero Flag

The Zero flag is set when the result of an arithmetic operation is zero. The following example show the state of Zero Flag when value of EAX become Zero after subtraction.

Example – Zero Flag

```
INCLUDE Irvine32.inc
.data
.code
main PROC
     Mov Eax,0h
     add Eax,10h
     Call DumpRegs
     Sub Eax, 10h
     Call DumpRegs
     Call ReadInt
     Exit
main ENDP
END main
```

Carry Flag

When adding two unsigned integers, the Carry is set to 1 if destination overflow. For example, If 255 is moved in AL and then try to add 1 in it, AL will overflow because it cannot store value greater than 255, CF will set to 1 to represent this overflow.

Example – Carry Flag

```
INCLUDE Irvine32.inc
.data
a BYTE 255
b BYTE 1
.code
main PROC
    Call DumpRegs
       Mov al,a
       add al,b
       Call DumpRegs
       Call ReadInt
       Exit
main ENDP
END main
```

On the other hand, if 1 is added to 255 in AX, the sum easily fits into 16 bits and the Carry flag is clear (set to 0).

Sign Flag

The Sign flag is set when the result of a signed arithmetic operation is negative. In example, we tried to move 5 in EAX and then Subtracted 10 from it, causing a negative answer SF to become 1.

Example – Sign Flag

```
INCLUDE Irvine32.inc
.data
.code
main PROC
    Call DumpRegs
       Mov Eax, 5
       Sub Eax, 10
       Call DumpRegs
       Call ReadInt
       Exit
main ENDP
END main
```

Overflow Flag

The Overflow flag is set when the result of a signed arithmetic operation overflows or underflows the destination operand. Adding 1 in +127 store in AL causes signed overflow.

```
.data
mov al,+127
add al,1                        ;OF = 1
```

Subtracting 1 from -128 causes Signed underflow. -128 is smallest value that can be store in BYTE or AL.

```
.data
mov al,-128
sub al,1                        ;OF = 1
```

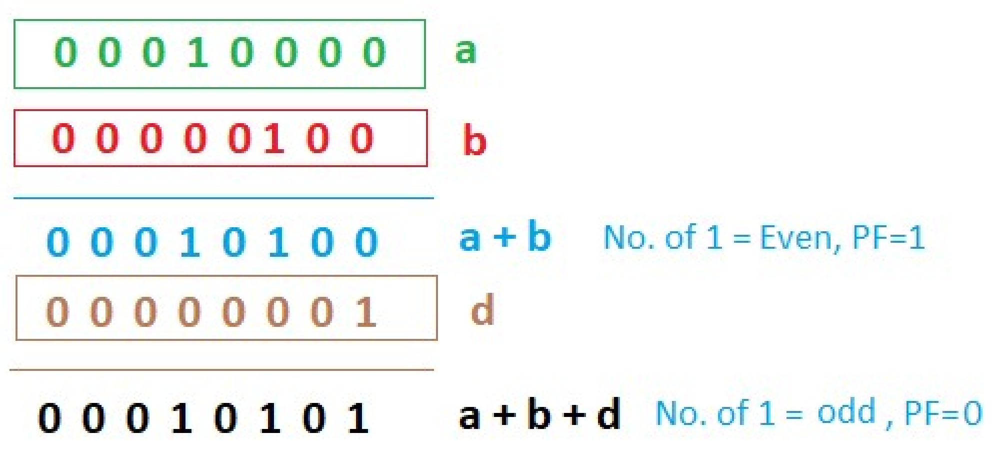
Example – Overflow Flag (Signed Underflow)

```
INCLUDE Irvine32.inc
.data
.code
main PROC
    Call DumpRegs
       Mov AL, -128
       Sub AL, 1
       Call DumpRegs
       Call ReadInt
       Exit
main ENDP
END main
```

Parity Flag

The Parity flag (PF) is set when the least significant byte of the destination has an even number of 1 bits, immediately after an arithmetic or Boolean instruction has executed. It get cleared when the result have odd number of 1 bits.

Consider following example, Result of adding two bytes have 2 bits set to one, which is even and the PF=1. If we add another byte in it, the resulting byte will have three 1 bits, making PF=0.


0 0 0 1 0 0 0 0 a
0 0 0 0 0 1 0 0 b
0 0 0 1 0 1 0 0 a + b No. of 1 = Even, PF=1
0 0 0 0 0 0 0 1 d
0 0 0 1 0 1 0 1 a + b + d No. of 1 = odd , PF=0

Check the code of this example on the next page.

Example – Parity Flags

```
INCLUDE Irvine32.inc
.data
a BYTE 010000b
b BYTE 000100b
d BYTE 000001b
.code
main PROC
      Mov AL,a
      Add AL,b
      Call DumpRegs
      Add AL,d
      Call DumpRegs
      Call ReadInt
      Exit
main ENDP
END main
```