Sign and Zero Extension

TABLE 4-1 Instruction Operand Notation, 32-Bit Mode.

Operand	Description
reg8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
reg16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
reg32	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
reg	Any general-purpose register
sreg	16-bit segment register: CS, DS, SS, ES, FS, GS
imm	8-, 16-, or 32-bit immediate value
imm8	8-bit immediate byte value
imm16	16-bit immediate word value
imm32	32-bit immediate doubleword value
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
reg/mem32	32-bit operand, which can be a 32-bit general register or memory doubleword
mem	An 8-, 16-, or 32-bit memory operand

I'll explain zero and sign extension in the context of moving smaller values to larger registers and provide simple assembly code examples.

ZERO EXTENSION:

Zero extension is a technique used when you want to copy a smaller unsigned value (like an unsigned integer) to a larger register.

To do this, you set the larger register to zero and then move the smaller value into the appropriate portion of the larger register.

Here's an example in simple assembly code:

```
.data
    count WORD 1    ; A 16-bit unsigned integer

.code
    mov ecx, 0      ; Set ECX to zero
    mov cx, count   ; Move the 16-bit value 'count' to the lower 16 bits of ECX
```

In this example, we set ECX to all ones in 32 bits, which represents a negative zero. Then, we move the 16-bit signed value signedVal to CX, which sign-extends it to fill the upper 16 bits of ECX while preserving the negative sign.

We start with count declared as a 16-bit unsigned integer with the value 1 (hex: 0001h).

mov ecx, 0 sets the entire 32-bit ECX register to zero:

```
ECX = 00000000h
```

mov cx, count moves the 16-bit value from count into the lower 16 bits of ECX, leaving the upper 16 bits as zero:

```
ECX = 00010000h
```

Here's a step-by-step explanation of what's happening:

• mov cx, count copies the value from count (16 bits) into the lower 16 bits of ECX. So, count's value, 0001h, is placed in the lower 16 bits of ECX.

• The upper 16 bits of ECX remain zero because we only moved a 16-bit value. So, the final value of ECX is 00010000h.

In simple math terms, we've copied the number 1 (hex: 0001h) into ECX and left-padded it with zeroes in the upper 16 bits. This results in 00010000h, which is 16 in decimal. So, after executing the code, the value in ECX is 16.

MOVZX and MOVSX Instructions

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

While the manual methods described above work, Intel processors provide dedicated instructions for these operations:

- MOVZX (Move with Zero Extension) for unsigned values.
- MOVSX (Move with Sign Extension) for signed values.

Here's how you would use these instructions:

```
.data
    count WORD 1      ; A 16-bit unsigned integer
    signedVal SWORD -16    ; A 16-bit signed integer with a value of -16 (hex: FFF0)

.code
    movzx ecx, word ptr [count]   ; Copy 'count' to ECX with zero extension
    movsx ecx, word ptr [signedVal] ; Copy 'signedVal' to ECX with sign extension
```

These instructions automatically perform zero or sign extension as needed, making your code more concise and efficient.

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

MOVZX is used when you want to copy a smaller unsigned value (like an unsigned integer) to a larger register, extending the value with zeroes.

```
.data
    count WORD 1     ; A 16-bit unsigned integer

.code
    mov ecx, 0       ; Set ECX to zero
    movzx cx, count ; Move 'count' to CX with zero extension
```

mov ecx, 0 sets ECX to zero.

```
ECX = 00000000h
```

movzx cx, count moves the 16-bit value from count into the lower 16 bits of CX with zero extension. The upper 16 bits remain zero:

```
CX = 0001h
```

Here, MOVZX explicitly extends the value of count with zero bits, making sure that the upper bits are set to zero.

MOVSX is used when you want to copy a smaller signed value (like a signed integer) to a larger register, extending the value while preserving its sign.

Example using MOVSX:

```
.data
    signedVal SWORD -16    ; A 16-bit signed integer with a value of -16 (hex: FFF0)

.code
    mov ecx, 0FFFFFFFFh    ; Set ECX to all ones in 32 bits (negative zero)
    movsx cx, signedVal    ; Move 'signedVal' to CX with sign extension
```

mov ecx, 0FFFFFFFFh sets ECX to all ones in 32 bits, representing negative zero:

```
ECX = FFFFFFFFh
```

movsx cx, signedVal moves the 16-bit signed value from signedVal to CX with sign extension. The upper bits are filled with the sign bit, maintaining the negative sign:

```
CX = FFF0h
```

In this case, MOVSX sign-extends the value of signedVal to fill the upper bits while preserving its negative sign.

These instructions (MOVZX and MOVSX) are useful because they handle the extension automatically, simplifying code and ensuring correct behavior when working with smaller values in larger registers.

In the example of movsx, must signed values be negative?

No, signed values don't have to be negative for MOVSX (Move with Sign Extension) to work.

MOVSX is used to sign-extend signed values, which means it preserves the sign of the value while extending it to fill a larger register. Whether the value is positive or negative, MOVSX will correctly perform the sign extension.

For example, let's say you have a 16-bit signed value that is positive:

```
.data
    signedVal SWORD 42    ; A 16-bit signed integer with a value of 42

;You can use MOVSX with this positive value, and it will correctly sign-extend it:

.code
    mov ecx, 0FFFFFFFFh    ; Set ECX to all ones in 32 bits (negative zero)
    movsx cx, signedVal    ; Move 'signedVal' to CX with sign extension
```

In this case, MOVSX will extend the positive value 42 to 0036h while preserving its positive sign.

EXPLANATION 1:

Here, signedVal is declared as a 16-bit signed integer with a value of 42. It's important to note that 42 is a positive value. Now, let's look at the code:

In this code, we first set ECX to all ones in 32 bits (0FFFFFFFFh), which represents negative zero in two's complement notation.

Then, we use MOVSX to move the value of signedVal (which is 42) into the lower 16 bits of CX while performing sign extension. Here's what happens step by step:

mov ecx, 0FFFFFFFFh sets ECX to all ones in 32 bits, representing negative zero:

```
ECX = FFFFFFFFh
```

movsx cx, signedVal moves the signed value 42 (hex: 002Ah) from signedVal to the lower 16 bits of CX while performing sign extension.

The sign bit of the source operand (signedVal) is preserved in the upper bits of CX. Since 42 is positive, the sign bit is 0, and the upper bits are filled with 0s:

```
CX = 002Ah
```

So, even though we started with a positive value (42), when using MOVSX, the sign extension ensures that the value in CX is correctly represented as 002Ah in 16 bits.

The sign bit is preserved, and the upper bits are filled accordingly.

So, MOVSX is used for both positive and negative signed values, ensuring that the sign is correctly preserved during the extension.

REPEAT EXPLANATION:

When we perform sign extension with MOVSX, the goal is to extend the sign bit of the source value to fill the upper bits of the destination register while preserving the value's sign. Here's a step-by-step explanation for the code:

```
.data
    signedVal SWORD 42    ; A 16-bit signed integer with a value of 42
```

signedVal is declared as a 16-bit signed integer with the **decimal value 42**, which in **hexadecimal is 002Ah.** It's important to note that 42 is a **positive value.**

```
.code
mov ecx, 0FFFFFFFFh    ; Set ECX to all ones in 32 bits (negative zero)
movsx cx, signedVal    ; Move 'signedVal' to CX with sign extension
```

sets ECX to all ones in 32 bits, representing negative zero in two's complement notation:

```
mov ecx, 0FFFFFFFFh
```

movsx cx, signedVal moves the signed value 42 (hex: 002Ah) from signedVal to the lower 16 bits of CX while performing sign extension. The sign bit of the source operand (signedVal) is preserved in the upper bits of CX. Since 42 is a positive value, its sign bit is 0, and the upper bits of CX are indeed filled with 0s:

```
CX = 002Ah
```

In this case, the sign extension correctly preserves the positive sign of 42, and the upper bits are filled with 0s to maintain a positive value in CX.

MORE DETAIL:

Then why do we need movzx if movsx can do both?

That's a great question! While both MOVZX (Move with Zero Extension) and MOVSX (Move with Sign Extension) can extend the value of a smaller operand to a larger one, they serve different purposes and have distinct use cases:

MOVZX (Move with Zero Extension):

• Purpose: MOVZX is primarily used when you want to extend an unsigned value (like an unsigned integer) to a larger register, and you want the upper bits to be set to zero.

• Example: When you have an unsigned value count and you want to extend it to a larger register, you can use MOVZX to ensure that the upper bits are zero-filled. This is useful when you don't want any sign-related information.

MOVSX (Move with Sign Extension):

• Purpose: MOVSX is used when you want to extend a signed value (like a signed integer) to a larger register, and you want to preserve the

sign of the original value.

• Example: When you have a signed value signedVal and you want to extend it to a larger register, you can use MOVSX to maintain the sign bit in the upper bits. This is essential when dealing with signed data to ensure that the sign is correctly preserved.

In summary, the choice between MOVZX and MOVSX depends on the nature of the data you're working with:

• Use MOVZX when dealing with unsigned values and you want to ensure that the upper bits are zero.
• Use MOVSX when dealing with signed values and you want to preserve the sign while extending the value.

Each instruction is designed to handle these scenarios correctly and efficiently, making it easier to work with different types of data in assembly language.

When you're performing sign extension with instructions like MOVSX, you need to consider the sign of the number in hexadecimal representation.

Here's a simple guideline:
• If the number in hexadecimal representation is positive (e.g., 002Ah for 42), the sign bit is 0. In this case, extending it with zeros in the upper bits preserves the positive sign.

• If the number in hexadecimal representation is negative (e.g., FFFAh for -6), the sign bit is 1. In this case, extending it with ones in the upper bits preserves the negative sign.

By extending the value with the appropriate sign bit, you ensure that the sign is correctly preserved when moving or copying the value to a larger register. This is crucial when working with signed integers in assembly language.

****
****

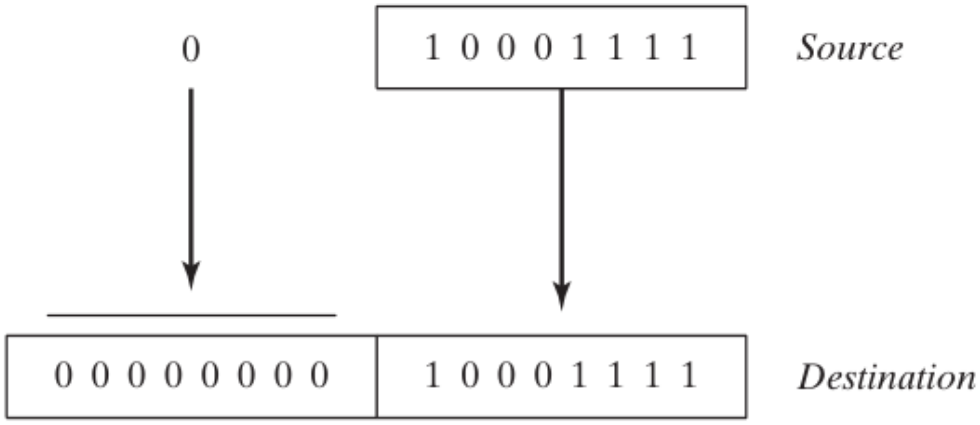
movzx repeated 3:

```
.data
    byteVal BYTE 10001111b     ;decimal 143

.code
    movzx ax,byteVal           ; AX = 0000000010001111b
```

What is happening here:

FIGURE 4–1 Using MOVZX to copy a byte into a 16-bit destination.



MOVZX (Move with Zero Extension) is used to copy a smaller operand into a larger one while zero-filling the upper bits of the larger operand. Let's go through the examples step by step:

Example 1: Register Operands

```
mov bx, 0A69Bh      ; BX = 0A69Bh
movzx eax, bx       ; EAX = 0000A69Bh
movzx edx, bl       ; EDX = 0000009Bh
movzx cx, bl        ; CX = 009Bh
```

In this example, we use the MOVZX instruction to move data from smaller registers (bl and bx) into larger ones (eax, edx, and cx).

1. mov bx, 0A69Bh loads the 16-bit value 0A69Bh into the bx register.

2. movzx eax, bx extends the 16-bit value in bx into a 32-bit value in eax, zero-filling the upper 16 bits. So, EAX becomes 0000A69Bh.

3. movzx edx, bl extends the 8-bit value in the lower byte of bx (bl) into a 32-bit value in edx, zero-filling the upper 24 bits. EDX becomes 0000009Bh.

4. movzx cx, bl extends the 8-bit value in bl into a 16-bit value in cx, zero-filling the upper 8 bits. CX becomes 009Bh.

Example 2: Memory Operands

```
.data
    byte1 BYTE 9Bh
    word1 WORD 0A69Bh

.code
    movzx eax, word1        ; EAX = 0000A69Bh
    movzx edx, byte1        ; EDX = 0000009Bh
    movzx cx, byte1         ; CX = 009Bh
```

In this example, we use memory operands (variables byte1 and word1) as the sources for MOVZX.

1. movzx eax, word1 extends the 16-bit value stored in word1 into a 32-bit value in eax, zero-filling the upper 16 bits. So, EAX becomes 0000A69Bh.

2. movzx edx, byte1 extends the 8-bit value stored in byte1 into a 32-bit value in edx, zero-filling the upper 24 bits. EDX becomes 0000009Bh.

3. movzx cx, byte1 extends the 8-bit value stored in byte1 into a 16-bit value in cx, zero-filling the upper 8 bits. CX becomes 009Bh.

In both examples, MOVZX ensures that the larger registers or memory locations are filled with zeros in the upper bits, maintaining the

integrity of the original value and preventing unintended sign extension.

****
****

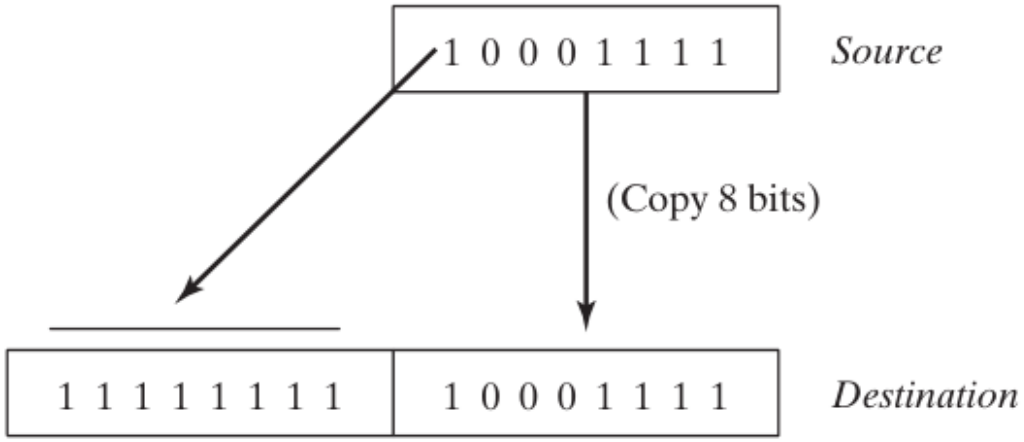
movsx repeated 3:

```
;movsx

mov bx, 0A69Bh      ; BX = 0A69Bh
movsx eax, bx        ; EAX = FFFFA69Bh
movsx edx, bl        ; EDX = FFFFFF9Bh
movsx cx, bl         ; CX = FF9Bh
```

What is happening here?

FIGURE 4–2 Using MOVSX to copy a byte into a 16-bit destination.



In these examples, you're dealing with hexadecimal constants and sign extension. Here's what happens step by step:

1. mov bx, 0A69Bh loads the 16-bit hexadecimal value 0A69Bh into the bx register. The leading "A" digit indicates that the highest bit (sign bit) is set.

2. movsx eax, bx extends the 16-bit value in bx into a 32-bit value in eax, while preserving the sign bit. Since the sign bit is set (due to the leading "A" digit), sign extension fills the upper 16 bits

with ones (F in hexadecimal), resulting in EAX = FFFFA69Bh.

3. movsx edx, bl extends the 8-bit value in the lower byte of bx (bl) into a 32-bit value in edx, while preserving the sign bit. Again, the sign bit is set, so the upper 24 bits are filled with ones, resulting in EDX = FFFFFF9Bh.

4. movsx cx, bl extends the 8-bit value in bl into a 16-bit value in cx, while preserving the sign bit. The sign bit is still set, so the upper 8 bits are filled with ones, resulting in CX = FF9Bh.

In these examples, the "A" in the hexadecimal constant 0A69Bh indicates that it's a negative value due to the set sign bit. Sign extension is applied accordingly to maintain the correct sign and value representation in the larger registers.