**Computer Organization & Assembly Language**

**Lab 07**

**Topics:**

1. **AND instruction**
2. **OR instruction**
3. **NOT instruction**
4. **TEST instruction**
5. **CMP instruction**

**AND Instruction:**

**Definition:** The AND instruction is a bitwise logical operation that performs a bitwise AND operation between the bits of two operands, resulting in a new value where each bit is set to 1 only if both corresponding bits in the operands are 1.

**Explanation:** The AND instruction is used to manipulate individual bits within data. It takes two operands, the source, and the destination, and performs a bitwise AND operation. The result is stored in the destination operand.

**Examples:**

**Example 1:** Let's say we have two 8-bit binary numbers, A = 11001100 and B = 10101010. When we perform the AND operation between them, we get the result C = 10001000.

**Example 2:** Consider another example with A = 11110000 and B = 11001100. When we apply the AND operation, the result is D = 11000000.

**Example 3:** Clearing bits in a register or memory location. For example, to clear the high-order bits of a register, you could AND it with a mask of all 0s.

; Clear the high-order bits of the AL register.

**and al, 0fh**

**OR Instruction:**

**Definition:** The OR instruction is a bitwise logical operation that performs a bitwise OR operation between the bits of two operands, resulting in a new value where each bit is set to 1 if at least one of the corresponding bits in the operands is 1.

**Explanation:** The OR instruction is used to combine individual bits within data. It takes two operands, the source, and the destination, and performs a bitwise OR operation. The result is stored in the destination operand.

**Examples:**

**Example 1:** Let's say we have two 8-bit binary numbers, A = 11001100 and B = 10101010. When we perform the OR operation between them, we get the result C = 11101110.

**Example 2:** Consider another example with A = 11110000 and B = 11001100. When we apply the OR operation, the result is D = 11111100.

**Example 3:** Setting bits in a register or memory location. For example, to set the high-order bits of a register, you could OR it with a mask of all 1s.

; Set the high-order bits of the AL register.

**or al, 0xf0**

**NOT Instruction:**

**Definition:** The NOT instruction is a bitwise logical operation that performs a bitwise negation operation on each bit of the operand, flipping 0s to 1s and 1s to 0s.

**Explanation:** The NOT instruction is used to invert or complement individual bits within data. It takes a single operand and negates all its bits, creating a new value.

**Examples:**

**Example 1:** Let's say we have an 8-bit binary number A = 11001100. When we perform the NOT operation on it, we get the result B = 00110011.

**Example 2:** Consider another example with an 8-bit binary number C = 11110000. When we apply the NOT operation, the result is D = 00001111.

**Example 3:**

; Invert the value of the AL register.

not al

; Create a mask of all 0s.

mov ax, 0xffff

not ax

**TEST Instruction:**

**Definition:** The TEST instruction is a bitwise logical operation that performs a bitwise AND operation between the bits of two operands, similar to the AND instruction. However, it does not store the result but only sets the condition flags in the CPU based on the result.

**Explanation:** The TEST instruction is used to check the bits of two operands without changing them. It takes two operands, the source, and the destination, and performs a bitwise AND operation. The result is not stored, but the condition flags (such as zero flag) are updated based on the result.

**Examples:**

**Example 1:** Let's say we have two 8-bit binary numbers, A = 11001100 and B = 10101010. When we perform the TEST operation between them, the condition flags are updated based on the result without storing it.

**Example 2:** Consider another example with A = 11110000 and B = 11001100. The TEST operation updates the condition flags based on the bitwise AND result.

**CMP (compare) instruction:**

The CMP (compare) instruction performs an implied subtraction of a source operand from a destination operand. Neither operand is modified:

CMP destination, source

CMP uses the same operand combinations as the AND instruction. Flags The CMP instruction changes the Overflow, Sign, Zero, Carry, Auxiliary Carry, and Parity flags according to the value the destination operand would have had if actual subtraction had taken place.

**conditional jump**

a conditional jump instruction tests the flags and causes a branch to a new address. Let’s look at a couple of examples.

**Example 1**

The CMP instruction in the following example compares EAX to Zero. The JZ (jump if Zero) instruction jumps to label L1 if the Zero flag was set by the CMP instruction:

cmp eax,0

jz L1 ; jump if ZF = 1

.

.

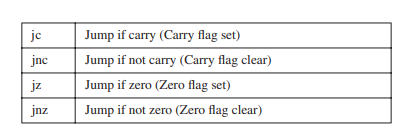
L1:

**Jcond Instruction**

A conditional jump instruction branches to a destination label when a status flag condition is true. Otherwise, if the flag condition is false, the instruction immediately following the conditional jump is executed. The syntax is as follows:

Jcond destination

cond refers to a flag condition identifying the state of one or more flags. The following examples are based on the Carry and Zero flags:



**Types of Conditional Jump Instructions:**

The x86 instruction set has a large number of conditional jump instructions. They are able to

compare signed and unsigned integers and perform actions based on the values of individual

CPU flags. The conditional jump instructions can be divided into four groups:

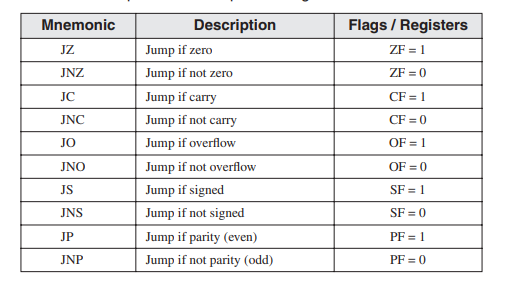
• Jumps based on specific flag values

• Jumps based on equality between operands or the value of (E)CX

• Jumps based on comparisons of unsigned operands

• Jumps based on comparisons of signed operands

Table below shows a list of jumps based on the Zero, Carry, Overflow, Parity, and Sign flags.



**Equality Comparisons:**

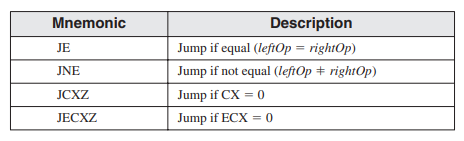
In some cases, two operands are compared; in other cases, a jump is taken based on the value of CX or ECX. In the table, the notations leftOp and rightOp refer to the left (destination) and right (source) operands in y. In some cases, two operands are compared; in other cases, a jump is taken based on the value of CX or ECX. In the table, the notations

leftOp and rightOp refer to the left (destination) and right (source) operands in a CMP instruction:

CMP leftOp,rightOp

The operand names reflect the ordering of operands for relational operators in algebra. For

example, in the expression X < Y, X is called leftOp and Y is called rightOp. a CMP instruction: CMP leftOp,rightOp The operand names reflect the ordering of operands for relational operators in algebra. For example, in the expression X < Y, X is called leftOp and Y is called rightOp.



Although the JE instruction is equivalent to JZ (jump if Zero) and JNE is equivalent to JNZ

(jump if not Zero), it’s best to select the mnemonic (JE or JZ) that best indicates your intention

to either compare two operands or examine a specific status flag.

Following are code examples that use the JE, JNE, JCXZ, and JECXZ instructions. Examine

the comments carefully to be sure that you understand why the conditional jumps were (or were

not) taken.

**Example 1:**

mov edx,0A523h

cmp edx,0A523h

jne L5 ; jump not taken

je L1 ; jump is taken

**Example 2:**

mov bx,1234h

sub bx,1234h

jne L5 ; jump not taken

je L1 ; jump is taken

**Example 3:**

mov cx,0FFFFh

inc cx

jcxz L2 ; jump is taken

**Example 4:**

xor ecx,ecx

jecxz L2 ; jump is taken

**Tasks:**

**Task 1: Separating Odd and Even Elements in an Array**

*Objective: Create two arrays of 10 random elements each. Loop through the arrays and separate the odd and even numbers.*

**Task 2: Separating Negative Elements in an Array**

*Objective: Create an array of 10 elements, each ranging from -128 to 127. Loop through the array and isolate the negative elements.*

**Task 3: Modify the higher bits in 16-bit registers**

a) Write a single instruction using 16-bit operands that clears the high 8 bits of AX and does not change the low 8 bits.

b) Write a single instruction using 16-bit operands that sets the high 8 bits of AX and does not change the low 8 bits.

**Task 4: Finding the Smallest Element in an Array**

*Objective: Consider an array of 10 elements. Loop through the array and determine the smallest element. Store the result in AX.*

**Task 5: Finding the Highest Element in an Array**

*Objective: Consider an array of 10 elements. Loop through the array and identify the highest element. Store the result in AX.*