Lecture No.14

Lecture Outlines

6.1 Conditional Branching

A programming language that permits decision making lets you alter the flow of control, using a technique known as *conditional branching*. Every IF statement, switch statement, or conditional loop found in high-level languages has built-in branching logic. Assembly language, as primitive as it is, provides all the tools you need for decision-making logic. In this chapter, we will see how the translation works, from high-level conditional statements to low-level implementation code.

Programs that deal with hardware devices must be able to manipulate individual bits in numbers. Individual bits must be tested, cleared, and set. Data encryption and compression also rely on bit manipulation. We will show how to perform these operations in assembly language.

6.2 Boolean and Comparison Instructions

In Chapter 1, we introduced the four basic operations of boolean algebra: AND, OR, XOR, and NOT. These operations can be carried out at the binary bit level, using assembly language instructions. These operations are also important at the boolean expression level, in IF statements, for example. First, we will look at the bitwise instructions. The techniques used here could be used to manipulate control bits for hardware devices, implement communication protocols, or encrypt data, just to name a few applications. The Intel instruction set contains the AND, OR, XOR, and NOT instructions, which directly implement boolean operations on binary bits, shown in Table 6-1. In addition, the TEST instruction is a nondestructive AND operation.

Table 6-1 Selected Boolean Instructions.

Operation	Description
AND	Boolean AND operation between a source operand and a destination operand.
OR	Boolean OR operation between a source operand and a destination operand.
XOR	Boolean exclusive-OR operation between a source operand and a destination operand.
NOT	Boolean NOT operation on a destination operand.
TEST	Implied boolean AND operation between a source and destination operand, setting the CPU flags appropriately.

6.2.1 The CPU Status Flags

Boolean instructions affect the Zero, Carry, Sign, Overflow, and Parity flags. Here's a quick review of their meanings:

- The Zero flag is set when the result of an operation equals zero.
- The Carry flag is set when an operation generates a carry out of the highest bit of the destination operand.
- The Sign flag is a copy of the high bit of the destination operand, indicating that it is negative if *set* and positive if *clear*. (Zero is assumed to be positive.)
- The Overflow flag is set when an instruction generates a result that is outside the signed range of the destination operand.
- The Parity flag is set when an instruction generates an even number of 1 bits in the low byte of the destination operand.

6.2.2 AND Instruction

The AND instruction performs a boolean (bitwise) AND operation between each pair of matching bits in two operands and places the result in the destination operand:

```
AND  destination,source
```

The following operand combinations are permitted, although immediate opperands can be no larger than 32 bits:

```
AND reg,reg
AND reg,mem
AND reg,imm
AND mem,reg
AND mem,imm
```

The operands can be 8, 16, 32, or 64 bits, and they must be the same size. For each matching bit in the two operands, the following rule applies: If both bits equal 1, the result bit is 1; otherwise, it is 0. The following truth table from Chapter 1 labels the input bits x and y. The third column shows the value of the expression $\mathbf{x} \wedge \mathbf{y}$:

x	y	x ∧ y
0	0	0
0	1	0
1	0	0
1	1	1

The AND instruction lets you clear 1 or more bits in an operand without affecting other bits. The technique is called bit *masking*, much as you might use masking tape when painting a house to cover areas (such as windows) that should not be painted. Suppose, for example, that a control byte is about to be copied from the AL register to a hardware device. Further, we will assume that the device resets itself when bits 0 and 3 are cleared in the control byte. Assuming that we want to reset the device without modifying any other bits in AL, we can write the following:

```
and AL,11110110b    ; clear bits 0 and 3, leave others unchanged
```

For example, suppose AL is initially set to 10101110 binary. After ANDing it with 11110110, AL equals 10100110:

```
mov al,10101110b
and al,11110110b              ; result in AL = 10100110
```

Flags The AND instruction always clears the Overflow and Carry flags. It modifies the Sign, Zero, and Parity flags in a way that is consistent with the value assigned to the destination operand. For example, suppose the following instruction results in a value of Zero in the EAX register. In that case, the Zero flag will be set:

```
and eax,1Fh
```

Converting Characters to Upper case

The AND instruction provides an easy way to translate a letter from lowercase to uppercase. If we compare the ASCII codes of capital **A** and lowercase **a**, it becomes clear that only bit 5 is different:

```
0 1 1 0 0 0 0 1 = 61h ('a')
0 1 0 0 0 0 0 1 = 41h ('A')
```

The rest of the alphabetic characters have the same relationship. If we AND any character with 11011111 binary, all bits are unchanged except for bit 5, which is cleared. In the following example, all characters in an array are converted to uppercase:

```
.data
array BYTE 50 DUP(?)
.code
    mov   ecx,LENGTHOF array
    mov   esi,OFFSET array
L1: and   BYTE PTR [esi],11011111b    ; clear bit 5
    inc   esi
    loop  L1
```

6.2.3 OR Instruction

The OR instruction performs a boolean OR operation between each pair of matching bits in two operands and places the result in the destination operand:

```
OR  destination,source
```

The OR instruction uses the same operand combinations as the AND instruction:

```
OR reg,reg
OR reg,mem
OR reg,imm
OR mem,reg
OR mem,imm
```

The operands can be 8, 16, 32, or 64 bits, and they must be the same size. For each matching bit in the two operands, the output bit is 1 when at least one of the input bits is 1. The following truth table (from Chapter 1) describes the boolean expression **x** ∨ **y**:

x	y	x ∨ y
0	0	0
0	1	1
1	0	1
1	1	1

The OR instruction is particularly useful when you need to set 1 or more bits in an operand without affecting any other bits. Suppose, for example, that your computer is attached to a servo motor, which is activated by setting bit 2 in its control byte. Assuming that the AL register contains a control byte in which each bit contains some important information, the following code only sets the bit in position 2:

```
or AL,00000100b                 ; set bit 2, leave others unchanged
```

For example, if AL is initially equal to 11100011 binary and then we OR it with 00000100, the result equals 11100111:

```
mov al,11100011b
or  al,00000100b                ; result in AL = 11100111
```

Flags The OR instruction always clears the Carry and Overflow flags. It modifies the Sign, Zero, and Parity flags in a way that is consistent with the value assigned to the destination operand. For example, you can OR a number with itself (or zero) to obtain certain information about its value:

```
or  al,al
```

The values of the Zero and Sign flags indicate the following about the contents of AL:

Zero Flag	Sign Flag	Value in AL Is . . .
Clear	Clear	Greater than zero
Set	Clear	Equal to zero
Clear	Set	Less than zero

6.2.4 Bit-Mapped Sets

Some applications manipulate sets of items selected from a limited-sized universal set. Examples might be employees within a company, or environmental readings from a weather monitoring station. In such cases, binary bits can indicate set membership. Rather than holding pointers or references to objects in a container such as a Java HashSet, an application can use a *bit vector* (or bit map) to map the bits in a binary number to an array of objects.

For example, the following binary number uses bit positions numbered from 0 on the right to 31 on the left to indicate that array elements 0, 1, 2, and 31 are members of the set named **SetX**:

```
SetX = 10000000 00000000 00000000 00000111
```

(The bytes have been separated to improve readability.) We can easily check for set membership by ANDing a particular member's bit position with a 1:

```
mov  eax,SetX
and  eax,10000b                ; is element[4] a member of SetX?
```

If the AND instruction in this example clears the Zero flag, we know that element [4] is a member of SetX.

Set Complement

The complement of a set can be generated using the NOT instruction, which reverses all bits. Therefore, the complement of the SetX that we introduced is generated in EAX using the following instructions:

```
mov  eax,SetX
not  eax                       ; complement of SetX
```

Set Intersection

The AND instruction produces a bit vector that represents the intersection of two sets. The following code generates and stores the intersection of SetX and SetY in EAX:

```
mov  eax,SetX
and  eax,SetY
```

This is how the intersection of SetX and SetY is produced:

```
         10000000000000000000000000000111    (SetX)
AND      10000010101000000000011101100011    (SetY)
-------------------------------------------------------
         10000000000000000000000000000011    (intersection)
```

It is hard to imagine any faster way to generate a set intersection. A larger domain would require more bits than could be held in a single register, making it necessary to use a loop to AND all of the bits together.

Set Union

The OR instruction produces a bit map that represents the union of two sets. The following code generates the union of SetX and SetY in EAX:

```
mov  eax,SetX
or   eax,SetY
```

This is how the union of SetX and SetY is generated by the OR instruction:

```
          10000000000000000000000000000111     (SetX)
OR        10000010101000000000011101100011     (SetY)
-------------------------------------------------------
          10000010101000000000011101100111     (union)
```

6.2.5 XOR Instruction

The XOR instruction performs a boolean exclusive-OR operation between each pair of matching bits in two operands and stores the result in the destination operand:

```
XOR  destination,source
```

The XOR instruction uses the same operand combinations and sizes as the AND and OR instructions. For each matching bit in the two operands, the following applies: If both bits are the same (both 0 or both 1), the result is 0; otherwise, the result is 1. The following truth table describes the boolean expression **x** ⊕ **y**:

x	y	x ⊕ y
0	0	0
0	1	1
1	0	1
1	1	0

A bit exclusive-ORed with 0 retains its value, and a bit exclusive-ORed with 1 is toggled (complemented). XOR reverses itself when applied twice to the same operand. The following truth table shows that when bit *x* is exclusive-ORed with bit *y* twice, it reverts to its original value:

x	y	x ⊕ y	(x ⊕ y) ⊕ y
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

As you will find out in Section 6.3.4, this "reversible" property of XOR makes it an ideal tool for a simple form of symmetric encryption.

Flags The XOR instruction always clears the Overflow and Carry flags. XOR modifies the Sign, Zero, and Parity flags in a way that is consistent with the value assigned to the destination operand.

Checking the Parity Flag Parity checking is a function performed on a binary number that counts the number of 1 bits contained in the number; if the resulting count is even, we say that the data has even parity; if the count is odd, the data has odd parity. In x86 processors, the Parity

flag is set when the lowest byte of the destination operand of a bitwise or arithmetic operation has even parity. Conversely, when the operand has odd parity, the flag is cleared. An effective way to check the parity of a number without changing its value is to exclusive-OR the number with zero:

```
mov  al,10110101b              ; 5 bits = odd parity
xor  al,0                      ; Parity flag clear (odd)
mov  al,11001100b              ; 4 bits = even parity
xor  al,0                      ; Parity flag set (even)
```

Visual Studio uses PE = 1 to indicate even parity, and PE = 0 to indicate odd parity.

16-Bit Parity You can check the parity of a 16-bit integer by performing an exclusive-OR between the upper and lower bytes:

```
mov  ax,64C1h                  ; 0110 0100 1100 0001
xor  ah,al                     ; Parity flag set (even)
```

Imagine the set bits (bits equal to 1) in each register as being members of an 8-bit set. The XOR instruction zeroes all bits belonging to the intersection of the sets. XOR also forms the union between the remaining bits. The parity of this union will be the same as the parity of the entire 16-bit integer.

What about 32-bit values? If we number the bytes from B_0 through B_3, we can calculate the parity as B_0 XOR B_1 XOR B_2 XOR B_3.

6.2.6 NOT Instruction

The NOT instruction toggles (inverts) all bits in an operand. The result is called the *one's complement*. The following operand types are permitted:

```
NOT reg
NOT mem
```

For example, the one's complement of F0h is 0Fh:

```
mov  al,11110000b
not  al                        ; AL = 00001111b
```

Flags No flags are affected by the NOT instruction.

6.2.7 TEST Instruction

The TEST instruction performs an implied AND operation between each pair of matching bits in two operands and sets the Sign, Zero, and Parity flags based on the value assigned to the destination operand. The only difference between TEST and AND is that TEST does not modify the destination operand. The TEST instruction permits the same operand combinations as the AND instruction. TEST is particularly valuable for finding out whether individual bits in an operand are set.

Example: Testing Multiple Bits The TEST instruction can check several bits at once. Suppose we want to know whether bit 0 or bit 3 is set in the AL register. We can use the following instruction to find this out:

```
test al,00001001b;              test bits 0 and 3
```

(The value 00001001 in this example is called a *bit mask*.) From the following example data sets, we can infer that the Zero flag is set only when all tested bits are clear:

```
0 0 1 0 0 1 0 1  <- input value
0 0 0 0 1 0 0 1  <- test value
0 0 0 0 0 0 0 1  <- result: ZF = 0

0 0 1 0 0 1 0 0  <- input value
0 0 0 0 1 0 0 1  <- test value
0 0 0 0 0 0 0 0  <- result: ZF = 1
```

Flags The TEST instruction always clears the Overflow and Carry flags. It modifies the Sign, Zero, and Parity flags in the same way as the AND instruction.

6.2.8 CMP Instruction

Having examined all of the bitwise instructions, let's now turn to instructions used in logical (boolean) expressions. The most common boolean expressions involve some type of comparison. The following pseudocode snippets support this idea:

```
if A > B ...
while X > 0 and X < 200  ...
if check_for_error( N ) = true
```

In x86 assembly language we use the CMP instruction to compare integers. Character codes are also integers, so they work with CMP as well. Floating-point values require specialized comparison instructions, which we cover in Chapter 12.

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. When two unsigned operands are compared, the Zero and Carry flags indicate the following relations between operands:

CMP Results	ZF	CF
Destination < source	0	1
Destination > source	0	0
Destination = source	1	0

When two signed operands are compared, the Sign, Zero, and Overflow flags indicate the following relations between operands:

CMP Results	Flags
Destination < source	SF ≠ OF
Destination > source	SF = OF
Destination = source	ZF = 1

CMP is a valuable tool for creating conditional logic structures. When you follow CMP with a conditional jump instruction, the result is the assembly language equivalent of an IF statement.

Examples Let's look at three code fragments showing how flags are affected by the CMP instruction. When AX equals 5 and is compared to 10, the Carry flag is set because subtracting 10 from 5 requires a borrow:

```
mov  ax,5
cmp  ax,10                      ; ZF = 0 and CF = 1
```

Comparing 1000 to 1000 sets the Zero flag because subtracting the source from the destination produces zero:

```
mov  ax,1000
mov  cx,1000
cmp  cx,ax                      ; ZF = 1 and CF = 0
```

Comparing 105 to 0 clears both the Zero and Carry flags because subtracting 0 from 105 generates a positive, nonzero value.

```
mov  si,105
cmp  si,0                       ; ZF = 0 and CF = 0
```

6.2.9 Setting and Clearing Individual CPU Flags

How can you easily set or clear the Zero, Sign, Carry, and Overflow flags? There are several ways, some of which require modifying the destination operand. To set the Zero flag, TEST or AND an operand with Zero; to clear the Zero flag, OR an operand with 1:

```
test  al,0                      ; set Zero flag
and   al,0                      ; set Zero flag
or    al,1                      ; clear Zero flag
```

TEST does not modify the operand, whereas AND does. To set the Sign flag, OR the highest bit of an operand with 1. To clear the Sign flag, AND the highest bit with 0:

```
or   al,80h                     ; set Sign flag
and  al,7Fh                     ; clear Sign flag
```

To set the Carry flag, use the STC instruction; to clear the Carry flag, use CLC:

```
stc                             ; set Carry flag
clc                             ; clear Carry flag
```

To set the Overflow flag, add two positive values that produce a negative sum. To clear the Overflow flag, OR an operand with 0:

```
mov  al,7Fh                    ; AL = +127
inc  al                        ; AL = 80h (-128), OF=1
or   eax,0                     ; clear Overflow flag
```

6.2.10 Boolean Instructions in 64-Bit Mode

For the most part, 64-bit instructions work exactly the same in 64-Bit mode as they do in 32-bit mode. For example, if the source operand is a constant whose size is less than 32 bits and the destination is a 64-bit register or memory operand, all bits in the destination operand are affected:

```
.data
allones QWORD 0FFFFFFFFFFFFFFFFh
.code
    mov  rax,allones           ; RAX = FFFFFFFFFFFFFFFF
    and  rax,80h               ; RAX = 0000000000000080
    mov  rax,allones           ; RAX = FFFFFFFFFFFFFFFF
    and  rax,8080h             ; RAX = 0000000000008080
    mov  rax,allones           ; RAX = FFFFFFFFFFFFFFFF
    and  rax,808080h           ; RAX = 0000000000808080
```

But when the source operand is a 32-bit constant or register, only the lower 32 bits of the destination operand are affected. In the following example, only the lower 32 bits of RAX are modified:

```
mov  rax,allones               ; RAX = FFFFFFFFFFFFFFFF
and  rax,80808080h             ; RAX = FFFFFFFF80808080
```

The same results are true when the destination operand is a memory operand. Clearly, 32-bit operands are a special case that you must consider separately from other operand sizes.

6.3 Conditional Jumps

6.3.1 Conditional Structures

There are no explicit high-level logic structures in the x86 instruction set, but you can implement them using a combination of comparisons and jumps. Two steps are involved in executing a conditional statement: First, an operation such as CMP, AND, or SUB modifies the CPU status flags. Second, 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:
```

Example 2 The AND instruction in the following example performs a bitwise AND on the DL register, affecting the Zero flag. The JNZ (jump if not Zero) instruction jumps if the Zero flag is clear:

```
        and   dl,10110000b
        jnz   L2                    ; jump if ZF = 0
        .
        .
L2:
```

6.3.2 J*cond* 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:

JC	Jump if carry (Carry flag set)
JNC	Jump if not carry (Carry flag clear)
JZ	Jump if zero (Zero flag set)
JNZ	Jump if not zero (Zero flag clear)

CPU status flags are most commonly set by arithmetic, comparison, and boolean instructions. Conditional jump instructions evaluate the flag states, using them to determine whether or not jumps should be taken.

Using the CMP Instruction Suppose you want to jump to label L1 when EAX equals 5. In the next example, if EAX equals 5, the CMP instruction sets the Zero flag; then, the JE instruction jumps to L1 because the Zero flag is set:

```
cmp  eax,5
je   L1                          ; jump if equal
```

(The JE instruction always jumps based on the value of the Zero flag.) If EAX were not equal to 5, CMP would clear the Zero flag, and the JE instruction would not jump.

In the following example, the JL instruction jumps to label L1 because AX is less than 6:

```
mov  ax,5
cmp  ax,6
jl   L1                         ; jump if less
```

In the following example, the jump is taken because AX is greater than 4:

```
mov  ax,5
cmp  ax,4
jg   L1                         ; jump if greater
```

6.3.3 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 6-2 shows a list of jumps based on the Zero, Carry, Overflow, Parity, and Sign flags.

Table 6-2 Jumps Based on Specific Flag Values.

Mnemonic	Description	Flags / Registers
JZ	Jump if zero	ZF = 1
JNZ	Jump if not zero	ZF = 0
JC	Jump if carry	CF = 1
JNC	Jump if not carry	CF = 0
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if signed	SF = 1
JNS	Jump if not signed	SF = 0
JP	Jump if parity (even)	PF = 1
JNP	Jump if not parity (odd)	PF = 0

Equality Comparisons

Table 6-3 lists jump instructions based on evaluating equality. In some cases, two operands are compared; in other cases, a jump is taken based on the value of CX, ECX, or RCX. 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*.

Table 6-3 Jumps Based on Equality.

Mnemonic	Description
JE	Jump if equal (*leftOp* = *rightOp*)
JNE	Jump if not equal (*leftOp* ≠ *rightOp*)
JCXZ	Jump if CX = 0
JECXZ	Jump if ECX = 0
JRCXZ	Jump if RCX = 0 (64-bit mode)

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
```

Unsigned Comparisons

Jumps based on comparisons of unsigned numbers are shown in Table 6-4. The operand names reflect the order of operands, as in the expression (*leftOp* < *rightOp*). The jumps in Table 6-4 are only meaningful when comparing unsigned values. Signed operands use a different set of jumps.

Signed Comparisons

Table 6-5 displays a list of jumps based on signed comparisons. The following instruction sequence demonstrates the comparison of two signed values:

```
mov  al,+127                     ; hexadecimal value is 7Fh
cmp  al,-128                     ; hexadecimal value is 80h
ja   IsAbove                     ; jump not taken, because 7Fh < 80h
jg   IsGreater                   ; jump taken, because +127 > -128
```

The JA instruction, which is designed for unsigned comparisons, does not jump because unsigned 7Fh is smaller than unsigned 80h. The JG instruction, on the other hand, is designed for signed comparisons—it jumps because +127 is greater than −128.

In the following code examples, examine the comments to be sure you understand why the jumps were (or were not) taken:

Example 1

```
mov   edx,-1
cmp   edx,0
jnl   L5                    ; jump not taken (-1 >= 0 is false)
jnle  L5                    ; jump not taken (-1 > 0 is false)
jl    L1                    ; jump is taken (-1 < 0 is true)
```

Example 2

```
mov   bx,+32
cmp   bx,-35
jng   L5                    ; jump not taken (+32 <= -35 is false)
jnge  L5                    ; jump not taken (+32 < -35 is false)
jge   L1                    ; jump is taken (+32 >= -35 is true)
```

Example 3

```
mov ecx,0
cmp ecx,0
jg  L5                      ; jump not taken (0 > 0 is false)
jnl L1                      ; jump is taken (0 >= 0 is true)
```

Example 4

```
mov ecx,0
cmp ecx,0
jl  L5                      ; jump not taken (0 < 0 is false)
jng L1                      ; jump is taken (0 <= 0 is true)
```

Table 6-4 Jumps Based on Unsigned Comparisons.

Mnemonic	Description
JA	Jump if above (if *leftOp* > *rightOp*)
JNBE	Jump if not below or equal (same as JA)
JAE	Jump if above or equal (if *leftOp* ≥ *rightOp*)
JNB	Jump if not below (same as JAE)
JB	Jump if below (if *leftOp* < *rightOp*)
JNAE	Jump if not above or equal (same as JB)
JBE	Jump if below or equal (if *leftOp* ≤ *rightOp*)
JNA	Jump if not above (same as JBE)

Table 6-5 Jumps Based on Signed Comparisons.

Mnemonic	Description
JG	Jump if greater (if *leftOp* > *rightOp*)
JNLE	Jump if not less than or equal (same as JG)
JGE	Jump if greater than or equal (if *leftOp* ≥ *rightOp*)
JNL	Jump if not less (same as JGE)
JL	Jump if less (if *leftOp* < *rightOp*)
JNGE	Jump if not greater than or equal (same as JL)
JLE	Jump if less than or equal (if *leftOp* ≤ *rightOp*)
JNG	Jump if not greater (same as JLE)