Conditional Jumps

==================================

Conditional Structures in x86

==================================

x86 does not have explicit high-level logic structures in its instruction set, but you can implement them using a combination of comparisons and jumps. Two steps are involved in executing a conditional statement:

An operation such as CMP, AND, or SUB modifies the CPU status flags. A conditional jump instruction tests the flags and causes a branch to a new address.

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      ; Compare the value in EAX with 0
jz L1           ; Jump to label L1 if the Zero Flag (ZF) is set
```

Here's a breakdown:

cmp eax, 0: This instruction compares the value in the EAX register to zero. After this instruction, the Zero Flag (ZF) will be set if EAX is equal to zero.

jz L1: This is a conditional jump instruction. It checks the Zero Flag (ZF). If ZF is set (meaning the comparison result was zero), it jumps to the label L1.

So, in simple terms, this code checks if the value in the EAX register is zero. If it is, it jumps to L1. If not, it continues executing the code below the jz instruction.

```
and dl, 10110000b   ; Perform a bitwise AND operation on the DL register
jnz L2              ; Jump to label L2 if the Zero Flag (ZF) is not set
```

Here's the explanation:

and dl, 10110000b: This instruction performs a bitwise AND operation between the value in the DL register and the binary value 10110000. This operation affects the Zero Flag (ZF). If the result of the AND operation is zero, ZF will be cleared; otherwise, it will be set.

jnz L2: This is a conditional jump instruction, just like in the previous example. It checks the Zero Flag (ZF), but this time, it jumps to the label L2 if the ZF is not set (meaning the result of the AND operation was not zero).

So, in this code, if the result of the bitwise AND operation is not zero, it will jump to L2. Otherwise, it will continue with the code after the jnz instruction.

These conditional jumps are essential for controlling program flow in assembly language and are often used for implementing conditional statements like if-else constructs. If you have any more questions or need further clarification, feel free to ask!

It seems like you've provided information about conditional jump instructions in assembly language, specifically using the CMP instruction and various flag conditions. Let's break this down further.

Conditional jump instructions in assembly language allow you to branch to a destination label based on the state of certain CPU status flags.

These flags are commonly set by arithmetic, comparison, and boolean instructions. Here's a breakdown of some common conditional jump instructions:

JE (Jump if Equal): This instruction jumps to a destination label when the Zero flag is set, indicating that the compared values are equal.

```
cmp eax, 5
je L1 ; Jump to L1 if EAX equals 5
```

JC (Jump if Carry): Jumps to a destination label if the Carry flag is set, indicating that a carry occurred in an arithmetic operation.

JNC (Jump if Not Carry): Jumps to a destination label if the Carry flag is clear, indicating no carry occurred in an arithmetic operation.

JZ (Jump if Zero): Jumps to a destination label when the Zero flag is set, indicating that a value is zero.

JNZ (Jump if Not Zero): Jumps to a destination label when the Zero flag is clear, indicating that a value is not zero.

In your example, you're using the CMP instruction to compare the value in the EAX register to 5. If EAX equals 5, the Zero flag is set by the CMP instruction, and the JE instruction jumps to the label L1. If EAX is not equal to 5, the Zero flag is cleared, and the JE instruction does not jump.

Jumps Based on Specific Flag Values:

Conditional jumps in this group rely on the states of specific CPU flags to determine whether to take the jump. Here are some common conditional jumps based on specific flag values:

JE (Jump if Equal): Jumps when the Zero flag (ZF) is set, indicating that the compared values are equal.

JNE (Jump if Not Equal): Jumps when the Zero flag (ZF) is clear, indicating that the compared values are not equal.

JZ (Jump if Zero): Similar to JE, jumps when the Zero flag (ZF) is set.

JNZ (Jump if Not Zero): Similar to JNE, jumps when the Zero flag (ZF) is clear.

JC (Jump if Carry): Jumps when the Carry flag (CF) is set, indicating a carry occurred.

JNC (Jump if Not Carry): Jumps when the Carry flag (CF) is clear, indicating no carry occurred.

JO (Jump if Overflow): Jumps when the Overflow flag (OF) is set, indicating signed overflow.

JNO (Jump if No Overflow): Jumps when the Overflow flag (OF) is clear, indicating no signed overflow.

overflow.

JS (Jump if Sign): Jumps when the Sign flag (SF) is set, indicating a negative result.

JNS (Jump if Not Sign): Jumps when the Sign flag (SF) is clear, indicating a non-negative result.

Jumps Based on Equality Between Operands or the Value of (E)CX:

These jumps are used for comparing values for equality. The value of (E)CX can also be used for comparisons. Examples include:

JE (Jump if Equal): Jumps if two values are equal.

JNE (Jump if Not Equal): Jumps if two values are not equal.

JCXZ (Jump if CX is Zero): Jumps if the (E)CX register is zero.

Jumps Based on Comparisons of Unsigned Operands:

These jumps are used for comparing unsigned integers. They consider values without their sign. Examples include:

JA (Jump if Above): Jumps if the result is strictly greater (unsigned) than another value.

JAE (Jump if Above or Equal): Jumps if the result is greater than or equal (unsigned) to another value.

JB (Jump if Below): Jumps if the result is strictly less (unsigned) than another value.

JBE (Jump if Below or Equal): Jumps if the result is less than or equal (unsigned) to another value.

Jumps Based on Comparisons of Signed Operands:

Similar to the previous group, but used for comparing signed integers, considering their sign. Examples include:

JG (Jump if Greater): Jumps if the result is strictly greater (signed) than another value.

JGE (Jump if Greater or Equal): Jumps if the result is greater than or equal (signed) to another value.

JL (Jump if Less): Jumps if the result is strictly less (signed) than another value.

JLE (Jump if Less or Equal): Jumps if the result is less than or equal (signed) to another value.

Example 1:

```
mov edx, 0A523h ; Move 0A523h into the edx register
cmp edx, 0A523h ; Compare edx with 0A523h
jne L5          ; Jump if not equal to L5
je L1           ; Jump if equal to L1
```

In this example, cmp compares the value in edx with 0A523h. Since they are equal, the jne instruction is not taken, but the je instruction is taken, leading to a jump to L1.

Example 2:

```
mov bx, 1234h    ; Move 1234h into the bx register
sub bx, 1234h    ; Subtract 1234h from bx
jne L5           ; Jump if not equal to L5
je L1            ; Jump if equal to L1
```

In this example, sub subtracts 1234h from bx, resulting in zero. Therefore, the jne instruction is not taken, but the je instruction is taken, leading to a jump to L1.

Example 3:

```
mov cx, 0FFFFh   ; Move FFFFh into the cx register
inc cx           ; Increment cx by 1
jcxz L2          ; Jump if cx is zero to L2
```

Here, jcxz checks if the cx register is zero after the inc instruction. Since inc increments cx by 1, it becomes zero. Hence, the jcxz instruction is taken, leading to a jump to L2.

Example 4:

```
xor ecx, ecx     ; Set ecx to zero using XOR
jecxz L2         ; Jump if ecx is zero to L2
```

In this case, xor is used to set ecx to zero. Then, jecxz checks if ecx is zero. Since it is zero, the jecxz instruction is taken, leading to a jump to L2.

These examples demonstrate how conditional jump instructions like je, jne, jcxz, and jecxz work in assembly language to control program flow based on the result of comparisons and the state of registers.

Unsigned Comparisons (Table Below):

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)

These comparisons are used when you are dealing with unsigned values, which means that they don't have a sign (positive or negative).

Signed Comparisons (Table Below):

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* $\geq$ *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* $\leq$ *rightOp*)
JNG	Jump if not greater (same as JLE)

These comparisons are used when you are dealing with signed values, which have both positive and negative numbers.

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)
```

In this example, you have a signed comparison. jl jumps because -1 is indeed less than 0.

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)
```

Again, this is a signed comparison. jge jumps because +32 is indeed greater than or equal to -35.

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)
```

This is a signed comparison. jnl jumps because 0 is greater than or equal to 0.

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)
```

Here, jng jumps because 0 is indeed less than or equal to 0.

----

1. Conditional Jump Applications:

This section discusses how conditional jump instructions in assembly language can be used to test and manipulate status bits. It demonstrates examples of jumping to labels based on specific bit conditions in a status byte. This is a fundamental concept in assembly programming, allowing you to make decisions in your code based on the state of specific bits.

Conditional jump instructions in assembly language are fundamental for controlling the flow of your program based on specific conditions. They are often used to examine and manipulate individual bits in a byte or word of data. The status bits, such as the Zero Flag (ZF), Sign Flag (SF), and others, are set or cleared by various instructions and can be tested using conditional jumps.

In your provided example:

```
mov al, status        ; Load the status byte into AL
test al, 00100000b    ; Test bit 5 in AL
jnz DeviceOffline     ; Jump to DeviceOffline label if bit 5 is set
```

Here's a breakdown of what's happening:

mov al, status: This instruction loads the status byte into the AL register. The AL register is commonly used for working with 8-bit data.

test al, 00100000b: The test instruction performs a bitwise AND operation between AL and the binary value 00100000b, which sets all bits to zero except bit 5. This effectively tests if bit 5 in AL is set without modifying AL.

jnz DeviceOffline: The jnz (Jump if Not Zero) instruction checks the Zero Flag (ZF). If the Zero Flag is not set, it means that bit 5 in AL was not zero (i.e., bit 5 was set). In this case, the program jumps to the DeviceOffline label.

This example demonstrates how conditional jumps can be used to make decisions based on the state of specific bits in the AL register without changing the value of AL.

Remember that conditional jumps can be used to implement complex logic in assembly language, enabling you to create branching and decision-making in your code.

You can use other conditional jump instructions like je (Jump if Equal), jg (Jump if Greater), jl (Jump if Less), and more to handle various conditions.

2. Larger of Two Integers:

Here, the code snippet compares two unsigned integers (EAX and EBX) and moves the larger value to EDX. It uses conditional jumps to make the comparison and assignment. This is a basic example of conditional branching based on integer comparisons.

Certainly, let's delve deeper into the code snippet that compares two unsigned integers (EAX and

EBX) and moves the larger value to EDX. This is a great example of conditional branching based on integer comparisons in assembly language:

```
mov edx, eax  ; Assume EAX is larger
cmp eax, ebx  ; Compare EAX and EBX
jae L1        ; Jump to L1 if EAX is greater or equal
mov edx, ebx  ; Move EBX to EDX (EAX was not greater)
L1:
; EDX now contains the larger integer
```

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

mov edx, eax: Initially, the code assumes that EAX contains the larger integer. It copies the value in EAX to EDX. This is the default assignment.

cmp eax, ebx: The cmp instruction compares the values in EAX and EBX without changing them. It sets or clears the appropriate flags (e.g., Zero Flag, Carry Flag) based on the comparison result.

jae L1: The jae (Jump if Above or Equal) instruction checks the Carry Flag. If the Carry Flag is not set, it means that EAX is greater than EBX (unsigned comparison). In this case, the program jumps to the L1 label.

mov edx, ebx: If the jae condition is not met (EAX is not greater than EBX), the program proceeds to this line and moves the value in EBX to EDX. This effectively updates EDX with the larger integer, which is now in EBX.

L1:: This is the label where execution continues after the conditional jump. At this point, EDX holds the larger of the two integers, whether it was initially in EAX or EBX.

holds the larger of the two integers, whether it was initially in EAX or EBX.

This code snippet demonstrates how conditional branching is used to compare two integers and select the larger one, updating the EDX register accordingly.

It's important to note that the jae instruction is used for unsigned integer comparison.

If you were comparing signed integers, you would use different conditional jump instructions like jge (Jump if Greater or Equal) or jl (Jump if Less).

3. Smallest of Three Integers:

This section shows how to find the smallest of three unsigned 16-bit integers (V1, V2, and V3) and assigns the result to AX. It uses a series of conditional jumps to compare and select the smallest value.

Certainly, let's go through the code snippet that finds the smallest of three unsigned 16-bit integers (V1, V2, and V3) and assigns the result to the AX register. This code uses a series of conditional jumps to make the comparisons and selection:

```
.data
    V1 WORD ?
    V2 WORD ?
    V3 WORD ?

.code
    mov ax, V1       ; Assume V1 is the smallest
    cmp ax, V2       ; Compare AX and V2
    jbe L1           ; Jump to L1 if AX is less than or equal to V2
    mov ax, V2       ; Move V2 to AX (V1 is not the smallest)
    L1:
    cmp ax, V3       ; Compare AX and V3
    jbe L2           ; Jump to L2 if AX is less than or equal to V3
    mov ax, V3       ; Move V3 to AX (V2 or V1 is not the smallest)
    L2:
    ; AX now contains the smallest integer among V1, V2, and V3
```

Here's a step-by-step explanation of how this code works:

The code starts with the assumption that V1 is the smallest integer and loads the value of V1 into the AX register.

It then compares the value in AX (which now holds V1) with the value of V2 using the cmp instruction. The jbe (Jump if Below or Equal) instruction checks whether AX is less than or equal to V2.

If AX is less than or equal to V2 (the jbe condition is met), the program jumps to the label L1. In this case, V1 remains the smallest integer in AX.

If AX is not less than or equal to V2 (the jbe condition is not met), it means V2 is smaller, and the program updates AX with the value of V2.

The program then continues to compare the current value in AX (either V1 or V2) with V3 using the same cmp and jbe instructions. If AX is less than or equal to V3, it keeps the smallest value. If not, it updates AX with V3.

After these comparisons and conditional jumps, AX will contain the smallest of the three unsigned 16-bit integers (V1, V2, and V3).

This code demonstrates how to find the smallest integer among three values using conditional branching in assembly language.

4. Loop until Key Pressed:

In this part, a loop continuously runs until a standard alphanumeric key is pressed. It uses the ReadKey method from the Irvine32 library to check for a key press. If no key is present, the loop continues with a 10-millisecond delay between iterations. This is a practical example of waiting for user input in assembly code.

Certainly, the provided code is an example of creating a loop that continuously runs until a standard alphanumeric key is pressed.

It uses the ReadKey method from the Irvine32 library to check for a key press, and if no key is present, it continues with a 10-millisecond delay between iterations.

This is a practical way to wait for user input in assembly code. Let's break down the code:

```
.data
char BYTE ?

.code
L1:
    mov eax, 10           ; Create a 10 ms delay
    call Delay
    call ReadKey          ; Check for a key press
    jz L1                 ; If no key is pressed, repeat the loop
    mov char, AL          ; Save the character in the 'char' variable
```

Here's how this code works step by step:

mov eax, 10: This line sets up a delay by loading the value 10 into the EAX register. The Delay subroutine is then called to introduce a 10-millisecond pause. This delay is important to give the system some time to process other tasks and to avoid rapidly consuming CPU resources in a tight loop.

call ReadKey: The ReadKey subroutine is called to check for a key press. The result of this function is stored in the AL register. If a key is pressed, AL will contain the ASCII code of the key; otherwise, it will be 0.

jz L1: The jz instruction (Jump if Zero) checks whether the Zero Flag (ZF) is set. If AL is 0, it means no key was pressed, and the program jumps back to the L1 label, continuing the loop.

mov char, AL: If a key is pressed (i.e., AL is not 0), the ASCII code of the pressed key is stored in the char variable.

The loop continues until a key is pressed, and when a key is pressed, its ASCII code is stored in the char variable. This way, you can wait for and capture user input in your assembly program.

This is a practical way to handle user input in assembly code, especially when you want to wait for specific keypresses in a controlled manner.

The provided code is a simple example of how to search for the first nonzero value in an array of 16-bit integers.

```
; Scanning an Array (ArrayScan.asm)
; Scan an array for the first nonzero value.
INCLUDE Irvine32.inc
.data
    intArray SWORD 0,0,0,0,1,20,35,-12,66,4,0
    noneMsg BYTE "A non-zero value was not found",0
.code
    main PROC
        ; Initialize registers and variables
        mov esi, 0               ; Index for array traversal
        mov ecx, LENGTHOF intArray  ; Length of the array
        mov ebx, ADDR intArray    ; Address of the array
        mov al, 0                ; Clear AL register to store the result (found or not)
    searchLoop:
        cmp word ptr [ebx + esi * 2], 0  ; Compare the current element with zero
        jnz foundNonZero          ; Jump if not zero
        inc esi                  ; Increment index
        loop searchLoop           ; Continue loop until ecx is zero
        mov al, 1                ; Set AL to 1 if no nonzero value found
        jmp done
    foundNonZero:
        mov al, 0                ; Set AL to 0 if a nonzero value is found
    done:
        ; Display appropriate message based on AL value
        cmp al, 0
        je noNonZeroFound
        mov edx, OFFSET noneMsg
        call WriteString
        jmp endProgram
```

```
    noNonZeroFound:
        ; Display the first nonzero value found
        mov edx, [ebx + esi * 2]
        call WriteInt

    endProgram:
        call Crlf
        exit
    main ENDP

END main
```

Explanation:

We define the array intArray containing 16-bit integers and a message noneMsg.

In the code section, we initialize registers and variables. esi is used to keep track of the array index, ecx holds the length of the array, and ebx stores the address of intArray. al is initially set to 0, which will be used to determine if a nonzero value is found.

We use a loop labeled as searchLoop to traverse the array and compare each element to 0 using the cmp instruction. If the element is not zero (jnz instruction), we jump to the foundNonZero label.

If we reach the end of the loop without finding a nonzero value, we set al to 1 to indicate that no nonzero value was found.

We have separate code for displaying messages based on the value of al. If al is 0, we display the nonzero value found; if it's 1, we display the "non-zero value not found" message.

The program then ends by calling Crlf and exiting.

You can uncomment different test data configurations in the .data section to test the program with various arrays.

===

Encryption Program Overview

===

This assembly program demonstrates a simple symmetric encryption technique using the XOR operation. The program follows these steps:

- **User Input:** The user enters a plain text message.

- **Encryption:** The program encrypts the plain text by XORing each character with a single character key and displays the cipher text.

- **Decryption:** It then decrypts the cipher text using the same key and displays the original plain text.

```
INCLUDE Irvine32.inc

KEY = 239                   ; The encryption key (single character)
BUFMAX = 128                ; Maximum buffer size for input

.data
    sPrompt BYTE "Enter the plain text:",0
    sEncrypt BYTE "Cipher text: ",0
    sDecrypt BYTE "Decrypted: ",0
    buffer BYTE BUFMAX+1 DUP(0)
    bufSize DWORD ?

.code
    main PROC
```

The program starts by including the Irvine32 library for input and output functions.

KEY is defined as the encryption key, set to 239.

BUFMAX defines the maximum buffer size for input.

```
    call InputTheString
    call TranslateBuffer
    mov edx, OFFSET sEncrypt
    call DisplayMessage
    call TranslateBuffer
    mov edx, OFFSET sDecrypt
    call DisplayMessage
    exit
main ENDP
```

The main procedure calls InputTheString to get user input, TranslateBuffer for encryption, and DisplayMessage to show the cipher text. It repeats this process for decryption.

```
InputTheString PROC
    pushad
    mov edx, OFFSET sPrompt
    call WriteString
    mov ecx, BUFMAX
    mov edx, OFFSET buffer
    call ReadString
    mov bufSize, eax
    call Crlf
    popad
    ret
InputTheString ENDP
```

InputTheString procedure prompts the user for input, reads it into the buffer, and stores its length in bufSize.

```
DisplayMessage PROC
    pushad
    call WriteString
    mov edx, OFFSET buffer
    call WriteString
    call Crlf
    call Crlf
    popad
    ret
DisplayMessage ENDP
```

DisplayMessage procedure displays a given message (in EDX) followed by the contents of the buffer and two line breaks.

```
TranslateBuffer PROC
    pushad
    mov ecx, bufSize
    mov esi, 0 L1:
    xor buffer[esi], KEY
    inc esi
    loop L1
    popad
    ret
TranslateBuffer ENDP
```

TranslateBuffer procedure translates the string in the buffer by XORing each byte with the encryption key (KEY).

Final Note:

The program uses a single-character key (which is not secure in real-world scenarios). The exercises suggest using a multi-character encryption key for stronger security. This program is a basic example to understand the concept of XOR-based encryption in assembly language. In practice, encryption algorithms like AES or RSA are used for secure data protection.

Which jump instructions follow unsigned integer comparisons?

Jump instructions following unsigned integer comparisons typically include JA (Jump if Above), JAE (Jump if Above or Equal), JB (Jump if Below), and JBE (Jump if Below or Equal).

Which jump instructions follow signed integer comparisons?

Jump instructions following signed integer comparisons usually include JG (Jump if Greater), JGE (Jump if Greater or Equal), JL (Jump if Less), and JLE (Jump if Less or Equal).

Which conditional jump instruction is equivalent to JNAE?

JNAE stands for "Jump if Not Above or Equal," and its equivalent for signed comparisons is JB which stands for "Jump if Below."

Which conditional jump instruction is equivalent to the JNA instruction?

The JNA instruction stands for "Jump if Not Above," and its equivalent for signed comparisons is JL, which stands for "Jump if Less."

Which conditional jump instruction is equivalent to the JNGE instruction?

JNGE stands for "Jump if Not Greater or Equal," and its equivalent for signed comparisons is JG, which stands for "Jump if Greater."

(Yes/No): Will the following code jump to the label named Target?

```
mov ax, 8109h
cmp ax, 26h
jg Target
```

Yes, the code will jump to the label named "Target" if the value in the ax register (8109h) is greater than the immediate value 26h. This is because jg stands for "Jump if Greater."