Contents

AND OPERATOR

Boolean Instructions in Assembly Language

Boolean instructions let us perform logical operations on bits or bytes. They're super handy for manipulating data, making decisions, and steering the flow of a program.

✨ AND

The **AND** instruction compares two operands bit by bit.

- If *both* bits are 1, the result is 1.
- Otherwise, the result is 0. 👉 The outcome is stored in the destination operand.

✨ OR

The **OR** instruction also works bit by bit.

- If *either* bit is 1, the result is 1.
- If both are 0, the result is 0. 👉 The result goes into the destination operand.

✨ XOR (Exclusive OR)

The **XOR** instruction checks two operands bit by bit.

- If *only one* of the bits is 1, the result is 1.
- If both are the same (both 0 or both 1), the result is 0. 👉 The result is stored in the destination operand.

✨ NOT

The **NOT** instruction flips every bit in a single operand.

- 1 becomes 0
- 0 becomes 1 👉 The inverted result is stored in the destination operand.

✨ TEST

The **TEST** instruction performs a bit-by-bit **AND** operation on two operands— but here's the twist: it doesn't save the result!

Instead, it updates the CPU flags based on the outcome. 👉 This makes it perfect for checking values in registers or memory before making a decision.

Sr.No.	Instruction	Format
1	AND	AND operand1, operand2
2	OR	OR operand1, operand2
3	XOR	XOR operand1, operand2
4	TEST	TEST operand1, operand2
5	NOT	NOT operand1

Here are some examples of how to use the Boolean instructions in assembly language:

```
; AND two registers
mov eax, 10h
mov ebx, 5h
and eax, ebx ; eax = 4h

; OR two registers
mov eax, 10h
mov ebx, 5h
or eax, ebx ; eax = 15h

; XOR two registers
mov eax, 10h
mov ebx, 5h
xor eax, ebx ; eax = 11h

; NOT a register
mov eax, 10h
not eax ; eax = 11111110h

; TEST two registers
mov eax, 10h
mov ebx, 5h
test eax, ebx ; CF flag is set to 0

; TEST a register against a value
mov eax, 10h
test eax, 0Fh ; CF flag is set to 1
```

CPU Flags

Flags are little indicators inside the CPU that light up based on the result of an operation. They help the processor make decisions and control program flow.

✨ Zero Flag (ZF)

- **Set when**: The result of an operation is 0.
- **Use case**: Perfect for checking equality or non-equality.
- **Example**: Skip instructions if the result is zero.

✨ Carry Flag (CF)

- **Set when**: An operation produces a carry out of the highest bit.
- **Use case**: Common in addition and subtraction.
- **Example**: Signals overflow in addition if set.

✨ Sign Flag (SF)

- **Set when**: The most significant bit (MSB) of the result is 1.
- **Use case**: Tells whether the result is positive or negative.
- **Example**: Set if the result is negative.

✨ Overflow Flag (OF)

- **Set when**: The result goes outside the signed number range.
- **Use case**: Detects arithmetic errors.
- **Example**: Adding two positive numbers gives a negative result.

✨ Parity Flag (PF)

- **Set when**: The destination operand has an even number of 1 bits.
- **Use case**: Helpful for error checking.
- **Example**: Can signal data corruption if not set when reading from memory.

AND Instruction

The **AND** instruction compares two operands bit by bit and stores the result in the destination operand.

Operation: Each pair of matching bits is checked.

- If both bits are 1, the result is 1.
- Otherwise, the result is 0.

```
30 AND reg, reg
31 AND reg, mem
32 AND reg, imm
33 AND mem, reg
34 AND mem, imm
```

```
AND destination, source
```

Operand Sizes: The operands can be 8, 16, 32, or 64 bits, and they must be the same size.

If both bits equal 1, the result bit is 1; otherwise, it is 0.

Example: x AND y, where x and y are bits.

x	y	x ∧ y
0	0	0
0	1	0
1	0	0
1	1	1

Bit Masking: Think of the **AND** instruction as a 'filter.' If you need to reset a hardware device by turning off specific bits (like bits 0 and 3) while leaving everything else exactly as it is, you can use a mask to 'wipe' those spots clean.

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

🌟 AND Instruction & Flags

- The **AND instruction** always clears the **Overflow** and **Carry** flags.
- It updates the **Sign**, **Zero**, and **Parity** flags depending on the result stored in the destination operand.

🎭 Masking with AND

The AND instruction isn't just about logic—it's also a handy tool for **masking**. Masking means you can clear specific bits in an operand while leaving the rest untouched.

- To clear a bit, you AND the operand with a mask that has a **0** in the position you want to wipe out.
- Example: To clear the **least significant bit (LSB)** of the **AL register**, use this mask: **0b11111110**

```
; Mask out the least significant bit of the AL register.

mov al, 0b00111011b ; AL = 00111011b
and al, 0b11111110b ; AL = 00111010b

; The least significant bit of the AL register is now cleared.
```

You can use the **same masking technique** to clear *any* bit in an operand, no matter its position.

To mask out bit **n**, simply AND the operand with a mask that has a **0** in that bit position.

🛠 Why Masking Matters

Masking is a powerful tool in assembly language programming. It lets you:

- 🔍 **Isolate** a specific bit (or group of bits).
- 🧹 **Clear or set** chosen bits without touching the rest.
- ✅ **Check** whether certain bits are set or cleared.
- 🤝 Perform logical **AND operations** across multiple operands.

✨ Example: Lowercase → Uppercase

Here's a neat trick: you can convert a character from **lowercase to uppercase** by clearing **bit 5** of the character.

- Bit 5 is the one that decides whether a character is lowercase or uppercase.
- By masking it out, you flip the character into its uppercase form.

🧩 Code Breakdown

```
.data
    array BYTE 50 DUP(?)

.code
    mov ecx, LENGTHOF array
    mov esi, OFFSET array
    L1:
    and byte ptr [esi], 11011111b
    inc esi
    loop L1
```

🔠 How the Code Works (Step by Step)

1. **Load the character** into a register.
2. **AND it** with a mask that clears bit 5.
3. The result is the **uppercase version** of the original character.

🏴 Sections of the Program

- .data **section** → Declares an array of 50 bytes. Each byte can hold a single character.
- .code **section** → Contains the actual instructions that perform the conversion to uppercase.

⚙️ Key Instructions Explained

- mov ecx, LENGTHOF array → Loads the length of the array into register ecx. This tells the program how many characters to process.
- mov esi, OFFSET array → Loads the address of the first element in the array into register esi. This sets up a pointer to the start of the data.
- L1: **label** → Marks the beginning of a loop. The loop will run once for each character in the array.
- and byte ptr [esi], 11011111b → Performs an AND operation on the byte at the address stored in esi with the mask 11011111b. This clears **bit 5**, which flips the character from lowercase to uppercase.
- inc esi → Increments esi by 1, moving the pointer to the next character in the array.
- loop L1 → Jumps back to the start of the loop until all characters have been processed.

🎉 End Result

After the loop finishes, every character in the array has been converted to **uppercase**. The program efficiently walks through the array, one character at a time, applying the bit-clearing trick to achieve the transformation.

BIT-MAPPED SETS

- **Representation**: A set is represented as a **bit vector**. Each bit corresponds to an element in the universal set.
- **Efficiency**: Instead of complex data structures, you manipulate sets directly with bitwise instructions.
- **Applications**: Common in **systems programming**, hardware control, and memory management where speed and compactness matter.

🔍 Checking Set Membership

To see if an element is in a set, AND the set with a mask that isolates the bit for that element.

```
.data
SetX DWORD 10000000h  ; SetX represented as a 32-bit value

.code
main PROC
    mov eax, SetX     ; Load SetX into EAX
    and eax, 00000010b ; Check if element[4] is a member
    cmp eax, 0         ; Compare the result with 0
    jz not_a_member    ; Jump if Zero flag is set (not a member)
    ; If Zero flag is not set, element[4] is a member of SetX
    ; Your code here for member case
    jmp done

not_a_member:
    ; Your code here for not a member case

done:
    ; Exit your program
    invoke ExitProcess, 0

main ENDP
```

OR

```
mov eax, SetX         ; Load the set into EAX
and eax, 00010000b    ; Mask to check element[4]
jz NotMember          ; If Zero flag is set, element not in set
; Otherwise, element is a member
```

🔄 Set Complement

Flip all bits with the NOT instruction to get the complement.

```
.data
SetX DWORD 10000000h  ; SetX represented as a 32-bit value

.code
main PROC
    mov eax, SetX     ; Load SetX into EAX
    not eax           ; Complement SetX
    ; EAX now contains the complement of SetX
    ; Your code here to work with the complemented set
    ; Exit your program
    invoke ExitProcess, 0

main ENDP
```

🤝 Set Intersection

Use AND to find elements common to both sets.

```
.data
SetX DWORD 10000000000000000000000000000111h
SetY DWORD 10000010101000000000011101100011h

.code
main PROC
    mov eax, SetX     ; Load SetX into EAX
    and eax, SetY     ; Calculate the intersection of SetX and SetY
    ; EAX now contains the intersection
    ; Your code here to work with the intersection
    ; Exit your program
    invoke ExitProcess, 0

main ENDP
```

🌐 Set Union

Use OR to combine elements from both sets.

Example of SetX and SetY:

```
.data
SetX DWORD 10000000000000000000000000000111h
SetY DWORD 10000010101000000000011101100011h

.code
main PROC
    mov eax, SetX      ; Load SetX into EAX
    or eax, SetY       ; Calculate the union of SetX and SetY
    ; EAX now contains the union
    ; Your code here to work with the union
    ; Exit your program
    invoke ExitProcess, 0

main ENDP
```

🎉 Why This Matters

Bit-mapped sets let you:

- Quickly check membership with a single instruction.
- Perform **set operations** (union, intersection, complement) at machine speed.
- Save memory by representing sets compactly as bit vectors.

OR OPERATION

The **OR instruction** performs a **bitwise OR** between each pair of matching bits in two operands.

The result is stored in the **destination operand**.

```
51 OR reg, reg
52 OR reg, mem
53 OR reg, imm
54 OR mem, reg
55 OR mem, imm
```

Combinations above: Same as the AND instruction.

📏 Operand Sizes

- Operands can be **8, 16, 32, or 64 bits**.
- Both operands must be the **same size**.

✨ Truth Table:

For each matching bit in the two operands:

Example: x OR y, where x and y are bits.

x	y	x ∨ y
0	0	0
0	1	1
1	0	1
1	1	1

👉 The output bit is **1** if *at least one* of the input bits is 1.

🔧 Setting Bits with OR

Think of the **OR** instruction as a surgical strike for your data. It's the go-to move in embedded programming when you need to flip a specific "switch" in a register without disturbing any of the other settings.

To turn on bit 2 in the AL register while leaving the rest of the configuration untouched, you'd use:

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

or

```
OR AL, 1 << 2
```

The OR instruction performs a bitwise OR operation on its two operands.

The **<< operator** shifts the number on its left by the number of bits specified by the number on its right.

In this case, the number 1 is shifted left by 2 bits, which results in the number 4.

The OR instruction then ORs the AL register with the number 4, which sets the bit in position 2 of the AL register.

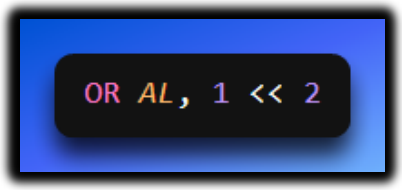
Here is an example of how to use the code above:

```
61 ; Set the bit in position 2 of the control byte in the AL register.
62
63 mov al, 0b00111010 ; AL = 00111010
64 OR AL, 1 << 2 ; AL = 00111110
65
66 ; The bit in position 2 of the AL register is now set.
```

You can use the **OR instruction** to set any bit in an operand, no matter its position.

- To set bit **n**, OR the operand with the value 1 shifted left by **n** positions.
- This creates a **bitmask** where only the target bit is 1, and all others are 0.

✨ Example: Setting Bit 2 in AL



Step-by-Step Breakdown:

Start with 00000001 (binary for 1).

Shift it left by 2 → 00000100.

This creates a mask with **bit 2** set to 1.

Perform OR AL, 00000100b.

- The OR operation sets **bit 2** in AL to 1.
- All other bits remain unchanged.

Result: After executing this instruction:

- **Bit 2** in the AL register is guaranteed to be 1.
- Every other bit in AL stays exactly as it was before.

Flags and the OR Instruction

- The **OR instruction** always **clears** the **Carry** and **Overflow** flags.
- It **updates** the **Sign**, **Zero**, and **Parity** flags based on the result stored in the destination operand.

Why It Matters

Bitwise instructions like **OR** are essential in assembly programming because they let you:

- 🔧 Perform precise **bit manipulation**.
- 🎯 Control or check **flags** for conditional logic.
- ⚡ Work efficiently with low-level data structures.

Typical Uses

- Setting specific bits without disturbing others.
- Combining values (like set union in bit-mapped sets).
- Flag setting/clearing in system-level code.

```
SetX:
10000000000000000000000000000111

SetY:
10000010101000000000011101100011

Union (SetX OR SetY):
10000010101000000000011101100111
```

It's clear that the OR operation combines the two sets by preserving any bits that are set in either SetX or SetY.

In binary representation:

- SetX has bits set at positions 0, 1, and 31.
- SetY has bits set at positions 0, 5, 9, 14, 18, 23, 26, 30, and 31.

When you perform a bitwise OR between SetX and SetY, the resulting union has bits set at all the positions where at least one of SetX or SetY had a bit set. In this case, the union contains bits set at positions 0, 1, 5, 9, 14, 18, 23, 26, 30, and 31.

This operation can be visualized as a union operation in set theory, where you're combining the elements of two sets while eliminating duplicates.

XOR OPERATION / INSTRUCTION

✨ XOR (Exclusive OR) Instruction

- The **XOR instruction** performs a **boolean exclusive-OR** operation between corresponding bits in two operands.
- The result is stored in the **destination operand**.

📏 XOR Rules

- If both bits are the **same** (both 0 or both 1), the result is 0.
- If the bits are **different** (0 and 1), the result is 1.
- A bit XORed with 0 → **retains its value**.
- A bit XORed with 1 → **toggles (complements)** its value.

🔄 Reversibility

XOR is **reversible**:

- Applying XOR twice with the same operand restores the **original value**.
- This property makes XOR useful in tasks like **encryption/decryption** and **bit manipulation tricks**.

🧮 Truth Table for XOR (x ⊕ y)

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

The XOR instruction performs a bitwise exclusive OR operation on its two operands.

```
xor destination_operand, source_operand
```

Flags and the XOR Instruction

- The **XOR instruction** always **clears** the **Overflow** and **Carry** flags.
- It **updates** the **Sign**, **Zero**, and **Parity** flags based on the result stored in the destination operand.

🔍 Parity Checking

- **Parity** is a way to check whether a binary number has an **even** or **odd** count of 1 bits.
  - Even number of 1s → **even parity**.
  - Odd number of 1s → **odd parity**.
- In **x86 processors**, the **Parity Flag (PF)** is set if the **lowest byte (8 bits)** of the result has **even parity**.
- If the lowest byte has **odd parity**, the PF is **cleared**.

✨ Example: Checking Parity Without Changing a Value

Here's how you can check parity in assembly without altering the actual byte:

```
mov al, byteValue    ; Load the byte into AL
test al, al          ; Perform a bitwise AND of AL with itself
jpo OddParity        ; Jump if Parity Flag is cleared (odd parity)
; If PF is set, the byte has even parity
```

🧩 Breakdown

- mov al, byteValue → Loads the byte into the AL register.
- test al, al → Performs an AND of AL with itself. This doesn't change AL, but it updates the flags (including PF).
- jpo OddParity → Jumps if the Parity Flag is **not set** (odd parity).
- If the jump doesn't occur, it means the byte has **even parity**.

```
199 mov al, 10110101b ; 5 bits = odd parity
200 xor al, 0          ; Parity flag clear (odd)
201
202 mov al, 11001100b ; 4 bits = even parity
203 xor al, 0          ; Parity flag set (even)
```

The XOR instruction can be used to toggle (invert) bits, check the parity of a number, and perform other bitwise operations. Here are some examples of how to use the XOR instruction:

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

To calculate parity for 32-bit values, you can XOR all the bytes together, like this:

```
212 B0 XOR B1 XOR B2 XOR B3
```

NOT OPERATION

The **NOT instruction** is used to invert or toggle all the bits in an operand. This operation is also known as taking the one's complement of the operand. Here's how it works:

NOT reg: This form of the NOT instruction operates on a register. It inverts all the bits in the specified register.

```
215 mov al, 11110000b ; AL = 11110000b (F0h in hexadecimal)
216 not al            ; AL = 00001111b (0Fh in hexadecimal)
```

NOT mem: This form of the NOT instruction operates on a memory location. It inverts all the bits in the value stored at that memory location.

```
218 mov byte ptr [ebx], 10101010b ; Store 10101010b at the memory location pointed to by EBX
219 not byte ptr [ebx]            ; Invert the bits at that memory location
```

In both examples, the NOT instruction flips all the bits. In the first example, it's applied to the AL register, and in the second example, it's applied to a byte stored in memory via the EBX register.

Flags: The NOT instruction does not affect any of the CPU flags. It simply performs the bitwise inversion without changing the status flags like Zero Flag, Sign Flag, etc.

TEST OPERATION

The **TEST instruction** is a handy tool for performing **bitwise logical checks** without changing the actual contents of the destination operand.

🧪 What TEST Does

- Performs a **bitwise AND** between two operands.
- Updates the **Sign (S)**, **Zero (Z)**, and **Parity (P)** flags based on the result.
- Unlike the **AND instruction**, it does **not modify** the destination operand.

👉 This makes TEST perfect for **checking whether specific bits are set** while leaving the original data untouched.

📊 Operand Combinations

The TEST instruction supports the same operand combinations as the AND instruction:

- Register ↔ Register
- Register ↔ Memory
- Register ↔ Immediate (constant)
- Memory ↔ Immediate

✨ Why It's Useful

- Efficient way to **check bit status**.
- Commonly used in **conditional branching** (e.g., jump if zero, jump if sign).
- Preserves the original data while still giving you flag information to act on.

Example: Testing Multiple Bits:

In the example you provided, the goal is to determine whether bit 0 or bit 3 is set in the AL register. The following instruction accomplishes this:

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

Think of the **TEST** instruction as a "silent inspector." It performs a bitwise AND between two values to see what's happening under the hood, but it doesn't actually change anything in your registers. It's the perfect way to "peek" at a status bit before deciding what your code should do next.

If you want to find out if either **bit 0** or **bit 3** is active in the AL register, you'd use a mask like this:

Reading the Flags (The "Secret Sauce")

Since TEST doesn't save the result, you have to look at the **CPU flags** to see what happened. The most important one here is the **Zero Flag (ZF)**:

- **If ZF = 1 (Zero):** Neither bit was set. The result was a total blank.
- **If ZF = 0 (Not Zero):** Success! At least one of those bits (0 or 3) was a 1.

What happens to the other flags?

While the TEST instruction is busy checking your bits, it also does a little housekeeping on the status register:

- **ZF, SF, and PF:** These are updated based on the result (just like a standard AND).
- **CF and OF:** These are always cleared (set to 0) because bitwise logic doesn't involve carries or overflows.

The TEST instruction is your go-to for **conditional branching**.

You use it to check a status, and then immediately follow it with a jump instruction (like JZ for "Jump if Zero" or JNZ for "Jump if Not Zero") to steer your program's logic.

Example of using a bit mask with the TEST instruction

(The value 00001001 in this example is called a bit mask.)

```
Input value:    0 0 1 0 0 1 0 1
Test value:     0 0 0 0 1 0 0 1
Result:         0 0 0 0 0 0 0 1
Zero Flag (ZF) = 0 (Not set)


Input value:    0 0 1 0 0 1 0 0
Test value:     0 0 0 0 1 0 0 1
Result:         0 0 0 0 0 0 0 0
Zero Flag (ZF) = 1 (Set)
```

The Big Correction: Equality vs. Bits

The text you have says TEST checks if two values are **equal**. That's not quite right.

- **CMP (Compare):** Subtracts values to see if they are equal.
- **TEST:** Performs an **AND** to see if specific bits are "on."

If you TEST AL, 09h, you aren't checking if AL is 9. You are checking if **Bit 0** or **Bit 3** (the bits that make up 9) have any life in them.

How the Flags Actually Work

Think of the **Zero Flag (ZF)** as a "Nothing Found" flag.

- **ZF = 1 (True/Set):** The result was zero. This means **none** of the bits you were looking for were turned on. It's like searching a room and finding nothing.
- **ZF = 0 (False/Clear):** The result was *not* zero. This means **at least one** of the bits you were testing is active.

The "Always" Flags

No matter what bits you are checking, the CPU always does a quick bit of house-cleaning:

- **Carry (CF) & Overflow (OF):** These are **always forced to 0**. Why? Because bitwise logic doesn't "carry over" or "overflow" like addition does.
- **Sign (SF) & Parity (PF):** These just report on the result. If the top bit of the result is 1, SF turns on. If the number of 1s is even, PF turns on.

Using it for Branching

In your example, the TEST is usually followed by a jump:

- **JZ (Jump if Zero):** Take this path if the bits we checked were **all zeros**.
- **JNZ (Jump if Not Zero):** Take this path if **at least one** of those bits was a 1.

Wait, a quick tip: If you actually wanted to see if AL is exactly equal to 9, you should use CMP AL, 9. Using TEST for equality only works in very specific, rare cases!

Here is a simple example of how to use the test operation in assembly language:

```
263 ; Test the value of the EAX register against the value 10.
264 ; If the two values are equal, then jump to the label "result".
265 test eax, 10
266 je result
267
268 ; The two values are not equal, so continue executing the next instruction.
```

CMP INSTRUCTION

The **CMP instruction** is used to compare two operands (integers, character codes, etc.) by performing an **implied subtraction**: DESTINATION – SOURCE.

- **No operands are modified**.
- Only the **CPU flags** are updated based on the result.
- This makes CMP essential for **conditional branching** and building logic structures similar to if statements in high-level languages.

The **CMP instruction** performs an implied subtraction of the source operand from the destination operand. However, the actual subtraction is not performed. Instead, the status flags are set according to the result of the subtraction.

Here's a breakdown of how the CMP instruction affects flags based on the comparison results:

CMP Results	ZF	CF	Unsigned Operands	Signed Operands
Destination < source	0	1	Carry flag is set	Sign flag and Carry flag are set
Destination > source	0	0	Carry flag is not set	Sign flag and Carry flag are not set
Destination = source	1	0	Carry flag is not set	Sign flag is not set

Unsigned Operands. When comparing two unsigned operands:

If the destination is less than the source, the Zero Flag (ZF) is set to 0, and the Carry Flag (CF) is set to 1. If the destination is greater than the source, ZF is set to 0, and CF is set to 0. If the destination equals the source, ZF is set to 1, and CF is set to 0.

Signed Operands. When comparing two signed operands:

If the destination is less than the source, the Sign Flag (SF) is not equal to the Overflow Flag (OF). If the destination is greater than the source, SF is equal to OF. If the destination equals the source, the Zero Flag (ZF) is set to 1.

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

In this example, when AX (with a value of 5) is compared to 10, the CMP instruction sets the Zero Flag (ZF) to 0 because 5 is not equal to 10. The Carry Flag (CF) is set to 1 because subtracting 10 from 5 would require a borrow. Example 2:

```
276 mov ax, 1000
277 mov cx, 1000
278 cmp cx, ax
279 ; ZF = 1 and CF = 0
```

Here, when AX and CX both contain 1000, the CMP instruction sets the Zero Flag (ZF) to 1 because subtracting one 1000 from the other results in zero. The Carry Flag (CF) is set to 0 because no borrow is required. Example 3:

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

In this case, when SI (with a value of 105) is compared to 0, the CMP instruction sets both the Zero Flag (ZF) and the Carry Flag (CF) to 0 because subtracting 0 from 105 generates a positive, nonzero value.

CMP, when used in conjunction with conditional jump instructions, allows you to create conditional logic structures, akin to high-level programming languages' IF statements, in assembly language. It's a powerful tool for controlling the flow of your programs based on comparisons between values in registers or memory locations.

SETTING AND CLEARING FLAGS

Setting the Zero Flag

To set the Zero flag, you can use the TEST or AND instruction. In the code:

```
287 test al, 0 ; set Zero flag
```

This instruction tests the value in the al register against 0. If the result is zero, the Zero flag is set.

Clearing the Zero Flag

To clear the Zero flag, you can use the OR instruction with 1:

```
290 or al, 1 ; clear Zero flag
```

This instruction logically ORs the al register with 1, ensuring that the Zero flag is cleared.

Setting the Sign Flag

To set the Sign flag, you can use the OR instruction with the highest bit of an operand (bit 7 in the al register) set to 1:

```
296 or al, 80h ; set Sign flag
```

This operation sets the highest bit of al to 1, which sets the Sign flag.

Clearing the Sign Flag

To clear the Sign flag, you can use the AND instruction with the highest bit (bit 7) set to 0:

```
299 and al, 7Fh ; clear Sign flag
```

This operation clears the highest bit of al, ensuring that the Sign flag is cleared.

Setting the Carry Flag

To set the Carry flag, you can use the STC (Set Carry) instruction:

```
306 stc ; set Carry flag
```

This instruction sets the Carry flag, indicating a carry condition.

Clearing the Carry Flag

To clear the Carry flag, you can use the CLC (Clear Carry) instruction:

```
310 clc ; clear Carry flag
```

This instruction clears the Carry flag, indicating no carry condition.

Setting the Overflow Flag

To set the Overflow flag, you can add two positive values that produce a negative sum. This condition naturally sets the Overflow flag.

Clearing the Overflow Flag

To clear the Overflow flag, you can use the OR instruction with an operand of 0:

```
313 or eax, 0 ; clear Overflow flag
```

This operation performs a logical OR with 0, ensuring that the Overflow flag is cleared.

Our code also mentions the relationship between flags (SF, OF, ZF) and the results of comparisons and arithmetic operations.

It's crucial to understand these flag behaviors in ASM.

BOOLEANS AND 64-BIT MODE

In 64-bit mode, instructions work similarly to how they do in 32-bit mode, but with some differences due to the larger register size.

Operand Size: When you operate on 64-bit registers or memory operands with a source operand that's smaller than 32 bits, all bits in the destination operand are affected eg.

```
316 mov rax, allones     ; RAX = FFFFFFFFFFFFFFFF
317 and rax, 80h         ; RAX = 0000000000000080
```

Here, the and operation affects all 64 bits of RAX.

When you use a **32-bit constant or register** as the source operand, only the **lower 32 bits** of the destination operand are modified.

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

Only the **lower 32 bits of RAX** are affected. The upper 32 bits are cleared to zero.

📂 Memory Operands

- The same rules apply when the **destination operand is in memory**.
- If you perform a 32-bit operation on a memory location, only the 32-bit portion is modified.

⚠️ Special Handling

In **64-bit mode**, 32-bit operands behave differently compared to other operand sizes:

- **8-bit and 16-bit operations** → Only affect the specified portion, leaving the rest unchanged.
- **32-bit operations** → Affect the lower 32 bits and **zero out the upper 32 bits** of the 64-bit register.
- **64-bit operations** → Affect the entire register.

🎯 Why This Matters

- Misunderstanding these distinctions can lead to **unexpected results** in 64-bit assembly programming.
- Always be mindful of operand size when working with registers or memory in 64-bit mode.

Question: How can you clear the high 8 bits of AX without changing the low 8 bits using a single 16-bit operand instruction?

Answer: You can clear the high 8 bits of AX by using the AND instruction with the 16-bit mask 00FFh. The instruction would look like and ax, 00FFh.

```
and ax, 00FFh
```

Question: How can you set the high 8 bits of AX without changing the low 8 bits using a single 16-bit operand instruction?

Answer: You can set the high 8 bits of AX by using the OR instruction with a 16-bit value.

```
or ax, FF00h
```

Question: What instruction can you use to reverse all the bits in EAX with a single instruction?

Answer: To reverse all the bits in EAX, you can use the XOR instruction with a mask where all bits are set to FFFFFFFFh. The instruction would be xor eax, FFFFFFFFh.

```
xor eax, FFFFFFFFh
```

Question: How can you set the Zero flag if the 32-bit value in EAX is even and clear the Zero flag if EAX is odd?

Answer: You can set the Zero flag if the 32-bit value in EAX is even and clear the Zero flag if EAX is odd using the TEST instruction and conditional jumps. Here's an example:

```
test eax, 1      ; Test if the least significant bit is set
jz even_label    ; Jump if Zero flag is set (EAX is even)
; Code for odd case here
even_label:
; Code for even case here
```

Question: How can you convert an uppercase character in AL to lowercase using a single instruction, but without modifying AL if it's already lowercase?

Answer: To convert an uppercase character in AL to lowercase without modifying it if it's already lowercase, you can use conditional instructions like this:

```
343 cmp al, 'A'       ; Compare AL with 'A'
344 jl not_uppercase ; Jump if AL is less than 'A' (not uppercase)
345 cmp al, 'Z'       ; Compare AL with 'Z'
346 jg not_uppercase ; Jump if AL is greater than 'Z' (not uppercase)
347 add al, 32        ; Convert uppercase to lowercase ('A'-'a' = 32)
348 not_uppercase:
349 ; Continue with your code here
```

This code first checks if AL is between 'A' and 'Z' (inclusive) using CMP and conditional jumps (JL and JG). If it's within that range, it adds 32 to AL, converting the uppercase letter to lowercase.

CONDITIONAL JUMPS

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:

```
08 cmp eax, 0     ; Compare the value in EAX with 0
09 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.

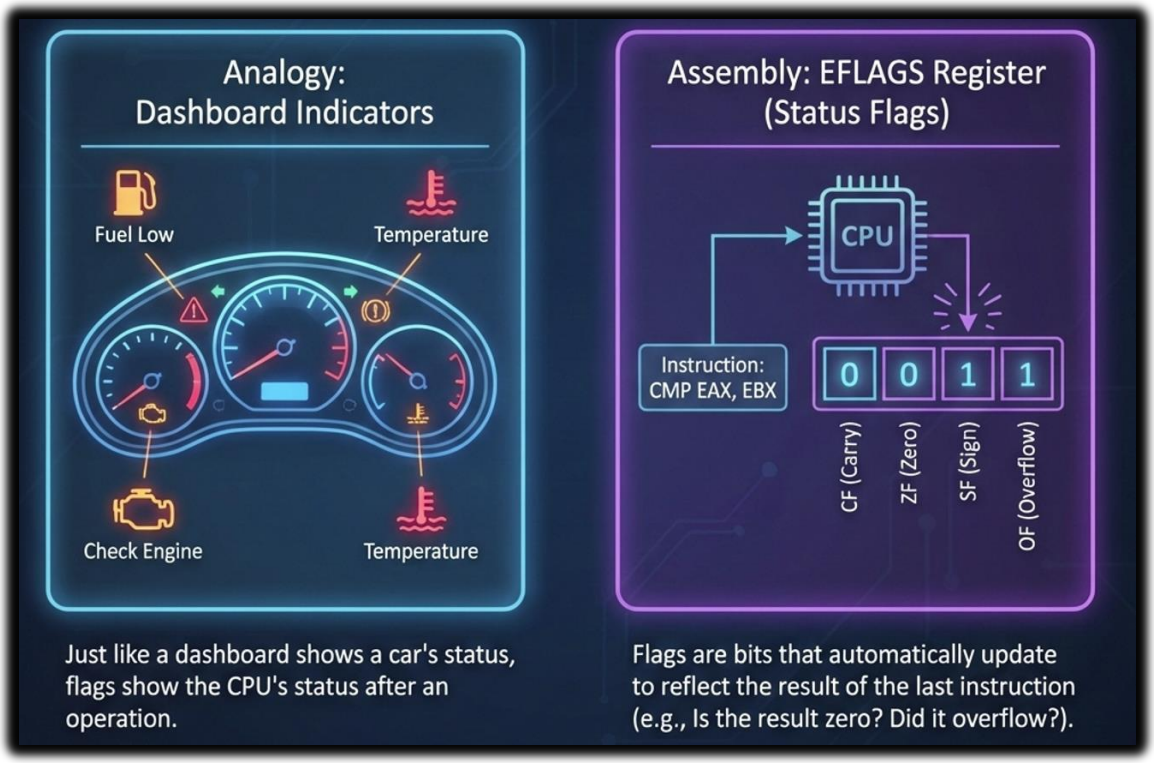
So, in simple terms, this code checks if the value in the EAX register is zero. If it is, it jumps to L1. If not, it continues executing the code below the jz instruction.

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

- **and dl, 10110000b**: This line performs a *bitwise AND* between the value stored in the DL register and the binary value 10110000.
- The result affects the Zero Flag (ZF). If the outcome of the AND operation is zero, the ZF is cleared. If the result is anything other than zero, the ZF gets set.
- **jnz L2**: This is a *conditional jump* instruction, just like the one we saw earlier.
- It checks the Zero Flag (ZF), and if ZF is not set (meaning the result of the AND operation wasn't zero), it jumps to the label L2.
- If ZF *is* set (meaning the AND result was zero), it won't jump and will continue with the next instructions.

So, if the bitwise AND result isn't zero, the program jumps to L2. If the result is zero, it keeps going to the next part of the code.

These jumps help guide the flow of the program, much like if-else statements in higher-level languages!



Let's dive into conditional jumps in assembly

Conditional jump instructions are super useful because they let you change the flow of your program based on certain flags set by previous instructions, like comparisons or arithmetic operations.

These flags are like indicators that tell the program how things are going. Some common jump instructions:

I. JE (Jump if Equal):

This instruction will jump to a specific label only if the Zero flag (ZF) is set. The Zero flag gets set when two values are equal, so this is the instruction you'd use when you want to take action if the comparison shows equality.

For example, if you compare two values and they turn out to be equal, JE will trigger a jump to a label you've set in the code, otherwise, it keeps going.

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

II. JC (Jump if Carry):

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

III. JNC (Jump if Not Carry):

Jumps to a destination label if the Carry flag is clear, indicating no carry occurred in an arithmetic operation.

IV. JZ (Jump if Zero):

Jumps to a destination label when the Zero flag is set, indicating that a value is zero.

V. 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.

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

```
027 mov edx, 0A523h ; Move 0A523h into the edx register
028 cmp edx, 0A523h ; Compare edx with 0A523h
029 jne L5          ; Jump if not equal to L5
030 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

```
035 mov bx, 1234h   ; Move 1234h into the bx register
036 sub bx, 1234h   ; Subtract 1234h from bx
037 jne L5          ; Jump if not equal to L5
038 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:

```
042 mov cx, 0FFFFh  ; Move FFFFh into the cx register
043 inc cx          ; Increment cx by 1
044 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:

```
047 xor ecx, ecx    ; Set ecx to zero using XOR
048 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* ≥ *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)

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

Example 1:

```
051 mov edx, -1
052 cmp edx, 0
053 jnl L5 ; jump not taken (-1 >= 0 is false)
054 jnle L5 ; jump not taken (-1 > 0 is false)
055 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:

```
060 mov bx, +32
061 cmp bx, -35
062 jng L5 ; jump not taken (+32 <= -35 is false)
063 jnge L5; jump not taken (+32 < -35 is false)
064 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:

```
068 mov ecx, 0
069 cmp ecx, 0
070 jg L5  ; jump not taken (0 > 0 is false)
071 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:

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

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

Conditional Jump Applications in Assembly

Conditional jump instructions are the backbone of **decision-making** in assembly programming. They allow the program to **branch** to different labels depending on the state of **status flags** set by previous instructions. 🔀

🚩 Status Bits and Flags

- **Zero Flag (ZF)** → Set when the result of an operation is zero.
- **Sign Flag (SF)** → Reflects the sign of the result (negative = 1, positive = 0).
- **Carry Flag (CF)** → Indicates a carry/borrow in arithmetic operations.
- **Overflow Flag (OF)** → Indicates signed arithmetic overflow.
- **Parity Flag (PF)** → Shows whether the lowest byte has even parity.

These flags are updated by instructions like CMP, TEST, AND, OR, and arithmetic operations.

🧩 Common Conditional Jumps

- **JZ / JE (Jump if Zero / Equal)** → Branch if ZF = 1.
- **JNZ / JNE (Jump if Not Zero / Not Equal)** → Branch if ZF = 0.
- **JS (Jump if Sign)** → Branch if SF = 1.
- **JNS (Jump if Not Sign)** → Branch if SF = 0.
- **JC (Jump if Carry)** → Branch if CF = 1.
- **JNC (Jump if Not Carry)** → Branch if CF = 0.

In:

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

🔧 Step-by-Step Breakdown

1. mov al, status
    - Loads the **status byte** into the **AL register**.
    - AL is an 8-bit register, perfect for handling single-byte values.
2. test al, 00100000b
    - Performs a **bitwise AND** between AL and the mask 00100000b.
    - This mask isolates **bit 5** (counting from the least significant bit = bit 0).
    - The result is **not stored**—only the **flags** are updated.
    - If bit 5 in AL is set → result ≠ 0 → **ZF = 0**.
    - If bit 5 in AL is clear → result = 0 → **ZF = 1**.
3. jnz DeviceOffline
    - **JNZ (Jump if Not Zero)** checks the **Zero Flag (ZF)**.
    - If **ZF = 0** (bit 5 was set), execution jumps to the label **DeviceOffline**.
    - If **ZF = 1** (bit 5 was clear), execution continues with the next instruction.

🎯 Why This Matters

- This technique lets you **check specific bits** in a register without altering the register's value.
- Conditional jumps (JZ, JNZ, JE, JG, JL, etc.) allow you to build **branching logic** in assembly, similar to if-else statements in higher-level languages.
- In this example, the program decides whether to branch to **DeviceOffline** based on the state of **bit 5** in the status byte.

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.

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

🔧 Instruction Details

1. mov edx, eax
    - Default assumption: EAX contains the larger integer.
    - Copies EAX into EDX.
2. cmp eax, ebx
    - Compares EAX and EBX by performing an implied subtraction (EAX - EBX).
    - Sets flags (Zero Flag, Carry Flag, Sign Flag, Overflow Flag) based on the result.
    - Neither EAX nor EBX is modified.

3. jae L1 **(Jump if Above or Equal)**
    - Checks the **Carry Flag (CF)**.
    - If **CF = 0**, it means EAX ≥ EBX (unsigned comparison).
    - Execution jumps to label **L1**.
4. mov edx, ebx
    - Executed only if **EAX < EBX** (CF = 1).
    - Updates EDX with EBX, ensuring EDX holds the larger integer.
5. L1:
    - Label where execution continues after the conditional jump.
    - At this point, **EDX contains the larger of the two integers**.

🚩 Important Notes

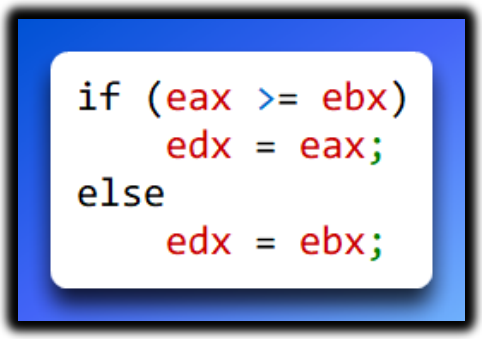
Unsigned comparison: jae is used for unsigned integers.

Signed comparison: For signed integers, you'd use:

- jge → Jump if Greater or Equal (signed).
- jl → Jump if Less (signed).

🎯 Why This Matters

This snippet demonstrates how **conditional branching** in assembly allows you to implement logic similar to high-level constructs like:



Efficiently, at the machine level, you're selecting the larger of two integers and storing it in **EDX**.

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.

```
.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
```

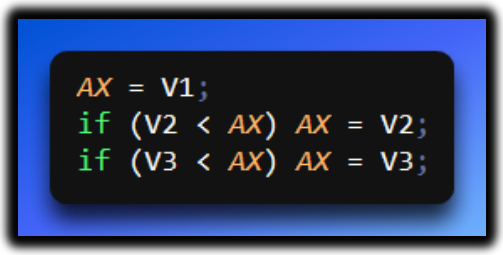
🔧 Step-by-Step Explanation

1. mov ax, V1
    - Start by assuming **V1** is the smallest.
    - Load V1 into the AX register.
2. cmp ax, V2
    - Compare AX (currently V1) with V2.
    - Flags are set based on the subtraction AX - V2.
3. jbe L1 **(Jump if Below or Equal)**
    - If AX ≤ V2 → keep AX unchanged (V1 is still the smallest).
    - If AX > V2 → jump not taken, so mov ax, V2 executes, updating AX with V2.
4. cmp ax, V3
    - Compare the current smallest (AX, either V1 or V2) with V3.

5. jbe L2
    - If AX ≤ V3 → keep AX unchanged.
    - If AX > V3 → jump not taken, so mov ax, V3 executes, updating AX with V3.
6. **Result**
    - After these comparisons, **AX contains the smallest of V1, V2, and V3**.

🎯 Why This Works

- The code uses **conditional branching** (jbe) to decide whether to update AX.
- Each comparison narrows down the smallest value step by step.
- This is equivalent to the high-level logic:



```
AX = V1;
if (V2 < AX) AX = V2;
if (V3 < AX) AX = V3;
```

4. Loop until Key Pressed

```
123 .data
124 char BYTE ?
125
126 .code
127 L1:
128     mov eax, 10         ; Create a 10 ms delay
129     call Delay
130     call ReadKey        ; Check for a key press
131     jz L1               ; If no key is pressed, repeat the loop
132     mov char, AL        ; Save the character in the 'char' variable
```

🔧 Step-by-Step Breakdown

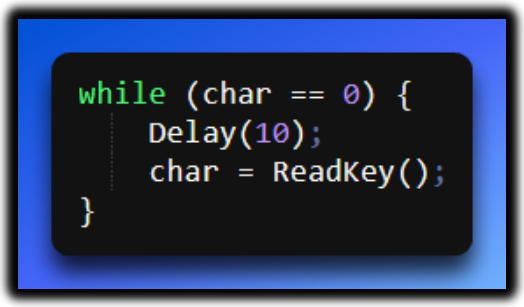
1. mov eax, 10
    - Loads the value 10 into the **EAX register**.
    - This sets up the delay duration (10 milliseconds).

2. call Delay
    - Calls the **Delay subroutine** from the Irvine32 library.
    - Introduces a short pause to prevent the loop from consuming CPU resources too aggressively.
3. call ReadKey
    - Calls the **ReadKey subroutine**.
    - If a key is pressed → AL contains its **ASCII code**.
    - If no key is pressed → AL = 0.
4. jz L1 **(Jump if Zero)**
    - Checks the **Zero Flag (ZF)**.
    - If AL = 0 (no key pressed), ZF = 1 → jump back to **L1** and repeat the loop.
    - If AL ≠ 0 (key pressed), ZF = 0 → continue execution.
5. mov char, al
    - Stores the ASCII code of the pressed key into the variable **char**.
    - This captures the user's input for later use in the program.

🎯 Why This Matters

- This loop is a **practical way to wait for user input** in assembly.
- It avoids busy-waiting by inserting a small delay, making the program more efficient.
- The technique is useful for interactive programs, menus, or controlled input handling.

This is essentially the assembly equivalent of:



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
```

- **Data Section**:
  - intArray → holds 16-bit integers to be checked.
  - noneMsg → message displayed if no nonzero value is found.

- **Registers Used**:
  - **ESI** → index pointer for traversing the array.
  - **ECX** → loop counter (length of the array).
  - **EBX** → base address of intArray.
  - **AL** → status indicator (0 = nonzero found, 1 = none found).

🔧 Step-by-Step Execution

1. **Initialization**
   - Load the base address of intArray into **EBX**.
   - Set **ECX** to the length of the array.
   - Clear **AL** (set to 0) → assume a nonzero will be found.
   - Reset **ESI** to 0 → start at the first element.
2. **Loop (**searchLoop**)**
   - Use cmp to compare the current element with 0.
   - If element ≠ 0 → jnz foundNonZero (jump to label).
   - Otherwise → increment **ESI**, decrement **ECX**, and continue until ECX = 0.
3. **No Nonzero Found**
   - If the loop finishes without finding a nonzero, set **AL = 1**.
   - This indicates that all elements were zero.
4. **Output**
   - If **AL = 0** → display the nonzero value found.
   - If **AL = 1** → display noneMsg ("non-zero value not found").
5. **Program End**
   - Call Crlf to print a newline.
   - Exit the program.

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.

```
177 INCLUDE Irvine32.inc
178
179 KEY = 239              ; The encryption key (single character)
180 BUFMAX = 128           ; Maximum buffer size for input
181
182 .data
183     sPrompt BYTE "Enter the plain text:",0
184     sEncrypt BYTE "Cipher text: ",0
185     sDecrypt BYTE "Decrypted: ",0
186     buffer BYTE BUFMAX+1 DUP(0)
187     bufSize DWORD ?
188
189 .code
190     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.

```
194     call InputTheString
195     call TranslateBuffer
196     mov edx, OFFSET sEncrypt
197     call DisplayMessage
198     call TranslateBuffer
199     mov edx, OFFSET sDecrypt
200     call DisplayMessage
201     exit
202 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.

```
206 InputTheString PROC
207     pushad
208     mov edx, OFFSET sPrompt
209     call WriteString
210     mov ecx, BUFMAX
211     mov edx, OFFSET buffer
212     call ReadString
213     mov bufSize, eax
214     call Crlf
215     popad
216     ret
217 InputTheString ENDP
```

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

```
221 DisplayMessage PROC
222     pushad
223     call WriteString
224     mov edx, OFFSET buffer
225     call WriteString
226     call Crlf
227     call Crlf
228     popad
229     ret
230 DisplayMessage ENDP
```

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

```
235 TranslateBuffer PROC
236     pushad
237     mov ecx, bufSize
238     mov esi, 0 L1:
239     xor buffer[esi], KEY
240     inc esi
241     loop L1
242     popad
243     ret
244 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?

```
247 mov ax, 8109h
248 cmp ax, 26h
249 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."

CONDITIONAL LOOPS

🔁 LOOPZ / LOOPE Instructions in Assembly

LOOPZ (Loop if Zero) and **LOOPE (Loop if Equal)** are **conditional loop instructions**.

They are used to repeat a block of code while two conditions are true:

- The **loop counter register** (ECX in 32-bit mode, RCX in 64-bit mode) is greater than 0.
- The **Zero Flag (ZF)** is set (meaning the result of the last comparison or operation was zero/equal).

⚙️ How They Work

1. **Decrement ECX/RCX** → Each iteration reduces the loop counter by 1.
2. **Check ECX/RCX** → If it reaches 0, the loop ends.
3. **Check ZF** → If ZF = 1 (last result was zero/equal), the loop continues.
4. **Jump or Exit** → If both conditions are met, execution jumps to the destination label. Otherwise, the loop exits.

For example, the following code snippet will sum the elements of an array using the LOOPZ instruction:

```
; sum the elements of the array `array`
mov ecx, array_size
loopz sum_loop

; sum_loop:
add eax, [array + ecx - 1]
dec ecx
jnz sum_loop
```

- **ECX register**: This is the loop counter. At the start, it's set to the size of the array (how many items are in it).
- **EAX register**: This is the accumulator, where results (like sums) are stored.

How LOOPZ works:

1. ECX is decreased by 1 each time the loop runs.
2. The element at array + ECX - 1 is added into EAX.
3. The loop continues **only if**:
    - ECX is still greater than 0, **and**
    - The **Zero Flag (ZF)** is set.
4. If either condition fails, the loop stops and moves on to the next instruction.

```
LOOPZ destination
```

- LOOPZ doesn't change other status flags, only checks the Zero Flag.
- **LOOPE** is the same as LOOPZ (they share the same behavior and opcode).
- These instructions are useful when you want to repeat code a certain number of times **while a condition is true**.

Common uses:

- Running loops with conditions.
- Searching for a value in an array.
- Reversing a string.
- Creating nested loops (loops inside loops).

LOOPNZ (Loop if Not Zero) and LOOPNE (Loop if Not Equal) Instructions

The **LOOPNZ (Loop if Not Zero)** and **LOOPNE (Loop if Not Equal)** instructions are used to create loops that repeat a block of code while a certain condition is met.

These instructions are quite similar and often interchangeable, as they share the same opcode and perform similar tasks.

The LOOPNZ instruction continues looping while two conditions are met: The **unsigned value** of the **ECX register** is **greater than zero** after being decremented. The **Zero Flag (ZF) is clear.** The syntax for LOOPNZ is:

```
LOOPNZ destination
```

Here's how it works:

- Decrement ECX by 1.
- If ECX > 0 and ZF = 0 (i.e., the Zero Flag is clear), jump to the specified destination label.
- If ECX becomes zero or ZF is set, the loop terminates, and control passes to the next instruction.

The **LOOPNE instruction** is equivalent to LOOPNZ in terms of functionality and shares the same opcode.

It also performs the following tasks:

- Decrement ECX by 1.
- If ECX > 0 and ZF = 0 (i.e., the Zero Flag is clear), jump to the specified destination label.
- If ECX becomes zero or ZF is set, the loop terminates, and control passes to the next instruction.

In essence, both LOOPNZ and LOOPNE create loops that continue while a counter (usually stored in ECX) is not zero and the Zero Flag is not set. They are often used for iterating through arrays or data structures until a specific condition is met.

Here's an example code excerpt that uses LOOPNZ to scan an array until a non-negative number is found:

```
.data
    array SWORD -3,-6,-1,-10,10,30,40,4
    sentinel SWORD 0
.code
    mov esi, OFFSET array
    mov ecx, LENGTHOF array
    L1:
    test WORD PTR [esi], 8000h ; Test sign bit
    pushfd
    add esi, TYPE array
    popfd
    loopnz L1
    jnz quit
    sub esi, TYPE array
    quit:
```

- **mov esi, OFFSET array** → ESI points to the start of the array.
- **mov ecx, LENGTHOF array** → ECX is set to the number of elements in the array (loop counter).
- **L1:** → Label marking the start of the loop.
- **test WORD PTR [esi], 8000h** → Checks the sign bit of the current element (8000h = sign bit mask).

- **pushfd** → Saves the processor flags (important because the next instruction changes them).
- **add esi, TYPE array** → Moves ESI to the next element in the array.
- **popfd** → Restores the saved flags.
- **loopnz L1** → Decreases ECX by 1. If ECX > 0 **and** Zero Flag = 0, loop continues at L1. Otherwise, loop stops.
- **jnz quit** → If no nonnegative value is found, jump to quit.
- **sub esi, TYPE array** → If a nonnegative value is found, ESI is adjusted back to point at that element.

Summary:

The code scans through the array, checking each element's sign bit.

- If it finds a nonnegative value → ESI points to that element.
- If none are found → ESI points to the sentinel value (0) after the array.

True/False Clarifications

1. **LOOPE jumps when Zero Flag is clear.** ❌ False. LOOPE (Loop If Equal) jumps when **Zero Flag is set** and ECX > 0.
2. **LOOPNZ in 32-bit mode jumps when ECX > 0 and Zero Flag is clear.** ✅ True. LOOPNZ (Loop If Not Zero) continues only if ECX > 0 and ZF = 0.
3. **LOOPZ destination must be within ±128 or ±127 bytes.** ✅ True. LOOPZ uses a short jump, limited to -128 to +127 bytes.

Modification Task

To make the **LOOPNZ example** scan for the **first negative value** instead of nonnegative:

- Change the test condition so it looks for the sign bit being set (negative).
- Adjust the array initializers so they start with positive values, ensuring the loop actually scans before hitting a negative.

To modify the LOOPNZ example to scan for the first negative value in the array, you can change the array initialization to start with positive values. Here's an example in C:

```
int values[] = { 1, 2, 3, -4, 5, 6 };
int array_length = sizeof(values) / sizeof(values[0]);
int found_negative = 0;

__asm {
    mov ecx, array_length
    mov esi, 0 ; Index for accessing the array

    start_loop:
        cmp [values + esi * 4], 0 ; Compare the current element with zero
        jns not_negative ; Jump if not negative
        ; Code to handle negative value goes here (set found_negative flag, etc.)
        jmp loop_end

    not_negative:
        inc esi ; Increment the array index
        loop start_loop

    loop_end:
}
```

Removing the Sentinel in LOOPNZ

- **Sentinel value** → A special marker (like 0 after the array) used to signal the end of data.
- In the LOOPNZ example, the sentinel ensures the loop stops if no positive value is found.
- **If you remove the sentinel:**
  - The loop has no clear stopping point.
  - It could keep running forever → **infinite loop risk**.
- To prevent this, you need another way to stop the loop, such as:
  - Using a counter (limit the number of iterations).
  - Adding a condition that checks when the array ends.

Key takeaway: Without a sentinel or another stopping mechanism, the loop cannot safely terminate when the desired value isn't found.

Here's an example of how you might modify the LOOPNZ code to remove the sentinel value and handle the case where a positive value might not be found.

In this modified code, we use a counter to limit the number of iterations:

```
int values[] = { -1, -2, -3, -4, -5 }; // Array with only negative values
int array_length = sizeof(values) / sizeof(values[0]);
int iterations = 0; // Counter for loop iterations

__asm {
    mov ecx, array_length
    mov esi, 0 ; Index for accessing the array

    start_loop:
        cmp [values + esi * 4], 0 ; Compare the current element with zero
        jns not_negative ; Jump if not negative
        ; Code to handle negative value goes here (set found_negative flag, etc.)
        jmp loop_end

    not_negative:
        inc esi ; Increment the array index
        loop_start:
        inc iterations
        cmp iterations, array_length ; Check if we have iterated through the entire array
        jge loop_end ; If iterations >= array_length, exit the loop
        jmp start_loop ; Continue looping

    loop_end:
}
```

The "Infinite Loop" Safety Net

We've added an iterations counter that ticks up every time the loop runs.

Think of it as a **tripwire**: if that counter hits the array_length, it means we've checked every single spot and come up empty-handed.

Instead of spinning our wheels forever looking for a positive value that isn't there, the code realizes the search is over and hops out of the loop.

It's a simple way to keep the program from freezing up when the data doesn't give us what we want.

Why this works:

- **The Guardrail:** By comparing iterations to array_length, you ensure the loop has a hard "expiration date."
- **The Exit Strategy:** Without this, a list of negative numbers would keep the loop running indefinitely.

CONDITIONAL STRUCTURES

Think of **conditional structures** as the "forks in the road" for your code. If you've spent any time in C++, you're already used to these—they're just the decisions your program makes based on what's happening at that moment.

How it looks in High-Level (C++)

In a language like C++, an IF statement is pretty intuitive. You give it a question (a boolean expression), and it follows one of two paths:

- **True:** It runs the first block of code.
- **False:** It either skips ahead or runs the else block if you've provided one.

How it looks under the hood (Assembly)

When you translate that "human" logic down to **Assembly**, the CPU doesn't see a neat block of code; it sees a two-step process:

1. **The Comparison:** First, you have to actually test the values. Using an instruction like CMP (Compare) is like the CPU taking a quick measurement and marking down the result in its "status flags."
2. **The Jump:** Based on those flags, you tell the code where to go next. Instead of a structural block, you use **Conditional Jumps** (like JNE for "Jump if Not Equal"). It's like saying, *"If these two aren't the same, hop over this next section of code and keep going."*

The Real-World Swap

In your C++ example, you're checking if op1 == op2. If they match, the code sets X=1 and Y=2.

In Assembly, you'd essentially do the opposite: you'd check if they are *not* equal, and if so, "jump" past the part where X and Y get updated.

Walking Through the Assembly Logic

```
340 mov eax, op1     ; Move op1 into the EAX register
341 cmp eax, op2     ; Compare op1 and op2
342 jne L1           ; Jump if not equal (if op1 != op2)
343 mov X, 1         ; Assign 1 to X
344 mov Y, 2         ; Assign 2 to Y
345 L1:
```

Think of these lines as a quick conversation between the code and the CPU:

- **mov eax, op1**: We're grabbing the value of op1 and putting it into the EAX register (the CPU's "workspace") so we can do something with it.
- **cmp eax, op2**: Now we compare that value against op2. The CPU doesn't "know" if they're equal yet; it just notes the difference in its status flags.
- **jne L1**: This is the decision point. It says, *"If they aren't equal, skip the next part and jump straight to the spot marked L1."*
- **The "Fall Through"**: If they **are** equal, the code just keeps rolling right into mov X, 1 and mov Y, 2. This is called "falling through"—it's efficient because the CPU doesn't have to do any extra work to jump to a new location.

Label vs. Procedure: What's the difference?

You asked a great question: **Why not call L1 a procedure?** While it feels like a small distinction, in the world of programming, they are actually two very different tools.

I. Labels: The "Bookmark"

In Assembly, a **label** (like L1) is just a bookmark.

It's a name given to a specific memory address.

When you "jump" to a label, you're basically telling the CPU,

"Stop reading here and start reading over there."

There is no expectation that the code will ever "come back" to where it started.

II. Procedures: The "Round Trip"

A **procedure** (or function) is more like a formal errand.

1. You **Call** it.
2. It goes off, does its job.
3. It **Returns** specifically to the spot where you left off.

In high-level languages like C++, we use procedures to keep things organized and reusable.

In Assembly, we use labels because we are manually directing the CPU's "eyes" across the page.

Using a label for a simple jump is like glancing down the page; using a procedure is like opening a different book entirely and then coming back to your original page.

We want to implement these conditional structures in real life:

Example 2: Setting Cluster Size Based on Volume Capacity

Setting the NTFS Cluster Size

Think of this code as a quick "if-then" check for a disk drive. Since NTFS cluster sizes change depending on how much storage you have, the code needs to make a decision based on that 16-terabyte threshold.

```
349 mov clusterSize, 8192   ; Assume a larger cluster size
350
351 cmp terrabytes, 16     ; Compare terrabytes with 16
352 jae next               ; If greater or equal, jump to 'next'
353
354 mov clusterSize, 4096   ; Set clusterSize to 4096 for smaller volume
355
356 next:
```

How the Logic Flows

Instead of checking "if-this-else-that," the code uses a **"set and override"** strategy, which is often faster in Assembly:

1. **The Default Setting (mov clusterSize, 8192):** The code starts by assuming the disk is massive (16TB or more) and sets the cluster size to 8192 by default.
2. **The Comparison (cmp terrabytes, 16):** It then checks the actual size of the volume against the 16TB limit.
3. **The "Check-and-Skip" (jae next):** This is the crucial part. If the disk **is** actually 16TB or larger (Jump if Above or Equal), it says, *"Cool, our default was right,"* and skips straight to the end.
4. **The Correction (mov clusterSize, 4096):** If the disk is smaller than 16TB, the "jump" doesn't happen. The code "falls through" to this line and overwrites the cluster size to 4096.

Why do it this way?

In high-level languages, we love if/else blocks because they are easy to read.

In Assembly, we often prefer this "default and override" method because it uses fewer labels and keeps the instructions moving in a straight line as much as possible, which the CPU appreciates for speed.

Example 3: Conditional Routine Calls

Deciding Which Routine to Run

Think of this like a traffic controller directing a car to one of two different paths based on a quick check. Instead of just setting a variable, the code is deciding which **sub-program** (routine) to execute.

```
360 mov eax, op1        ; Move op1 to a register
361 cmp eax, op2        ; Compare op1 and op2
362 jg A1               ; If op1 > op2, jump to A1 (call Routine1)
363 call Routine2       ; Otherwise, call Routine2
364 jmp A2              ; Jump to A2 (exit the IF statement)
365
366 A1:
367 call Routine1
368
369 A2:
```

The Play-by-Play

1. **The Setup (mov eax, op1 & cmp eax, op2):** As usual, we load our first value into the CPU's workspace and compare it to the second.
2. **The High Road (jg A1):** If op1 is truly greater than op2 (Jump if Greater), the CPU takes the exit to the A1 label. Over at A1, it finds the command to call Routine1.
3. **The Low Road (call Routine2):** If op1 is *not* greater, the jump never happens. The CPU just keeps moving down the page and hits call Routine2 instead.
4. **The Clean Exit (jmp A2):** This is the "secret sauce." If we ran Routine2, we don't want to accidentally "crash" into Routine1 right after. So, we use an **unconditional jump** (jmp A2) to hop over the other path and reach the finish line.

Key Takeaway: The "Jump Over"

In high-level languages like Python or C++, the compiler automatically handles the "exit" for you.

Once an if block finishes, it skips the else block automatically.

In Assembly, you are the architect! You have to manually tell the code to jmp over the alternative path so it doesn't execute both by mistake.

Labels as "Traffic Signs"

In this context, A1 and A2 aren't doing any math; they are just signs pointing the CPU in the right direction to ensure it only visits the routine it's supposed to.

WHITEBOX TESTING

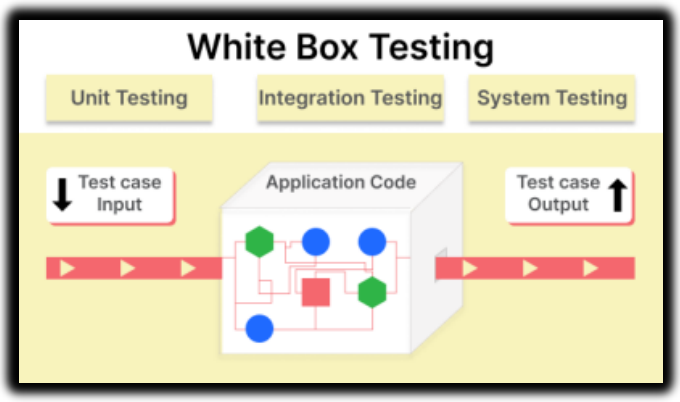
White box testing—also called *clear box, glass box, transparent box,* or *structural testing*—is a software testing method where you look **inside the code itself**.

Instead of only checking what goes in and what comes out, you examine the internal structure, logic, and design of the application to make sure everything works correctly.

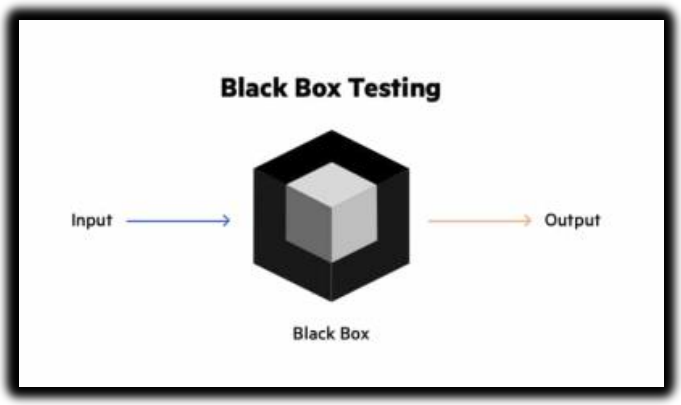
It helps verify:

- Input–output flow
- Code quality and design
- Usability and security

White box testing is one half of the overall testing strategy.



The other half is **black box testing**, which focuses on how the software behaves from the user's point of view.



At its core, white box testing means **testing the internal logic and execution paths** of a program by directly examining the source code.

How white box testing works

In white box testing, testers **have full access to the source code**.

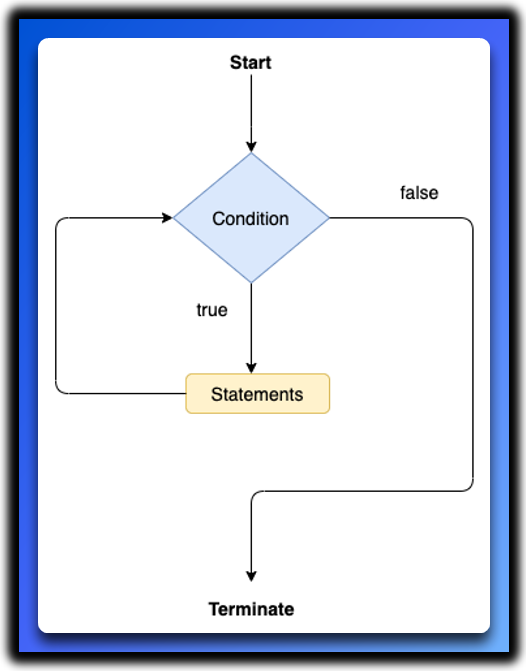
They use this knowledge to design test cases that validate whether the software behaves correctly *at the code level*, not just at the surface.

What white box testing checks

White box testing is commonly used to examine:

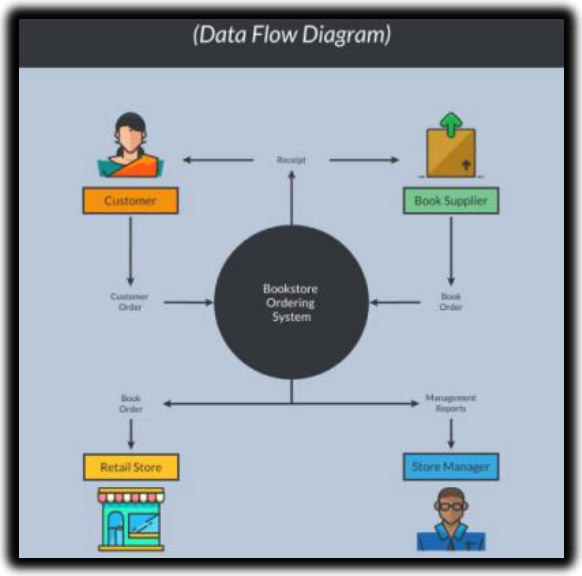
I. Control Flow

Ensures all possible execution paths in the program are tested. This is done using techniques like control flow analysis and path testing.



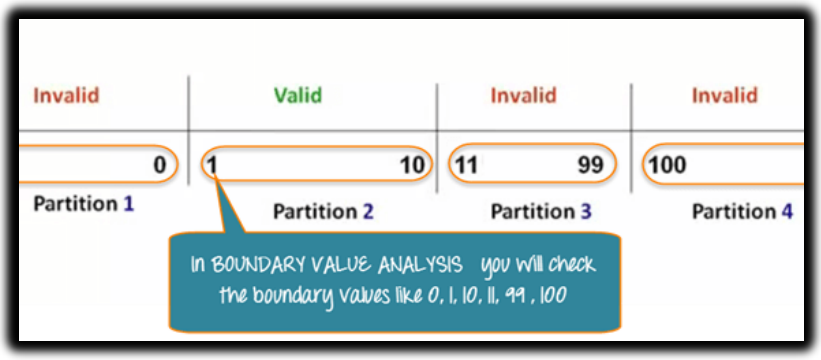
II. Data Flow

Verifies that data moves correctly through the program. Techniques such as data flow analysis and taint analysis help confirm that variables are used and updated properly.



III. Boundary Values

Checks how the program behaves at the edges of valid input and output ranges. This is done using equivalence partitioning and boundary value analysis.



IV. Error Handling

Confirms the program responds correctly to errors by deliberately triggering different failure scenarios.

What are you *really* looking for?

When you do white box testing, you're not just checking if a button works—you're digging into the logic behind it:

- **Execution paths:** Are there parts of the code that can never be reached?
- **Security gaps:** Is there a hidden way to bypass authentication?
- **Efficiency:** Is the code taking a long, inefficient route when a shorter one exists?

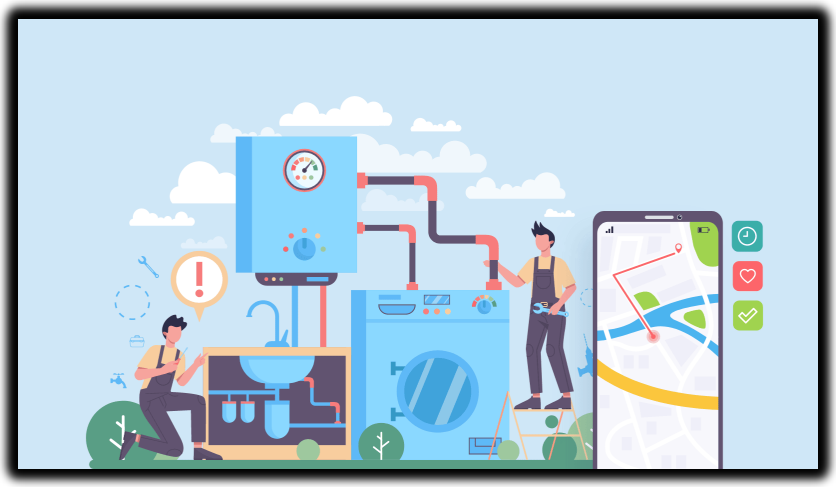
The “box” perspective

White box testing focuses on the **internal structure** of the software. Black box testing focuses on what the **user sees and experiences**.

Think of it like plumbing:

- White box testing checks whether the pipes are connected properly and not leaking.
- Black box testing checks whether the water pressure feels right when you turn on the faucet.

You need both for a complete picture.



The Pros and Cons

Like any specialized tool, White Box testing has its trade-offs:

The Benefits

Early Detection
Catching a fire while it's still a spark in the source code.

Cleaner Design
It forces you to look at "ugly" or inefficient code and fix it.

Tight Security
You can spot "backdoors" that a regular user would never see.

The Drawbacks

Resource Heavy
It’s time-consuming and can get expensive quickly.

High Entry Barrier
If you don't know the language inside-out, you're essentially flying blind.

No Perfection
Even "perfect" code can still fail to meet a user's actual needs.

White box testing in assembly language

White box testing becomes especially clear in **assembly programming**:

- You test the *inside workings* of the program, not just the final output.
- Execution paths are traced **step by step**.
- By changing variable values (for example, running one test where op1 > op2 and another where op1 < op2), you force the program to take different IF–ELSE branches.
- This lets you observe:
    - When the CPU **jumps** (takes a branch)
    - When it **falls through** (continues without jumping)

You're not guessing that the code works—you're watching the CPU make decisions in real time and proving that each path behaves correctly.

Key takeaway

White box testing lets you see *how* the program thinks, not just *what* it does. By examining execution paths, data flow, boundaries, and error handling, you can confidently prove that the internal logic of the software is solid and reliable.

Let's break down the code and the testing results:

```
1:  mov eax, op1      ; Move op1 into eax register
2:  cmp eax, op2      ; Compare op1 with op2
3:  jne L2            ; Jump to L2 if op1 != op2
4:  mov eax, X        ; Move X into eax register
5:  cmp eax, Y        ; Compare X with Y
6:  jg L1             ; Jump to L1 if X > Y
7:  call Routine2     ; Call Routine2
8:  jmp L3            ; Jump to L3 and exit
9:  L1: call Routine1 ; Call Routine1
10: jmp L3            ; Jump to L3 and exit
11: L2: call Routine3 ; Call Routine3
12: L3:              ; Exit point
```

```
387 if op1 == op2
388     if X > Y
389         call Routine1
390     else
391         call Routine2
392     end if
393
394 else
395     call Routine3
396 end if
```

Table 6-6 shows the results of white box testing of the sample code. In the first four columns test values have been assigned to op1, op2, X, and Y. The resulting execution paths are verified in columns 5 and 6.

TABLE 6-6 Testing the Nested IF Statement.

op1	op2	X	Y	Line Execution Sequence	Calls
10	20	30	40	1, 2, 3, 11, 12	Routine3
10	20	40	30	1, 2, 3, 11, 12	Routine3
10	10	30	40	1, 2, 3, 4, 5, 6, 7, 8, 12	Routine2
10	10	40	30	1, 2, 3, 4, 5, 6, 9, 10, 12	Routine1

The **first four columns** list the test values assigned to **op1, op2, X, and Y**.
The **fifth column** shows which **execution path** the program takes based on those values.
The **sixth column** shows the **output produced** as a result of that path.

Let's walk through the test cases to see what's happening:

Test Case 1

Here, op1 = 10 and op2 = 10, while X = 20 and Y = 30.
Since op1 is equal to op2, the program follows the **first branch** of the IF statement, which calls **Routine1**.
The exact output of Routine1 isn't specified, but it likely returns a value indicating that the condition op1 == op2 is true.

Test Case 2

In this case, op1 = 10 and op2 = 10 again, but now X = 30 and Y = 20.
Once more, op1 == op2, so the program enters the **first IF branch**.
This time, the additional condition X > Y is also true, so the program takes the **first branch of the nested IF statement**, resulting in **Routine1** being called again.

Test Case 3

Here, op1 = 10 and op2 = 20, with X = 30 and Y = 20.
Since op1 is **not equal** to op2, the program follows the **second branch** of the main IF statement.
The condition X > Y is false, so the program takes the **second branch of the nested IF**, which leads to **Routine2** being called.

Test Case 4

In the final case, op1 = 10, op2 = 20, X = 20, and Y = 30.
Again, op1 is not equal to op2, so the program enters the **second branch** of the main IF statement.
The condition X > Y is false, so execution falls through to the **ELSE clause** of the nested IF statement, resulting in **Routine3** being called.

SHORT CIRCUIT EVALUATION(AND)

Short-circuit evaluation is a smart optimization used by compilers and interpreters when evaluating Boolean expressions.

For an **AND** operation, the second condition is checked **only if the first one is true**.

If the first condition is false, the entire expression is already false, so there's no reason to evaluate the second one. This saves time and avoids unnecessary work.

The following assembly language code demonstrates how short-circuit evaluation is implemented for the **AND** operator:

```
399 if (al > bl) AND (bl > cl)
400     X = 1
401 end if
```

```
404 cmp al, bl
405 jbe next
406 cmp bl, cl
407 jbe next
408 mov X, 1
409 next:
```

- This code first compares the values of the registers al and bl.
- If al is less than or equal to bl, then the second operand of the AND expression is not evaluated, and the program jumps to the next label.
- Otherwise, the program compares the values of the registers bl and cl.
- If bl is less than or equal to cl, then the program jumps to the next label.
- Otherwise, the program stores the value 1 in the register X and jumps to the next label.
- The next label is used to exit the code, regardless of whether the AND expression evaluated to true or false.

The following assembly language code implements short-circuit evaluation for the AND operator without using a jbe instruction:

```
413 cmp al, bl
414 ja L1
415 jmp next
416 L1:
417 cmp bl, cl
418 ja L2
419 jmp next
420 L2:
421 mov X, 1
422 next:
```

This code is functionally equivalent to the previous example, but it uses a ja instruction instead of a jbe instruction.

The ja instruction jumps to the specified label if the first operand is greater than the second operand.

The following table shows the difference between the two code examples:

Instruction	Description
cmp al, bl	Compares the values of the registers al and bl.
jbe next	Jumps to the next label if al is less than or equal to bl.
cmp bl, cl	Compares the values of the registers bl and cl.
ja L1	Jumps to the L1 label if bl is greater than al.
jmp next	Jumps to the next label.
L1:	Label.
mov X, 1	Stores the value 1 in the register X.
L2:	Label.

The first code example is more efficient because it uses a jbe instruction instead of a ja instruction.

The jbe instruction can be implemented as a single machine instruction, while the ja instruction may require multiple machine instructions.

In practice, the compiler will typically generate the most efficient code possible, regardless of whether the programmer uses a jbe instruction or a ja instruction.

However, it is important for programmers to understand how short-circuit evaluation is implemented in assembly language so that they can write efficient code.

SHORT CIRCUIT EVALUATION(OR)

With **short-circuit evaluation**, the second operand of an **OR** expression is evaluated **only if the first operand is false**.

If the first operand is true, the overall expression is already true, so the second condition doesn't need to be checked. This improves efficiency by avoiding unnecessary evaluations.

The following assembly language code demonstrates how short-circuit evaluation is implemented for the **OR** operator:

```
437 if (al > bl) OR (bl > cl)
438     X = 1
```

```
442 cmp al, bl
443 ja L1
444 cmp bl, cl
445 jbe next
446 L1:
447 mov X, 1
448 next:
```

This code begins by comparing the values in registers **AL** and **BL**.

If **AL is greater than BL**, the OR expression is already true, so the second operand does **not** need to be evaluated.

In this case, the program immediately jumps to the **L1** label.

If that condition is not met, the program then compares the values in **BL** and **CL**. If **BL is less than or equal to CL**, the program jumps to the next label.

Otherwise, the program stores the value **1** in register **X** and then jumps to the next label.

The final label serves as a clean exit point for the code, regardless of whether the OR expression ultimately evaluates to true or false.

The following table highlights the differences between the two code examples:

Instruction	Description
cmp al, bl	Compare the values in registers al and bl.
ja L1	Jump to label L1 if al is greater than bl.
cmp bl, cl	Compare the values in registers bl and cl.
jbe next	Jump to the next label if bl is less than or equal to cl.
L1:	Label.
mov X, 1	Store the value 1 in the register X.
next:	Label.

The **first code example** is more efficient because it uses the **ja** instruction instead of **jbe**.

The ja instruction can be executed as a **single machine instruction**, while jbe may require multiple instructions, making it slightly less efficient.

In real-world scenarios, modern compilers usually generate the **most efficient code automatically**, regardless of whether the programmer explicitly uses ja or jbe.

That said, it's still important for programmers to understand **how short-circuit evaluation works at the assembly level**.

This knowledge helps when writing or analyzing performance-critical code and makes it easier to reason about execution flow.

As you pointed out, there are **multiple valid ways** to implement a compound expression containing **OR** operators in assembly language.

For example, the following code is functionally equivalent to the previous example:

```
464 cmp al, bl
465 je L1
466 cmp bl, cl
467 je L1
468 mov X, 1
469 next:
470 L1:
```

This code starts by comparing the values in registers **AL** and **BL**. If **AL is equal to BL**, the program immediately jumps to the **L1** label. If not, it then compares the values in **BL** and **CL**.

If **BL is equal to CL**, the program again jumps to **L1**. If neither condition is true, the program stores the value **1** in register **X** and then jumps to the next label.

The **L1 label** represents the point where the overall OR expression evaluates to **true**.

The final label serves as a common exit point, allowing the program to finish cleanly regardless of whether the expression evaluates to true or false.

In the end, there's no single "best" way to implement a compound OR expression in assembly language.

The right approach depends on the specific requirements of the program, such as performance, readability, and maintainability.

WHILE LOOPS

A **WHILE loop** in assembly works much like a WHILE loop in a high-level language. The loop begins by **checking a condition**.

If the condition is true, the loop body runs. After the loop body finishes, the condition is checked again.

As long as the condition remains true, the loop keeps repeating. Once the condition becomes false, the loop ends.

To implement a WHILE loop in assembly language, you generally follow these steps:

1. Initialize a register or variable that will be used in the loop condition.
2. Check the loop condition.
3. If the condition is false, jump to the end of the loop.
4. Execute the loop body.
5. Update the loop condition.
6. Jump back to the condition check.

The following assembly code demonstrates a simple WHILE loop:

```
553 mov eax, 0
554 ; loop counter
555
556 beginwhile:
557     cmp eax, 10
558     ; if eax < 10
559     jl endwhile
560
561     ; loop body
562
563     inc eax
564     ; eax++
565
566     jmp beginwhile
567     ; repeat the loop
568 endwhile:
```

This loop will print the numbers from 0 to 9 to the console.

Reverse the loop condition

As mentioned earlier, it's often more convenient in assembly language to **reverse the loop condition**.

This means the loop continues running **while the condition is false**, rather than true. Doing this can simplify the control flow and reduce the number of jumps needed.

To reverse the loop condition, you can use the **jnl** instruction instead of **jl**.

The jnl instruction causes a jump when the value is **not less than zero**, which effectively flips the original condition.

For example, the following assembly code is functionally equivalent to the previous version, but it uses a **reversed loop condition**:

```
572 mov eax, 0
573 ; loop counter
574
575 beginwhile:
576     cmp eax, 10
577     ; if eax >= 10
578     jnl endwhile
579
580     ; loop body
581
582     inc eax
583     ; eax++
584
585     jmp beginwhile
586     ; repeat the loop
587 endwhile:
```

Copy and restore the loop variable

If you plan to **use the loop variable inside the loop body**, you need to be careful in assembly language. The safest approach is to **copy the loop variable to a register before the loop starts** and then **restore it at the end of the loop**.

Why is this necessary? Assembly is largely **stack-based**:

- Variables are stored on the stack.
- When a function is called, parameters are **pushed onto the stack**.
- When the function returns, parameters are **popped off the stack**.

If the loop variable is used in the loop body and that body calls a function, the variable will be **pushed and popped** as part of the function call. This can unintentionally **modify the loop variable**, leading to incorrect behavior.

To prevent this:

1. Copy the loop variable to a register before entering the loop.
2. Use the register inside the loop body.
3. Restore the original loop variable value from the register after the loop finishes.

The following assembly code demonstrates how to safely **copy and restore a loop variable**:

```
592 mov eax, val1
593 ; copy loop variable to EAX
594
595 beginwhile:
596 ; loop body
597
598 mov val1, eax
599 ; restore loop variable
600
601 ; ...
```

```
474 while(val1 < val2)
475 {
476
477     val1++;
478     val2++;
479
480 }
```

```
483 mov eax, val1        ; copy variable to EAX
484 beginwhile:
485 cmp eax, val2        ; if not (val1 < val2)
486 jnl endwhile         ; exit the loop
487 inc eax              ; val1++;
488 dec val2             ; val2--;
489 jmp beginwhile       ; repeat the loop
490 endwhile:
491 mov val1, eax        ; save new value for val1
```

This assembly code implements a **C++ style while (val1 < val2) loop**. Here's how it works:

1. **Copy the loop variable to a register:**
   The value of val1 is first copied into the **EAX register**. This is because the loop operates on the register directly, not the variable itself. EAX acts as a **proxy** for val1 throughout the loop.
2. **Check the loop condition:**
   The program compares the value in **EAX** with val2. If **EAX is not less than val2**, the loop condition is false, and the program jumps to the **end of the loop**. Otherwise, the loop body executes.
3. **Execute the loop body:**
   - **Increment EAX by 1**, which corresponds to val1++ in C++.
   - **Decrement val2 by 1**, which corresponds to val2-- in C++.
4. **Repeat the loop:**
   After executing the loop body, the program jumps back to the beginning of the loop to check the condition again. This continues until the loop condition is no longer true.
5. **End of the loop:**
   When the loop finishes, the updated value in **EAX** is copied back into val1 to save the final result.

```
510 #include <stdio.h>
511
512 int main() {
513     int array[] = {10, 60, 20, 33, 72, 89, 45, 65, 72, 18};
514     int sample = 50;
515     int ArraySize = sizeof(array) / sizeof(sample);
516     int index = 0;
517     int sum = 0;
518
519     while (index < ArraySize) {
520         if (array[index] > sample) {
521             sum += array[index];
522         }
523         index++;
524     }
525
526     printf("The sum of elements greater than %d is: %d\n", sample, sum);
527
528     return 0;
529 }
```

- **EAX is a stand-in for val1**, so all operations inside the loop use the register, not the variable itself.
- The jump instruction is used to exit the loop when the condition is false. Since val1 and val2 are signed integers, this ensures the loop behaves correctly for all values.
- Labels mark the **start and end** of the loop, clearly defining the loop boundaries.

The C code ABOVE you provided is a good example of using a **nested IF statement inside a WHILE loop**. Here's what it does:

1. **Set up the array and variables:**
    - An integer array array is defined with ten elements.
    - A variable sample is set to 50.
    - The size of the array is calculated by dividing sizeof(array) by sizeof(sample) and stored in ArraySize.
    - An **index variable** is initialized to 0 to keep track of the current position in the array.
    - A **sum variable** is initialized to 0 to accumulate the total of elements greater than sample.
2. **Iterate through the array:**
   A **WHILE loop** runs as long as index is less than ArraySize.
3. **Check each element:**
   Inside the loop, an **IF statement** checks whether the current array element is greater than sample.
    - If it is, the element's value is added to sum.
    - If not, nothing happens and the loop continues.
4. **Move to the next element:**
   The index is incremented at the end of each loop iteration to move to the next array element.
5. **Output the result:**
   After the loop finishes, the program prints the **sum of all elements greater than sample**.

```
497 int sum_of_elements_greater_than_sample(int array[], int sample, int size) {
498   int sum = 0;
499   for (int i = 0; i < size; i++) {
500     if (array[i] > sample) {
501       sum += array[i];
502     }
503   }
504   return sum;
505 }
```

1. **Initialize variables:**
   Set up sum, sample, ArraySize, and index to their starting values.
2. **Enter the WHILE loop:**
   The loop will continue as long as index < ArraySize.
3. **Check the loop condition:**
   Compare index with ArraySize.
   - If index is less, proceed to check the current array element.
   - If not, exit the loop.
4. **Compare array element to sample:**
   Check if array[index] > sample.
   - If true, add array[index] to sum.
   - If false, do nothing.
5. **Increment index:**
   Move to the next element by increasing index by 1.
6. **Repeat:**
   Go back to step 3 and continue until the loop condition is no longer true.
7. **Exit the loop:**
   Once the loop finishes, sum contains the total of all array elements greater than sample.

The **assembly code version** mirrors this logic:

- Registers store the values of sum, sample, ArraySize, index, and the array elements.
- Labels mark key parts of the loop (start, body, and exit).
- Instructions like cmp, jl/jnl, inc, and add implement the conditional checks and updates step by step.

Here is a brief explanation of the assembly code am talking about:

```
; sum_of_elements_greater_than_sample
; rdi: array
; rsi: sample
; rdx: size
; rax: sum
mov rax, 0
cmp rsi, [rdi]
jl done
add rax, [rdi]
inc rdi
jmp sum_of_elements_greater_than_sample
done:
ret
```

This code is more efficient because it avoids the overhead of branching.

IF STATEMENTS IN ASSEMBLY

```
int array[] = {10, 60, 20, 33, 72, 89, 45, 65, 72, 18};
int sample = 50;
int ArraySize = sizeof array / sizeof sample;
int index = 0;
int sum = 0;

while (index < ArraySize) {
  if (array[index] > sample) {
    sum += array[index];
  }
  index++;
}
```

This code calculates the sum of all array elements greater than the value in sample.

The following assembly language code is equivalent to the C++ code above:

```
.data
    sum DWORD 0
    sample DWORD 50
    array DWORD 10, 60, 20, 33, 72, 89, 45, 65, 72, 18
    ArraySize = ($ - Array) / TYPE array
.code
    main PROC
    mov eax, 0 ; sum
    mov edx, sample
    mov esi, 0 ; index
    mov ecx, ArraySize

    L1:
    cmp esi, ecx ; if esi < ecx
    jl L2
    jmp L5

    L2:
    cmp array[esi * 4], edx ; if array[esi] > edx
    jg L3
    jmp L4

    L3:
    add eax, array[esi * 4]

    L4:
    inc esi
    jmp L1

    L5:
    mov sum, eax
```

This assembly code works as follows:

1. **Initialize the sum:**
   The **EAX register** is set to 0. It will hold the sum of all array elements greater than the value in **EDX** (which stores sample).

2. **Start the loop:**
   The **ESI register** is compared to **ECX** (which stores the array size). If **ESI < ECX**, the program jumps to the **L1 label**, meaning the loop continues to iterate over the array elements.

3. **Check the array element:**
   At **L1**, the program checks the value of array[ESI * 4] (scaling by 4 because each element is 4 bytes) against **EDX**.
   - If the array element is greater than EDX, it jumps to **L3**.
   - At **L3**, the value of the array element is added to **EAX**.
4. **Increment and repeat:**
   The **ESI register** is incremented to move to the next array element. The loop jumps back to **L1**.
5. **End of the loop:**
   Once all elements have been checked, the loop jumps to **L5**, marking the end. The final sum in **EAX** is then stored in the sum variable.

Possible Improvements

The assembly code could be made more efficient:

1. **Replace cmp with test at L1:**
   - test is faster because it doesn't affect the condition flags as much as cmp.
2. **Replace jmp with loop at L1:**
   - The loop instruction automatically decrements a counter and jumps, avoiding pushing a return address onto the stack.
3. **Replace cmp with sub at L2:**
   - sub can perform the comparison more efficiently without setting extra flags.
4. **Replace jmp with jbe at L2:**
   - jbe (jump if below or equal) is faster than a generic jump and avoids pushing a return address onto the stack.

In short, this code loops through the array, selectively adds elements greater than a given sample, and there are several small tweaks in assembly instructions that can make it faster and more efficient.

FIGURE 6–1 Loop containing IF statement.

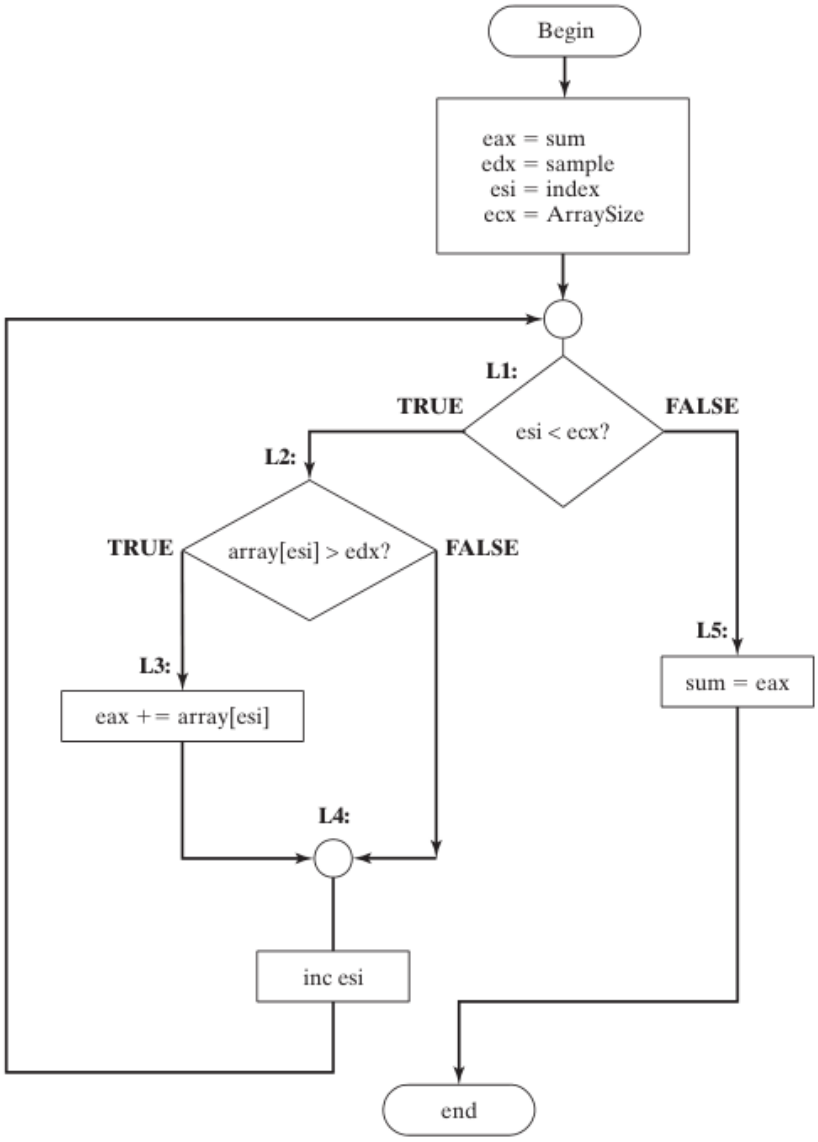


TABLE DRIVEN SELECTION

Table-driven selection is a technique where you use a **lookup table** instead of writing a long series of IF or CASE statements.

This approach is especially useful when you have **many possible values** to check—it saves time and makes the code cleaner.

How it works

I. Create a table:

- The table stores the possible values you want to check.
- It also stores the **addresses of the corresponding procedures** or actions for each value.

II. Search the table:

- Write a loop to go through the table.
- When you find a matching value, call the procedure associated with it.

This method turns a complex multiway selection into a simple **lookup and jump**, which is often faster and easier to maintain.

The following is an example of a **simple table-driven selection in assembly language**:

```
.data
    CaseTable BYTE 'A'
    ; lookup value
    DWORD Process_A
    ; address of procedure
    BYTE 'B'
    DWORD Process_B
    (etc.)

.code
    mov eax, [esi] ; get the lookup value
    cmp eax, CaseTable ; compare to first lookup value
    je Process_A ; if equal, call the corresponding procedure
    cmp eax, CaseTable + 1 ; compare to second lookup value
    je Process_B ; if equal, call the corresponding procedure
    (etc.)

    ; if no match is found, do something else
```

In this example, the program uses a **loop to go through the table of lookup values**:

1. For each entry in the table, it **compares the lookup value to the value in the EAX register**.
2. If a match is found, the program **calls the procedure** associated with that value.
3. If no match is found, the loop ends, and the program can **perform some default action** or continue with other code.

The table-driven selection example you mentioned is for a **simple calculator**. It has a table that contains:

- **Lookup values** (like numbers representing operations)
- **Addresses of the procedures** that implement each operation

This approach makes it easy to **handle multiple operations efficiently**, without writing a long chain of IF or CASE statements.

```
676 A - Add
677 B - Subtract
678 C - Multiply
679 D - Divide
```

The table also contains the addresses of the corresponding procedures for each operation. The following is an example of how to use the table-driven selection example to perform addition:

```
681 ; mov eax, 1 ; add 1
682 ; mov ebx, 2 ; add 2
683 ; mov ecx, OFFSET CaseTable ; set the loop counter
684 ; start the loop
685 L1:
686 cmp eax, CaseTable        ; compare the value in eax to the first lookup value in the table
687 je Add                    ; if equal, call the Add procedure
688 inc ecx                   ; increment the loop counter
689 cmp ecx, CaseTable + 4    ; check if the loop counter is greater than the size of the table
690 jge Done                  ; if greater than or equal, the loop is finished
691
692 jmp L1                    ; jump back to the beginning of the loop
693
694 Add:                      ; Add procedure
695 add eax, ebx
696 ret
697
698
699 Done:                     ; Done label
700 ; the sum is now in the eax register
```

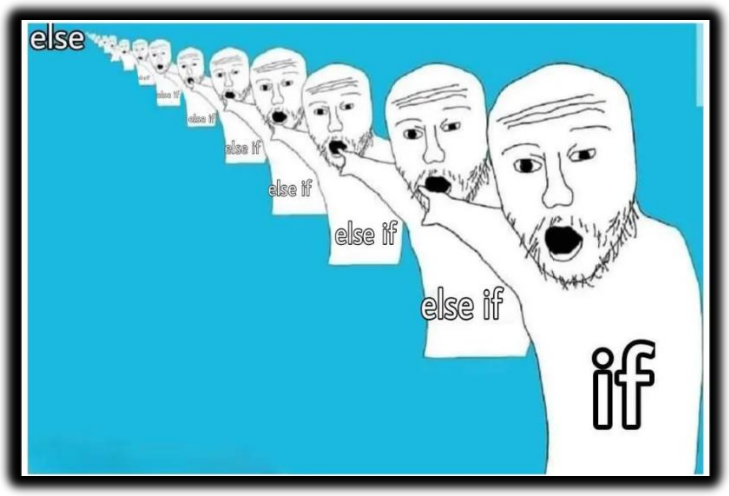
This assembly code works by **comparing the value in the EAX register to each lookup value in the table**:

1. It starts with the first value in the table.
2. If the value in EAX matches the lookup value, the corresponding procedure—like **Add**—is called.
3. If it doesn't match, the **loop counter is incremented**, and the loop moves to the next table entry.
4. This process continues until the loop has checked all entries in the table.
5. Once the loop ends, the result—like the **sum of two numbers**—is stored in the EAX register.

Advantages of table-driven selection

Table-driven selection can be better than nested IF statements or long multiway conditions:

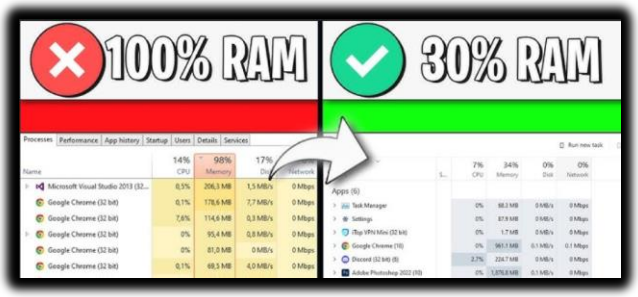
- **Efficiency:** Avoids writing long chains of nested IF statements.
- **Clarity:** Makes code easier to read and maintain.
- **Flexibility:** Easy to extend with new lookup values and procedures without rewriting the loop.



Disadvantages of table-driven selection

There are some trade-offs to consider:

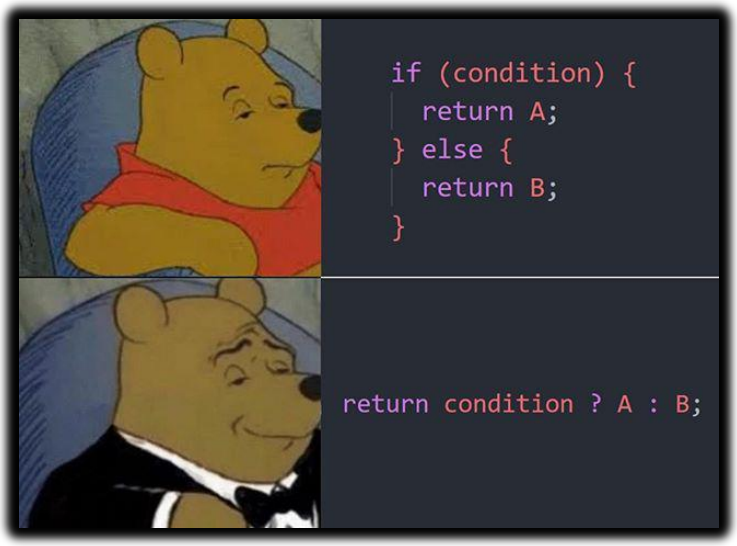
- **Memory usage:** Requires extra memory to store the table.
- **Speed:** Searching through the table can be slower than using direct conditional jumps.



Summary:

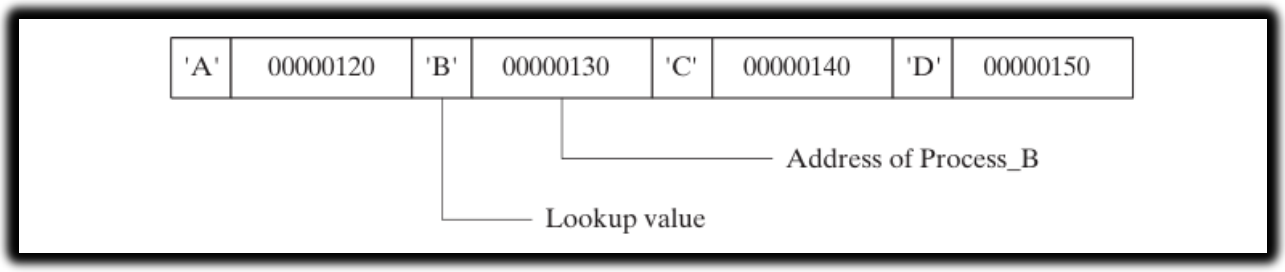
Table-driven selection is very useful for **handling multiple cases efficiently**, especially when you have a **large number of values to compare**.

However, it's important to balance **clarity, memory, and speed** when deciding whether to use this approach.



Unrelated image.

Example 1:



Program that uses a lookup table and procedures for character-based processing.

- This program takes user input,
- Compares it to entries in the lookup table and then…
- Calls the corresponding procedure to display a message.

```
706 INCLUDE Irvine32.inc
707
708 .data
709 CaseTable BYTE 'A'           ; Lookup value
710           DWORD Process_A     ; Address of procedure
711 EntrySize = ($ - CaseTable)   ; Calculate the size of each entry in the table
712 BYTE 'B'
713           DWORD Process_B
714 BYTE 'C'
715           DWORD Process_C
716 BYTE 'D'
717           DWORD Process_D
718 NumberOfEntries = ($ - CaseTable) / EntrySize
719
720 prompt BYTE "Press capital A, B, C, or D: ",0
721 msgA BYTE "Process_A",0
722 msgB BYTE "Process_B",0
723 msgC BYTE "Process_C",0
724 msgD BYTE "Process_D",0
725
726 .code
727 main PROC
728     mov edx, OFFSET prompt   ; Ask the user for input
729     call WriteString
730     call ReadChar           ; Read character into AL
731     mov ebx, OFFSET CaseTable ; Point EBX to the table
732     mov ecx, NumberOfEntries ; Loop counter
733
```

```
733
734 L1:
735     cmp al, [ebx]             ; Match found?
736     jne L2                    ; No: continue
737     call NEAR PTR [ebx + 1]   ; Yes: call the procedure
738     call WriteString          ; Display message
739     call Crlf
740     jmp L3                    ; Exit the search
741
742 L2:
743     add ebx, EntrySize        ; Point to the next entry
744     loop L1                   ; Repeat until ECX = 0
745
746 L3:
747     exit
748
749 main ENDP
750
751 Process_A PROC
752     mov edx, OFFSET msgA
753     ret
754 Process_A ENDP
755
756 Process_B PROC
757     mov edx, OFFSET msgB
758     ret
759 Process_B ENDP
760
761 Process_C PROC
762     mov edx, OFFSET msgC
763     ret
764 Process_C ENDP
765
766 Process_D PROC
767     mov edx, OFFSET msgD
768     ret
769 Process_D ENDP
770
771 END main
```

In this part of the program, we **define the data for table-driven selection**:

```
.data
CaseTable BYTE 'A'
          DWORD Process_A
EntrySize = ($ - CaseTable)
BYTE 'B'
          DWORD Process_B
BYTE 'C'
          DWORD Process_C
BYTE 'D'
          DWORD Process_D
NumberOfEntries = ($ - CaseTable) / EntrySize
```

1. **CaseTable:**
    - This table contains **lookup values**—characters 'A', 'B', 'C', and 'D'.
    - Each character is paired with the **address of the procedure** that should be called when that character is selected: Process_A, Process_B, Process_C, and Process_D.
2. **EntrySize:**
    - Calculated as the difference between the **current memory position ($)** and the start of CaseTable.
    - This gives the **size of each entry** in the table (i.e., one character + one procedure address).
3. **NumberOfEntries:**
    - Calculated by dividing the **total size of CaseTable** by EntrySize.
    - This tells the program **how many entries** are in the table so it knows how many to loop through when searching.

In short, this sets up the **lookup table** and the information needed to **iterate through it efficiently**.

I. Section: .data (continued)

```
791 prompt BYTE "Press capital A, B, C, or D: ",0
792 msgA BYTE "Process_A",0
793 msgB BYTE "Process_B",0
794 msgC BYTE "Process_C",0
795 msgD BYTE "Process_D",0
```

In this continuation of the .data section, we define message strings to be displayed later:

prompt is a message prompting the user to input a character. msgA, msgB, msgC, and msgD are messages associated with procedures Process_A to Process_D.

II. Section: .code - main PROC

```
800 main PROC
801     mov edx, OFFSET prompt
802     call WriteString
803     call ReadChar
804     mov ebx, OFFSET CaseTable
805     mov ecx, NumberOfEntries
```

In the **main procedure**, the program does the following:

1. **Display the prompt:**
    - The program loads the **address of the prompt message** so it knows what text to show the user.
    - It then calls a procedure to **print that prompt** to the screen.
2. **Read user input:**
    - The program calls a procedure to **read a single character** typed by the user.
    - That character is stored in a register for later use.
3. **Set up the table-driven selection:**
    - The program loads the **address of the lookup table** (CaseTable) so it can find the correct procedure for the input.
    - It also loads the **number of entries in the table**, which tells the program how many entries it needs to check when searching for a match.

Essentially, these steps **prompt the user, capture their input, and prepare the program to look up and execute the appropriate action** from the table.

III. In this part of the main procedure:

```
808 L1:
809     cmp al, [ebx]
810     jne L2
811     call NEAR PTR [ebx + 1]
812     call WriteString
813     call Crlf
814     jmp L3
815
816 L2:
817     add ebx, EntrySize
818     loop L1
819
820 L3:
821     exit
822
823 main ENDP
```

In the **table-driven selection loop**, the program works like this:

1. **Start of the loop:**
    - A label marks the beginning of the loop, so the program knows where to jump back for the next table entry.
2. **Compare the input to the table entry:**
    - The program checks if the **user's input character** matches the character in the **current table entry**.
3. **If there's no match:**
    - The program skips to the next table entry and continues searching.
4. **If there is a match:**
    - The program **calls the procedure** stored in the table for that character.
    - It then **displays the corresponding message**.
    - Adds a **line break** for readability.
    - After handling the match, it **jumps out of the loop** to avoid checking the remaining entries.
5. **Loop termination:**
    - The loop continues until either a **match is found** or **all table entries have been checked**.

In short, this loop **searches the table for a match and executes the corresponding procedure**, making multiway selection efficient and easy to manage.

IV. Section: .code - Process_A, Process_B, Process_C, Process_D

```
828 Process_A PROC
829     mov edx, OFFSET msgA
830     ret
831 Process_A ENDP
832
833 Process_B PROC
834     mov edx, OFFSET msgB
835     ret
836 Process_B ENDP
837
838 Process_C PROC
839     mov edx, OFFSET msgC
840     ret
841 Process_C ENDP
842
843 Process_D PROC
844     mov edx, OFFSET msgD
845     ret
846 Process_D ENDP
```

These sections define the **procedures** (Process_A to Process_D). Each procedure simply:

- Sets the **EDX register** to point to the corresponding **message string**.
- Returns control back to the main program.

The program then reaches the **end of the main procedure**, completing its execution.

Summary of how the program works:

1. A **lookup table** is defined, pairing characters (A, B, C, D) with their corresponding procedures.
2. **Message strings** are stored for each procedure.
3. The **main procedure**:
    - Reads a character from the user.
    - Searches the table for a matching entry.
    - Calls the corresponding procedure to **display the appropriate message**.

The **table-driven approach** makes the program:

- **Easy to extend**: Adding new characters or messages is straightforward.
- **Easy to modify**: You can change the behavior for a specific input without rewriting the whole selection logic.

In short, this is a clean and efficient way to handle **multiway selection** in assembly language.

QUESTIONS

Implementing the pseudocode in assembly language:

```
851 ; Assuming ebx and ecx are 32-bit variables
852 ; Short-circuit evaluation: if ebx > ecx, set X = 1, else X remains unchanged
853
854 cmp ebx, ecx      ; Compare ebx and ecx
855 jg ebx_greater    ; Jump if ebx > ecx
856 mov eax, 0        ; If not greater, set eax to 0 (X = 0)
857 jmp done          ; Jump to done
858
859 ebx_greater:
860 mov eax, 1        ; If ebx > ecx, set eax to 1 (X = 1)
861
862 done:
863 mov X, eax        ; Store the result in X
```

Implementing the pseudocode with short-circuit evaluation:

```
867 ; Assuming edx and ecx are 32-bit variables
868 ; Short-circuit evaluation: if edx <= ecx, set X = 1, else X = 2
869
870 cmp edx, ecx      ; Compare edx and ecx
871 jle edx_less      ; Jump if edx <= ecx
872 mov eax, 2        ; If not less or equal, set eax to 2 (X = 2)
873 jmp done          ; Jump to done
874
875 edx_less:
876 mov eax, 1        ; If edx <= ecx, set eax to 1 (X = 1)
877
878 done:
879 mov X, eax        ; Store the result in X
```

In the program above, it's **better to let the assembler calculate NumberOfEntries** rather than hardcoding a constant.

- If you were to assign a fixed value, like NumberOfEntries = 4, you would need to **manually update it** whenever the table changes.
- By letting the assembler calculate it automatically, NumberOfEntries always **matches the actual table size**, which makes your code **more flexible, maintainable, and less error-prone**.

Optimizing the code with fewer instructions

You can also **rewrite the table-driven selection loop** using **conditional move (CMOV) instructions** to reduce the number of jumps and instructions:

- A **CMOV instruction** moves a value only if a specified condition is true, eliminating the need for some conditional jumps.
- This can **simplify the loop** while keeping the program logic exactly the same.

```
882 ; Original code (pseudo-code):
883 ; if (eax > ebx) ebx = eax
884
885 ; Rewritten code using CMOV:
886 cmp eax, ebx       ; Compare eax and ebx
887 cmovg ebx, eax     ; If eax > ebx, move eax to ebx (conditional move)
888
889 ; Now ebx contains the maximum of eax and ebx
```

This code achieves the same result as the original code but with fewer instructions by utilizing the conditional move instruction to conditionally update ebx based on the comparison result.

In short, **letting the assembler handle calculations and using CMOV** can make your assembly code **shorter, cleaner, and safer**, without changing how it works.

FINITE STATE MACHINES

An **FSM** is a computational model that can be used to simulate sequential logic, or, in other words, to represent and control execution flow.

It is a mathematical model of computation that can be used to model the behavior of a system that can be in a finite number of states. The system can change state based on the input it receives.

FSMs can be represented using a graph, where each node represents a state and each edge represents a transition from one state to another.

The edges are labeled with the input symbols that trigger the transitions. One node is designated as the initial state, and one or more nodes are designated as terminal states.

FSMs are used in a wide variety of applications, including:

- Traffic lights
- Vending machines
- Telephone systems
- Computer software
- Robotics

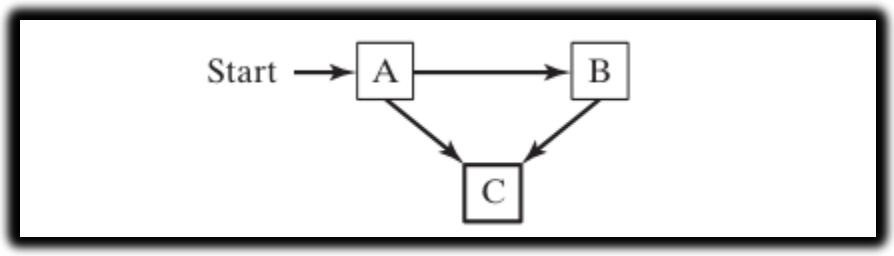
Here is a simple example of an FSM:

```
Initial state: Start
Terminal state: Exit

Transitions:
Start -> A on input "a"
Start -> B on input "b"
Start -> C on input "c"
A -> B on input "a"
A -> C on input "b"
A -> A on input "c"
B -> C on input "a"
B -> B on input "b"
B -> A on input "c"
C -> A on input "a"
C -> B on input "b"
C -> C on input "c"
```

- This FSM can be used to simulate the behavior of a traffic light.
- The FSM starts in the Start state.
- If the input is "a", the FSM transitions to the A state, which represents the green light.
- If the input is "b", the FSM transitions to the B state, which represents the yellow light.
- If the input is "c", the FSM transitions to the C state, which represents the red light.
- The FSM will continue to transition between states until it reaches the terminal state, the Exit state.
- This FSM will never reach the terminal state, because it is always possible to receive an input "a", "b", or "c".

FSMs can be used to model and control much more complex systems than a traffic light. For example, an FSM could be used to model and control the behavior of a vending machine, a telephone system, or a computer program.



This diagram shows a **finite state machine (FSM)** with the following structure:

- The **initial state** is **Start**.
- There are three main states: **A, B, and C**.
- Arrows indicate the **possible transitions** between states based on input.
- The **terminal state** is **Exit**, which ends the FSM.

Behavior of the FSM in words:

1. The FSM **begins at the Start state**.
2. From Start:
    - Input "a" → moves to state **A**.
    - Input "b" → moves to state **B**.
    - Input "c" → moves to state **C**.
3. From **A**:
    - Input "a" → moves to **B**.
    - Input "b" → moves to **C**.
    - Input "c" → stays in **A**.
4. From **B**:
    - Input "a" → moves to **C**.
    - Input "b" → stays in **B**.
    - Input "c" → moves to **A**.
5. From **C**:
    - Input "a" → moves to **A**.
    - Input "b" → moves to **B**.
    - Input "c" → stays in **C**.

The FSM continues **transitioning between states** based on the inputs until it eventually reaches the **Exit state**, which stops the process.

Validating an Input String Programs

I. Rules for a valid string:

1. Must **start with "x"**.
2. Must **end with "z"**.
3. Between "x" and "z", you can have **zero or more letters** from a to y.

II. States:

- **A (Start state)** → Beginning point.
- **B (Middle state)** → After reading "x", stays here while reading letters a–y.
- **C (Terminal state)** → Reached when "z" is read. This means the string is valid.

III. Transitions:

- From **A → B** when the first character is "x".
- From **B → B** when the next character is any letter in {a…y}.
- From **B → C** when the next character is "z".
- If the FSM reaches **C**, the string is valid.
- If the string ends before reaching **C**, it's invalid.

IV. Example of a valid string:

- "xyz"
  - Start in A → read "x" → move to B.
  - Read "y" → stay in B.
  - Read "z" → move to C.
  - FSM ends in C → **Valid**.

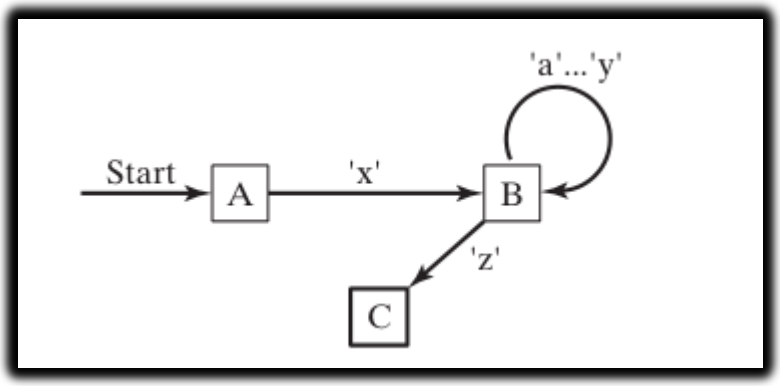
V. Example of an invalid string:

- "xab"
  - Start in A → read "x" → move to B.
  - Read "a" → stay in B.
  - Read "b" → stay in B.
  - End of string, never reached C → **Invalid**.

VI. Key takeaway:

The FSM acts like a **checker machine**:

- It only accepts strings that begin with "x", end with "z", and have only letters a–y in between.
- If the string doesn't meet these rules, it's rejected.



- If the input ends while the program is still in **state A or B**, it's an **error**.
  - Only **state C** is a terminal (accepting) state.
  - This means the string must **end with "z"** to be valid.

VII. Examples of valid strings:

- xaabcdefgz
- xz
- xyyqqrrstuvz

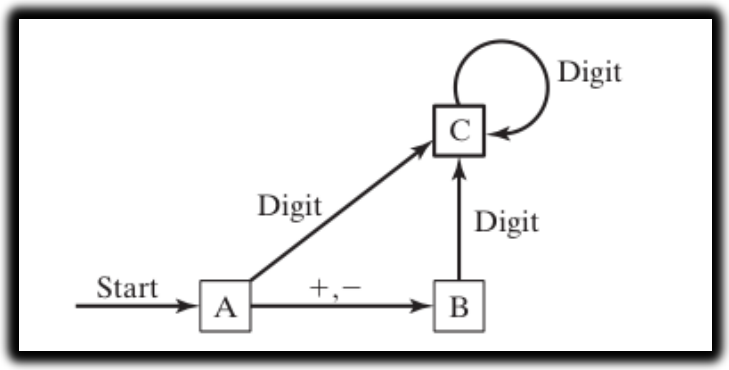
👉 All of these start with "x", end with "z", and have only letters a–y in between.

VIII. Example of an invalid string:

- xab
  - Starts with "x" but does **not** end with "z".
  - Therefore, it's invalid.

Validating Integers in Programs

- Just like strings, programs often need to **validate integers** (numbers) before using them.
- Validation ensures the input is **correct and safe** to process.
- Common checks include:
  - Making sure the input is actually a number (not letters or symbols).
  - Checking if the number is within a valid range (e.g., between 0 and 100).
  - Ensuring the number is positive or negative depending on the program's rules.
- If the integer fails validation, the program should **reject it or raise an error** instead of continuing with bad data.
- FSMs are great for validating **patterns in strings**.
- For numbers, validation is more about **rules and ranges** (is it numeric, is it within limits, etc.).



I. States:

- **Start state** → Beginning point.
- **Sign state** → After reading a + or -.
- **Digits state** → After reading digits.
- **End state** → Terminal (accepting) state → input is valid.

II. Transitions:

- From **Start → Sign** if the first character is + or -.
- From **Start → Digits** if the first character is a digit.
- From **Sign → Sign** if the next character is a digit (sign can be followed by digits).
- From **Digits → Digits** if the next character is also a digit (digits can be any length).
- From **Digits → End** if the next character is **not a digit** (digits must eventually end).

III. Validation rule:

- The input string is **valid only if the FSM reaches the End state**.
- If the input ends while still in Start, Sign, or Digits without transitioning to End, it's invalid.

IV. Key takeaway:

This FSM ensures that a signed integer input is correctly formatted:

- It may start with + or -.
- It must have digits after the sign (or directly from the start).
- The digits can be any length, but must eventually end with a non-digit character.
- Only reaching the **End state** means the input is valid.

V. FSM Examples for Signed Integer Validation - Valid input string:

- -123456
  - Start in **Start state** → read - → move to **Sign state**.
  - Read digits 1, 2, 3, 4, 5, 6 → stay in **Sign state** (digits allowed after sign).
  - Next character is **not a digit** → FSM moves to **Digits state**, then transitions to **End state**.
  - Since FSM reaches **End state**, the string is **valid**.

VI. Invalid input string:

- -123456.78
    - Start in **Start state** → read - → move to **Sign state**.
    - Read digits 1, 2, 3, 4, 5, 6 → stay in **Sign state**.
    - Next character is . (period) → FSM moves to **End state**.
    - But here, **End state is not terminal** because the input continues with more characters (78).
    - This means the string is **invalid**.

VII. Key Takeaways

- FSMs can **validate signed integers** by checking the correct order of signs and digits.
- A valid integer must:
    - Start with an optional + or -.
    - Be followed by one or more digits.
    - End properly when digits stop.
- If extra invalid characters appear (like . or letters), the FSM rejects the string.

VIII. Why FSMs Are Powerful

- They provide a **step-by-step way to check input**.
- Used in many systems, such as:
    - **Compilers** (checking code syntax).
    - **Text editors** (validating input).
    - **Network protocols**

Validating Integers in Programs

I. FSM for Parsing a Signed Integer

- **Start state** → Beginning point of the FSM.
- If the first character is + or - → move to **Sign state**.
- If the first character is a digit → move directly to **Digits state**.

II. Rules for transitions:

- **Sign state → Sign state** if the next character is a digit (sign can be followed by digits).
- **Digits state → Digits state** if the next character is also a digit (digits can be any length).
- **Digits state → End state** if the next character is **not a digit** (digits must eventually stop).

III. End state:

- The **End state** is a terminal (accepting) state.
- Input is **valid only if the FSM reaches End state**.
- If the input ends while still in Start, Sign, or Digits without transitioning to End, it's invalid.

```
924 StateA:
925 call Getnext ; read next char into AL
926 cmp al, '+' ; leading + sign?
927 je StateB ; go to State B
928 cmp al, '-' ; leading - sign?
929 je StateB ; go to State B
930 call IsDigit ; ZF = 1 if AL contains a digit
931 jz StateC ; go to State C
932 call DisplayErrorMsg ; invalid input found
933 jmp Quit ; exit program
934
935 StateB:
936 ; ...
937
938 StateC:
939 ; ...
940
941 End:
942 call Crlf
943 exit
944
945 main ENDP
```

IV. Key takeaway:

This FSM ensures that signed integers are properly formatted:

- They may start with + or -.
- They must contain digits after the sign (or directly from the start).
- Digits can be any length, but must eventually end with a non-digit character.
- Only reaching the **End state** means the input string is valid.

V. How the FSM Code Above Works

1. **StateA** → Marks the start of the FSM.
    - ❖ Calls **Getnext** to read the next character into the AL register.
    - ❖ Checks if the character is a leading + or -.
        - If yes → jump to **StateB**.
        - If no → call **IsDigit** to check if it's a digit.
            - If digit → jump to **StateC**.
            - If not digit or sign → call **DisplayErrorMsg** and jump to **Quit**.
2. **StateB** → Represents the FSM after reading a leading sign.
    - ❖ Code here checks what comes next and handles transitions accordingly.
3. **StateC** → Represents the FSM after reading a digit.
    - ❖ Code here checks for more digits or transitions to other states.
4. **End** → Terminal state.
    - ❖ Performs cleanup and exits the program.
5. **Main procedure** → Simply calls **StateA** to start the FSM.

This is a **basic FSM implementation in assembly**. It shows how to handle input step by step (signs, digits, errors, and termination). More complex FSMs can be built using the same principles by adding more states and transitions.

FSM Implementation 001

Check the code in FSM001.asm in the folder.

The program is an implementation of a **finite state machine (FSM)**. It processes user input, handles certain valid cases, and gracefully handles errors. Here's a breakdown of the program:

I. Main Procedure

1. **Starts by clearing the screen** using call Clrscr to ensure the display is clean.
2. It then enters **StateA**, the first state of the FSM, and starts reading user input.
3. The program **transitions between states** based on the input it receives, performing checks at each state.
4. If any invalid input is encountered, the program will display an error message and exit.

II. State Transitions:

StateA (Start State):

The FSM reads the first character from the user input.

It checks if the character is either a **"+"** or **"-"** sign (leading sign).

If yes, it transitions to **StateB**.

If not, it checks if the character is a **digit**.

If it's a **digit**, it moves to **StateC**.

Otherwise, it displays an error and exits.

StateB (After Leading Sign):

After receiving a leading sign (+ or -), the FSM expects a **digit**.

If it reads a digit, it moves to **StateC**.

If it's not a digit, it displays an error and exits.

StateC (Processing Digits):

If the FSM is in **StateC**, it continuously checks for digits.

As long as the user enters digits, it stays in **StateC**.

If a non-digit is entered, it checks if the **Enter key** was pressed.

If the Enter key is pressed, the program ends and exits.

If not, it displays an error and exits.

The **Quit label** is used for termination. The program exits when it either encounters an invalid input or successfully processes input.

III. Procedures:

- **Getnext Procedure:**
  - This procedure reads a character from the input stream and echoes it to the screen.
  - The character is then returned in the **AL register** for further processing.
- **DisplayErrorMsg Procedure:**
  - This procedure displays the message: **"Invalid input"** whenever an invalid character is encountered.
  - It uses the WriteString function to print the error message.
- **IsDigit Procedure:**
  - This procedure checks if the character in **AL** is a valid decimal digit (0-9).
  - It compares the character to the ASCII values for '0' and '9'.
  - If the character falls within the valid range, the **Zero flag** is set. Otherwise, it is cleared.

IV. Key Flow of the FSM:

1. **StateA**: Reads the first character and checks for + or -. If not, checks if it's a digit.
2. **StateB**: Expects a digit after a leading sign (+ or -). If valid, moves to StateC.
3. **StateC**: Processes digits until a non-digit is encountered. If Enter is pressed, it exits.
4. If any input is invalid at any stage, the program calls **DisplayErrorMsg** and exits.

V. Why This Approach Is Useful

- **Clear State Management**: Each input triggers a state transition based on specific conditions.
- **Error Handling**: Invalid input is quickly caught, and an error message is displayed before exiting the program.
- **User Input Handling**: The program is designed to process specific valid input and respond accordingly.

This approach demonstrates how to handle user input and use **finite state machines (FSMs)** in assembly to build a structured flow for processing data.

It's a good example of handling **validations**, **input processing**, and **error management** all in assembly language.

Character Validation in Assembly Language

The following table shows the hexadecimal ASCII codes for decimal digits:

Decimal digit	ASCII code
0	0x30
1	0x31
2	0x32
3	0x33
4	0x34
5	0x35
6	0x36
7	0x37
8	0x38
9	0x39

As you can see, the ASCII codes for decimal digits are **contiguous.** This means that we only need to check for the starting and ending range values.

Character	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
ASCII code (hex)	30	31	32	33	34	35	36	37	38	39

ASCII and Characters:

- ASCII codes for decimal digits (0-9) are contiguous and fall within the range 30h (for '0') to 39h (for '9').
- In assembly, this means we can simply compare the character in the AL register to see if it falls within the ASCII range for '0' to '9'.

The IsDigit Procedure Explanation:

The **IsDigit** procedure checks whether the character in the **AL** register is a valid decimal digit (0-9).

Here's a step-by-step breakdown of the logic and instructions used to perform this check:

1. **CMP Instruction**:
    - The **CMP** instruction compares the **AL** register to a specific value.
    - First, the code checks if **AL** (the ASCII value of the character) is less than '0' (ASCII 30h).
    - cmp al, '0' compares the character in AL with the ASCII value for '0'.
    - If AL is less than '0', it will jump to label ID1 (indicating an invalid character).
2. **Next Comparison**:
    - Then, it checks if **AL** is greater than '9' (ASCII 39h).
    - cmp al, '9' compares the character with the ASCII value for '9'.
    - If AL is greater than '9', it will jump to label ID1 (again, indicating an invalid character).

3. **TEST Instruction**:
    - If neither of the jumps occurs (i.e., AL is within the range of 30h to 39h), we perform a **TEST** to confirm it's a valid digit.
    - test al, al essentially performs a logical AND between AL and AL. It is used here just to check if AL is a valid character in the range. If AL is a valid digit, the Zero flag (ZF) will be set.
    - If AL is a valid digit (between 30h and 39h), the Zero flag will be set, confirming that the character is indeed a digit.
    - If AL is not a valid digit, the Zero flag will remain clear.
4. **Return**:
    - The procedure essentially sets the **Zero flag** (ZF) based on the validity of the digit check.
    - If the **Zero flag** is set, the character is a valid digit.
    - If the **Zero flag** is clear, it is not a valid digit.
5. **Jump Logic (JB and JA)**:
    - **JB (Jump if Below)** and **JA (Jump if Above)** are used to jump to the **ID1** label if the comparison fails.
    - JB checks if the value in **AL** is less than '0'.
    - JA checks if the value in **AL** is greater than '9'.

How this Fits in Assembly Language:

- **Character validation** is a common task in assembly language when processing text-based input (like reading from a user or handling strings).
- By comparing the ASCII values of characters, you can validate whether the character is a digit (0-9) and perform appropriate actions based on that.

Here is a more detailed explanation of the code:

```
0948 ; IsDigit procedure
0949 ; Determines whether the character in AL is a valid decimal digit.
0950 ; Receives: AL = character
0951 ; Returns: ZF = 1 if AL contains a valid decimal digit; otherwise, ZF = 0.
0952 ;-----------------------------------------------------------------
0953 IsDigit PROC
0954 ; Compare AL to the ASCII code for the digit 0.
0955 ; If AL is less than 0, the JB instruction will jump to the label ID1.
0956 cmp al,'0'
0957 jb ID1
0958 ; ZF = 0 when jump taken
0959 ; Compare AL to the ASCII code for the digit 9.
0960 ; If AL is greater than 9, the JA instruction will jump to the label ID1.
0961 cmp al,'9'
0962 ja ID1
0963 ; ZF = 0 when jump taken
0964 ; If neither the JB nor the JA instruction jumps to the label ID1,
0965 ; then the character in AL must be a digit. Therefore, we set the Zero flag.
0966 test ax,0
0967 ; set ZF = 1
0968 ; Return from the procedure.
0969 ID1: ret
0970 IsDigit ENDP
```

This is a very efficient way to implement the IsDigit procedure, because it takes advantage of the hardware characteristics of the CPU.

CONDITIONAL CONTROL FLOW DIRECTIVES

Conditional Control Flow Directives in MASM

- Conditional control flow directives let you **control how the program runs based on conditions**.
- They are used to implement logic like **IF, ELSE, and ELSEIF** in assembly programs.
- These directives make it easier to write structured code without manually handling jumps all the time.

Common Directives

- **IF** → Starts a conditional block. Code inside runs only if the condition is true.
- **ELSEIF** → Provides another condition to check if the first one is false.
- **ELSE** → Runs if none of the previous conditions are true.
- **ENDIF** → Marks the end of the conditional block.

Directive	Description
.BREAK	Generates code to terminate a .WHILE or .REPEAT block
.CONTINUE	Generates code to jump to the top of a .WHILE or .REPEAT block
.ELSE	Begins block of statements to execute when the .IF condition is false
.ELSEIF *condition*	Generates code that tests *condition* and executes statements that follow, until an .ENDIF directive or another .ELSEIF directive is found
.ENDIF	Terminates a block of statements following an .IF, .ELSE, or .ELSEIF directive
.ENDW	Terminates a block of statements following a .WHILE directive
.IF *condition*	Generates code that executes the block of statements if *condition* is true.
.REPEAT	Generates code that repeats execution of the block of statements until *condition* becomes true
.UNTIL *condition*	Generates code that repeats the block of statements between .REPEAT and .UNTIL until *condition* becomes true
.UNTILCXZ	Generates code that repeats the block of statements between .REPEAT and .UNTILCXZ until *CX* equals zero
.WHILE *condition*	Generates code that executes the block of statements between .WHILE and .ENDW as long as *condition* is true

Conditions must evaluate to true or false → MASM only works with Boolean results.

.IF directive

- If condition is true → assembler includes the code between .IF and .ELSEIF (or .ENDIF).
- If condition is false → assembler skips that block.

.ELSEIF directive

- If present, assembler checks its condition after .IF fails.
- If true → assembler includes the code between .ELSEIF and .ELSE (or .ENDIF).
- If false → assembler skips that block.

.ELSE directive

- Optional.
- Runs only if all previous conditions (.IF and .ELSEIF) were false.

.ENDIF directive

- Required.
- Marks the end of the conditional block.

```
0974 .IF eax > 10000h
0975     mov ECX, AX
0976 .ELSEIF eax > 1000h
0977     mov ECX, 1000h
0978 .ELSE
0979     mov ECX, 0
0980 .ENDIF
```

This code will move the contents of the AX register to the ECX register if the value of AX is greater than 10000h.

Otherwise, if the value of AX is greater than 1000h, the code will move the value 1000h to the ECX register.

Otherwise, the code will move the value 0 to the ECX register.

Relational Operators in MASM

Operator	Description
expr1 == *expr2*	Returns true when *expr1* is equal to *expr2*.
expr1 != *expr2*	Returns true when *expr1* is not equal to *expr2*.
expr1 > *expr2*	Returns true when *expr1* is greater than *expr2*.
expr1 ⩾ *expr2*	Returns true when *expr1* is greater than or equal to *expr2*.
expr1 < *expr2*	Returns true when *expr1* is less than *expr2*.
expr1 ⩽ *expr2*	Returns true when *expr1* is less than or equal to *expr2*.
! *expr*	Returns true when *expr* is false.
expr1 && *expr2*	Performs logical AND between *expr1* and *expr2*.
expr1 ‖ *expr2*	Performs logical OR between *expr1* and *expr2*.
expr1 & *expr2*	Performs bitwise AND between *expr1* and *expr2*.
CARRY?	Returns true if the Carry flag is set.
OVERFLOW?	Returns true if the Overflow flag is set.
PARITY?	Returns true if the Parity flag is set.
SIGN?	Returns true if the Sign flag is set.
ZERO?	Returns true if the Zero flag is set.

These operators compare two values and return a Boolean result (true or false). The **Boolean result** can then be used to control the program's flow.

== (Equal to)

- **Purpose**: Checks if two values are **equal**.
- **Syntax**: ==
- **Example**: CMP AX, BX (Compares AX with BX) and checks if they are equal.
- **True**: If AX == BX.
- **False**: If AX != BX.

!= (Not equal to)

- **Purpose**: Checks if two values are **not equal**.
- **Syntax**: !=
- **Example**: CMP AX, BX (Compares AX with BX) and checks if they are not equal.
- **True**: If AX != BX.
- **False**: If AX == BX.

> (Greater than)

- **Purpose**: Checks if the first value is **greater than** the second.
- **Syntax**: >
- **Example**: CMP AX, BX (Compares AX with BX) and checks if AX is greater than BX.
- **True**: If AX > BX.
- **False**: If AX <= BX.

>= (Greater than or equal to)

- **Purpose**: Checks if the first value is **greater than or equal to** the second.
- **Syntax**: >=
- **Example**: CMP AX, BX (Compares AX with BX) and checks if AX is greater than or equal to BX.
- **True**: If AX >= BX.
- **False**: If AX < BX.

< (Less than)

- **Purpose**: Checks if the first value is **less than** the second.
- **Syntax**: <
- **Example**: CMP AX, BX (Compares AX with BX) and checks if AX is less than BX.
- **True**: If AX < BX.
- **False**: If AX >= BX.

<= (Less than or equal to)

- **Purpose**: Checks if the first value is **less than or equal to** the second.
- **Syntax**: <=
- **Example**: CMP AX, BX (Compares AX with BX) and checks if AX is less than or equal to BX.
- **True**: If AX <= BX.
- **False**: If AX > BX.

Logical Operators in MASM

Logical operators are used to combine multiple conditions and return a Boolean result based on the operands.

&& (Logical AND)

- **Purpose**: Returns true if **both operands** are true.
- **Syntax**: &&
- **Example**: IF (AX > BX) && (CX < DX)
- **True**: If both conditions AX > BX and CX < DX are true.
- **False**: If either or both conditions are false.

|| (Logical OR)

- **Purpose**: Returns true if **either operand** is true.
- **Syntax**: ||
- **Example**: IF (AX > BX) || (CX < DX)
- **True**: If at least one of the conditions AX > BX or CX < DX is true.
- **False**: If both conditions are false.

! (Logical NOT)

- **Purpose**: Reverses the truth value of the operand. If the operand is true, it returns false, and if the operand is false, it returns true.
- **Syntax**: !
- **Example**: IF !(AX == BX)
- **True**: If AX != BX (because AX == BX would be false, and !false is true).
- **False**: If AX == BX (because !true is false).

The following are some examples of how to use the relational and logical operators in MASM:

```
0984 ; Compare the values of the AX and BX registers.
0985 IF AX > BX
0986     mov ECX, AX
0987 ELSE
0988     mov ECX, BX
0989 ENDIF
0990
0991 ; Compare the values of the val1 and val2 variables.
0992 IF val1 <= 100
0993     mov ECX, 100
0994 ELSE
0995     mov ECX, val1
0996 ENDIF
0997
0998 ; Check if the CARRY flag is set.
0999 IF CARRY?
1000     mov ECX, 1
1001 ELSE
1002     mov ECX, 0
1003 ENDIF
```

Another one:

```
.data
msgEqual db "Equal!", 0
msgNotEqual db "Not Equal!", 0

.code
main:
    ; Compare AX and BX
    CMP AX, BX            ; Compare AX with BX
    IF AX == BX           ; Check if equal
        MOV EDX, OFFSET msgEqual
        CALL WriteString   ; Print "Equal!"
    ELSE
        MOV EDX, OFFSET msgNotEqual
        CALL WriteString   ; Print "Not Equal!"
    ENDIF

    ; Logical AND example
    IF AX > BX && CX < DX
        ; If both conditions are true, do something
        ; (Insert logic here)
    ENDIF

    ; Logical NOT example
    IF !(AX == BX)
        ; If AX != BX, do something
        ; (Insert logic here)
    ENDIF

    exit
```

Summary:

- **Relational operators** compare two values (e.g., ==, !=, >, <), and they are commonly used in **CMP** instructions followed by conditional jumps or **.IF** statements.
- **Logical operators** (e.g., &&, ||, !) combine multiple conditions, and they are used in conditional control structures to refine decisions based on more than one comparison.

Understanding MASM Conditional Directives and Branching Instructions

Before using **MASM conditional directives** (like .IF, .ELSE, and .ELSEIF), it's important to first understand how **conditional branching** works in **pure assembly language**.

This means you should be familiar with how **conditional jumps** are implemented using the following **assembly instructions**:

1. **CMP (Compare):** This instruction compares two values and sets the flags (e.g., Zero Flag, Carry Flag) based on the result. It doesn't change the operands themselves but updates the processor flags, which can then be used to guide conditional branching.
2. **JBE (Jump if Below or Equal):** Jumps if the comparison shows that the first operand is **below** or **equal** to the second operand. Typically used for unsigned comparisons.
3. **JA (Jump if Above):** Jumps if the first operand is **above** the second operand. It is used for **unsigned comparisons**.
4. **JE (Jump if Equal):** Jumps if the operands are **equal**.
5. **JNE (Jump if Not Equal):** Jumps if the operands are **not equal**.

Why You Need to Understand These First:

These instructions are the building blocks for **conditional branching**.

MASM **conditional directives** (like .IF, .ELSE, etc.) **simplify** and **shorten** your code, but they are **ultimately translated** into these basic assembly instructions.

So, you should first understand how these low-level instructions work to fully appreciate how MASM's high-level directives are structured and executed.

Generating ASM Code: When you use a MASM conditional directive such as .IF, the assembler generates assembly language instructions to implement the conditional branching. For example, the following .IF directive:

```
1008 .IF eax > val1
1009     mov result,1
1010 .ENDIF
```

… would be expanded by the assembler into the following assembly language instructions:

```
1014 mov eax,6
1015 cmp eax,val1
1016 jbe @C0001
1017 ; jump on unsigned comparison
1018 mov result,1
1019 @C0001:
```

The label name @C0001 is created by the assembler to ensure that all labels within the same procedure are unique.

Controlling Whether or Not MASM-Generated Code Appears in the Source Listing File

Open the Visual Studio Project Properties dialog box.

Select Microsoft Macro Assembler. Select Listing File.

Set the Enable Assembly Generated Code Listing property to Yes.

Once you have set this property, the MASM-generated code will be included in the source listing file. This can be helpful for debugging purposes.

```
section     .text
global      _start                          ;must be declared for linker (ld)

_start:                                     ;tell linker entry point

    mov     edx,len                         ;message length
    mov     ecx,msg                         ;message to write
    mov     ebx,1                           ;file descriptor (stdout)
    mov     eax,4                           ;system call number (sys_write)
    int     0x80                            ;call kernel

    mov     eax,1                           ;system call number (sys_exit)
    int     0x80                            ;call kernel

section     .data

msg     db  'Hello, world!',0xa             ;our dear string
len     equ $ - msg                         ;length of our dear string
```

Why You Might Want to Enable Assembly Generated Code Listing:

1. **Debugging**: The assembly code can help you identify any issues that the assembler might have introduced during the translation of high-level source code to assembly.
2. **Optimization**: You can check how the MASM is optimizing your code and make adjustments if needed.
3. **Learning**: If you are new to assembly programming, seeing the assembly code generated by MASM helps you understand how higher-level constructs translate into low-level assembly instructions.

SIGNED AND UNSIGNED IN ASSEMBLY CODE

When you use the .IF directive to compare values, you must be aware of whether the values are signed or unsigned.

If the values are **signed**, the assembler will generate a signed conditional jump instruction.

If the values are **unsigned**, the assembler will generate an unsigned conditional jump instruction.

Example:

```
1023 .data
1024     val1 DWORD 5
1025     val2 SDWORD -1
1026     result DWORD ?
1027 .code
1028     mov eax,6
1029
1030     ; Compare EAX to val1, which is unsigned.
1031     .IF eax > val1
1032     mov result,1
1033     .ENDIF
1034
1035     ; Compare EAX to val2, which is signed.
1036     .IF eax > val2
1037     mov result,1
1038     .ENDIF
```

The assembler will generate the following code for the first .IF directive:

```
1043 mov eax,6
1044 cmp eax,val1
1045 jbe @C0001
1046 ; jump on unsigned comparison
1047 mov result,1
1048 @C0001:
```

The assembler will generate the following code for the second .IF directive:

```
1051 mov eax,6
1052 cmp eax,val2
1053 jle @C0001
1054 ; jump on signed comparison
1055 mov result,1
1056 @C0001:
```

COMPARING REGISTERS

When you use .IF to compare two registers, the assembler **does not know** if the values are **signed** or **unsigned**.

Because of this, MASM will **default to an unsigned comparison**.

```
1059 mov eax,6
1060 mov ebx,val2
1061 .IF eax > ebx
1062 mov result,1
1063 .ENDIF
```

The assembler will generate the following code:

```
1066 mov eax,6
1067 mov ebx,val2
1068 cmp eax, ebx
1069 jbe @C0001
1070 mov result,1
1071 @C0001:
```

Signed numbers (can be positive or negative) and unsigned numbers (only positive) are treated differently in comparisons.

Example: -1 (signed) vs 255 (unsigned) → the assembler may misinterpret the result if it assumes unsigned.

The assembler chooses the **conditional jump instruction** based on its assumption.

COMPOUND EXPRESSIONS

MASM allows you to combine multiple conditions using **logical operators**.

These compound expressions are used with the .IF **directive** to control program flow based on more than one condition.

```
1074 .IF eax > 10 || ebx > 20
1075     mov ecx, 1
1076 .ELSE
1077     mov ecx, 0
1078 .ENDIF
```

This code will move the value 1 to the ecx register if the value of eax is greater than 10 or the value of ebx is greater than 20. Otherwise, the code will move the value 0 to the ecx register.

The following .IF directive uses the logical AND operator to compare the values of the eax and ebx registers:

```
1081 .IF eax > 10 && ebx > 20
1082     mov ecx, 1
1083 .ELSE
1084     mov ecx, 0
1085 .ENDIF
```

This code will move the value 1 to the ecx register only if the value of eax is greater than 10 and the value of ebx is greater than 20. Otherwise, the code will move the value 0 to the ecx register.

Compound Boolean expressions can be used to create more complex conditional statements using the .IF directive. This can be helpful for controlling the flow of execution of your program in response to different conditions.

SETCURSORPOSITION

he **SetCursorPosition** procedure moves the cursor to a specific spot on the screen, based on the X and Y coordinates you give it. It takes two inputs: the X-coordinate (stored in **DL**) and the Y-coordinate (stored in **DH**).

First, the procedure checks if the coordinates are within the valid range. If either the X or Y value is out of bounds, it shows an error message and stops.

Here's how the range check works in the **SetCursorPosition** procedure:

```
1090 .IF (dl < 0) || (dl > 79)
1091 mov edx,OFFSET BadXCoordMsg
1092 call WriteString
1093 jmp quit
1094 .ENDIF
1095
1096 .IF (dh < 0) || (dh > 24)
1097 mov edx,OFFSET BadYCoordMsg
1098 call WriteString
1099 jmp quit
1100 .ENDIF
```

The .IF directive checks whether the X or Y coordinates are outside the allowed range. The **OR** operator (||) combines the two conditions. If either one is true (meaning the coordinates are out of range), an error message is displayed, and the procedure exits.

If both coordinates are valid, the procedure then calls the **Gotoxy** procedure to actually set the cursor at the correct position on the screen.

```
call Gotoxy
```

The **Gotoxy** procedure is a built-in MASM function that takes care of moving the cursor to the specified location.

This example shows how the **SetCursorPosition** procedure uses the .IF directive to check for input errors, which helps ensure the program runs smoothly without unexpected issues.

COLLEGE REGISTRATION EXAMPLE

The college registration uses the .IF, .ELSEIF, and .ENDIF directives to implement a multiway branch structure.

The structure checks the student's grade average and number of credits to determine whether or not the student can register.

The following is a simpler explanation of the code:

```
1109 .data
1110     TRUE = 1
1111     FALSE = 0
1112     gradeAverage WORD 275
1113     ; test value
1114     credits WORD 12
1115     ; test value
1116     OkToRegister BYTE ?
1117 .code
1118     mov OkToRegister,FALSE
1119
1120     ; Check if the student's grade average is greater than 350.
1121     .IF gradeAverage > 350
1122         mov OkToRegister,TRUE
1123     .ELSEIF (gradeAverage > 250) && (credits <= 16)
1124         mov OkToRegister,TRUE
1125     .ELSEIF (credits <= 12)
1126         mov OkToRegister,TRUE
1127     .ENDIF
```

The **.IF** directive checks if the student's grade average is above 350. If it is, the mov instruction sets the **OkToRegister** variable to **TRUE**, meaning the student is allowed to register.

The **.ELSEIF** directive checks the next condition: it looks at whether the student's grade average is greater than 250 **and** if the student wants to take 16 or fewer credits. If both conditions are true, it also sets the **OkToRegister** variable to **TRUE**.

There's another **.ELSEIF** that checks if the student wants to take 12 or fewer credits. If this condition is true, the **OkToRegister** variable gets set to **TRUE** as well.

If none of these conditions are met, the **OkToRegister** variable stays as **FALSE**, meaning the student can't register.

The following is a breakdown of the generated code that you sent:

```
1132 mov OkToRegister,FALSE
1133 cmp word ptr gradeAverage, 350
1134 jbe @C0006
1135 mov byte ptr OkToRegister, TRUE
1136 jmp @C0008
1137 @C0006:
1138 cmp word ptr gradeAverage, 250
1139 jbe @C0009
1140 cmp word ptr credits, 16
1141 ja @C0009
1142 byte ptr OkToRegister, TRUE
1143 mov
1144 jmp @C0008
1145 @C0009:
1146 cmp word ptr credits,12
1147 ja @C0008
1148 mov
```

- The first line of code moves the value FALSE to the OkToRegister variable.
- The next two lines of code compare the student's grade average to 350. If the grade average is greater than 350, the program jumps to the label @C0008. Otherwise, the program continues to the next line of code.
- The next three lines of code compare the student's grade average to 250 and the number of credits the student wants to take to 16. If both conditions are true, the program jumps to the label @C0008. Otherwise, the program continues to the next line of code.
- The next two lines of code compare the number of credits the student wants to take to 12. If the number of credits is less than or equal to 12, the program jumps to the label @C0008. Otherwise, the program continues to the next line of code.
- The label @C0008 is a jump target. If the program jumps to this label, the OkToRegister variable will be set to TRUE.
- The program exits at the end of the code.

CREATING LOOPS WITH .REPEAT AND .WHILE

.REPEAT Directive

The **.REPEAT** directive creates a loop that runs the instructions inside the loop **at least once**, before checking if the loop should continue. The loop keeps running until the condition specified by the **.UNTIL** directive becomes **true**.

So, the basic idea is:

1. The loop body (the instructions inside the loop) will execute first.
2. After that, the condition is checked.
3. If the condition is **false**, the loop will run again.
4. This process repeats until the condition turns **true**.

Syntax:

```
1153 .REPEAT
1154 statements
1155 .UNTIL condition
```

```
1159 mov eax, 0
1160 .REPEAT
1161     inc eax
1162     call WriteDec
1163     call Crlf
1164 .UNTIL eax == 10
```

.WHILE Directive

The **.WHILE** directive creates a loop that checks the condition **before** running the loop's instructions. If the condition is **false** right from the start, the loop body won't execute at all.

As long as the condition remains **true**, the loop keeps running, and it'll stop as soon as the condition becomes **false**.

So, the flow looks like this:

1. The condition is checked first.
2. If the condition is **true**, the loop body runs.
3. If the condition is **false**, the loop body is skipped.
4. The loop keeps going, checking the condition each time, until it finally becomes **false**.

Syntax:

```
1169 .WHILE condition
1170 statements
1171 .ENDW
```

```
1173 mov eax, 0
1174 .WHILE eax < 10
1175     inc eax
1176     call WriteDec
1177     call Crlf
1178 .ENDW
```

I. Differences Between .REPEAT and .WHILE

The main difference between the **.REPEAT** and **.WHILE** directives comes down to when the loop body is executed:

- The **.REPEAT** directive **always** runs the loop body at least once, even if the condition is false right away. This makes it useful when you want to ensure the loop runs at least once before checking the condition.
- The **.WHILE** directive, however, checks the condition **before** running the loop body. If the condition is false at the start, the loop body will be skipped entirely.

II. Which Directive to Use?

- **Use the .WHILE directive** when you need to check the condition **before** executing the loop body. It's more efficient because it won't waste resources running the loop if the condition is false from the start.
- However, there are cases where the **.REPEAT** directive is better. For example, you might use **.REPEAT** if you want to make sure some code runs at least once—maybe to initialize a variable before checking the condition.

III. Conclusion

Both the **.REPEAT** and **.WHILE** directives are essential for creating loops in MASM. By understanding when and why to use each one, you can choose the best tool for the job depending on whether you want to run the loop body at least once or only when the condition is true.

Here is a more complete explanation using the .WHILE and .IF directives:

```
.data
    X DWORD 0
    op1 DWORD 2
    ; test data
    op2 DWORD 4
    ; test data
    op3 DWORD 5
    ; test data
.code
    mov eax, op1
    mov ebx, op2
    mov ecx, op3

    .WHILE eax < ebx
        inc eax

        .IF eax == ecx
            mov X, 2
        .ELSE
            mov X, 3
        .ENDIF
    .ENDW
```

The loop starts with a **starting value** (op1) and keeps running until it reaches a **limit** (op2).

On each pass, the loop **adds 1** to the current value of op1.

Inside the loop, there's a **check**:

- If the current value equals another variable (op3), then a result variable (X) is set to **2**.
- If it doesn't match, X is set to **3**.

The loop keeps repeating this process until the starting value grows large enough to be **greater than or equal to** the limit.

Simplified View

Inputs: three numbers (op1, op2, op3).

Output: one number (X).

Process:

- Start at op1.
- Keep counting up until you reach op2.
- At each step, check if the current number equals op3.
- If yes → mark X as 2.
- If no → mark X as 3.

👉 Think of it like a **counting game**: you start at one number, keep ticking upward, and at each tick you ask, "Did I hit the special number yet?" If yes, you shout "2!" If not, you shout "3!"

Questions

Convert an ASCII digit in AL to its corresponding binary value:

```
1211 cmp al, '0'    ; Compare AL with ASCII '0'
1212 jb done        ; If AL is less than '0', it's not a valid digit
1213 cmp al, '9'    ; Compare AL with ASCII '9'
1214 ja done        ; If AL is greater than '9', it's not a valid digit
1215 sub al, '0'    ; Convert ASCII digit to binary by subtracting '0'
1216 done:
```

Calculate the parity of a 32-bit memory operand:

```
1221 xor eax, eax  ; Clear EAX (parity result)
1222 xor ebx, ebx  ; Clear EBX (loop counter)
1223 loop_start:
1224   xor al, [edi + ebx]  ; XOR AL with the next byte in memory
1225   inc ebx              ; Increment loop counter
1226   cmp ebx, 32          ; Compare loop counter with 32
1227   jl loop_start        ; If not all 32 bits processed, continue
```

Generate a bit string in EAX representing members in SetX not in SetY:

```
; Assuming SetX and SetY are two memory operands of the same size (e.g., 32 bits)
mov eax, SetX          ; Load SetX into EAX
and eax, not SetY      ; Apply NOT operation to SetY and AND with SetX
```

Jump to label L1 when DX <= CX:

Jump to label L2 when AX > CX (signed comparison):

Clear bits 0 and 1 in AL and jump based on the destination operand:

```
cmp dx, cx    ; Compare DX and CX
jbe L1        ; Jump to L1 if DX <= CX

cmp ax, cx    ; Compare AX and CX (signed comparison)
jg L2         ; Jump to L2 if AX > CX

and al, 0xFC  ; Clear bits 0 and 1 in AL
test al, al   ; Test if AL is zero
jz L3         ; Jump to L3 if AL is zero
jmp L4        ; Jump to L4 (if AL is not zero)
```

Let's start with implementing the pseudocode for the first exercise using short-circuit evaluation in assembly language. The pseudocode is as follows:

```
if( val1 > ecx ) AND ( ecx > edx )
    X = 1
else
    X = 2;
```

Here's the corresponding assembly code:

```
; Assuming val1, ecx, edx, and X are 32-bit variables
; Also, assuming the condition is checked within a function

cmp dword [val1], ecx      ; Compare val1 with ecx
jle else_condition         ; Jump to else_condition if val1 <= ecx

cmp ecx, edx               ; Compare ecx with edx
jle else_condition         ; Jump to else_condition if ecx <= edx

mov dword [X], 1           ; Set X to 1 if both conditions are met
jmp done                   ; Jump to done to skip the else block

else_condition:
mov dword [X], 2           ; Set X to 2 if conditions are not met

done:
; Rest of the code continues here
```

In this code, we first compare val1 with ecx. If val1 is less than or equal to ecx, we jump to the else_condition label, effectively skipping the X = 1 assignment.

Then, we compare ecx with edx. If ecx is less than or equal to edx, we also jump to the else_condition label.

If both conditions are met (val1 > ecx and ecx > edx), we set X to 1. Otherwise, if either condition is not met, we set X to 2.

The jmp done statement ensures that we skip the else_condition block when both conditions are met.

Exercise: Implement the following pseudocode using short-circuit evaluation:

```
1280 if( ebx > ecx ) OR ( ebx > val1 )
1281     X = 1
1282 else
1283     X = 2
```

Here's the corresponding assembly code:

```
1286 ; Assuming ebx, ecx, val1, and X are 32-bit variables
1287 ; Also, assuming the condition is checked within a function
1288
1289 cmp ebx, ecx                ; Compare ebx with ecx
1290 jg set_X_to_1               ; Jump to set_X_to_1 if ebx > ecx
1291
1292 cmp ebx, val1               ; Compare ebx with val1
1293 jg set_X_to_1               ; Jump to set_X_to_1 if ebx > val1
1294
1295 ; If neither condition is met, set X to 2 and continue
1296 mov dword [X], 2
1297 jmp done
1298
1299 set_X_to_1:
1300 mov dword [X], 1          ; Set X to 1 if either condition is met
1301
1302 done:
1303 ; Rest of the code continues here
```

In this code, we first compare ebx with ecx. If ebx is greater than ecx, we jump to the set_X_to_1 label, effectively setting X to 1.

Next, we compare ebx with val1. If ebx is greater than val1, we again jump to set_X_to_1, ensuring that X is set to 1 if either condition is met.

If neither condition is met, we set X to 2 and continue with the code.

Implement the following pseudocode using short-circuit evaluation:

```
if( ebx > ecx AND ebx > edx) OR ( edx > eax )
    X = 1
else
    X = 2
```

Here's the corresponding assembly code:

```
 ; Assuming ebx, ecx, edx, eax, and X are 32-bit variables
; Also, assuming the condition is checked within a function

; Check the first condition: ebx > ecx AND ebx > edx
cmp ebx, ecx               ; Compare ebx with ecx
jle check_second_condition ; If ebx <= ecx, skip the second condition

cmp ebx, edx               ; Compare ebx with edx
jle check_second_condition ; If ebx <= edx, skip the second condition

; If we reach here, both conditions are met, so set X to 1
mov dword [X], 1
jmp done

check_second_condition:
; Check the second condition: edx > eax
cmp edx, eax               ; Compare edx with eax
jle set_X_to_2             ; If edx <= eax, set X to 2 and skip to done

; If we reach here, the second condition is met, so set X to 1
mov dword [X], 1
jmp done

set_X_to_2:
; If neither condition is met, set X to 2
mov dword [X], 2

done:
; Rest of the code continues here
```

In this code, we first check the first condition: ebx > ecx AND ebx > edx. If either of these subconditions is not met, we skip to check_second_condition.

If both subconditions are met, we set X to 1 and jump to done.

In check_second_condition, we check the second condition: edx > eax. If this condition is met, we set X to 1 and jump to done. If the second condition is not met, we set X to 2.

This code implements the pseudocode with short-circuit evaluation as requested.

Implement the following pseudocode using short-circuit evaluation:

```
while N > 0
    if N != 3 AND (N < A OR N > B)
        N = N - 2
    else
        N = N - 1
```

Here's the corresponding assembly code:

```
; Assuming N, A, B are 32-bit signed integers
; Also, assuming this code is part of a larger program
while_loop:
    cmp dword [N], 0      ; Compare N with 0
    jle end_while         ; If N <= 0, exit the loop
    cmp dword [N], 3      ; Compare N with 3
    je skip_decrement     ; If N == 3, skip to else (decrement by 1)
    ; Check the second condition: N < A OR N > B
    cmp dword [N], [A]    ; Compare N with A
    jge else_decrement     ; If N >= A, skip to else (decrement by 1)
    cmp dword [N], [B]    ; Compare N with B
    jle else_decrement     ; If N <= B, skip to else (decrement by 1)
    ; If neither condition is met, decrement by 2
    sub dword [N], 2
    jmp continue_while    ; Continue the loop
skip_decrement:
    ; N == 3, decrement by 1
    sub dword [N], 1
    jmp continue_while
else_decrement:
    ; If any condition is met, decrement by 1
    sub dword [N], 1
continue_while:
    ; Loop back to the beginning of the while loop
    jmp while_loop
end_while:
; Rest of the code continues here
```

In this code, we use a while loop to repeatedly check the conditions and decrement N accordingly. The loop continues as long as N is greater than 0.

If N is equal to 3, we decrement it by 1 (skip to the else part). If N is not equal to 3 and either N is greater than A or N is less than B, we decrement N by 2.

Otherwise, we decrement N by 1. The loop continues until N becomes less than or equal to 0.

Labels in Assembly Loops (Plain Talk)

- **continue_while** → Think of this as the "go back to the top" marker. If none of the special conditions apply, the program jumps here to keep looping.
- **end_while** → This is the "exit point." When the loop condition fails (like when N <= 0), the program jumps here to break out of the loop and move on.
- **else_decrement** → This is the "fallback action." If the fancy condition (N != 3 AND (N < A OR N > B)) isn't true, the program comes here to just subtract 1 from N instead of 2.

Big Picture

These labels aren't functions you call — they're **signposts** inside the loop.

The CPU uses jumps (jmp) to move between them depending on what's happening with your variables.

It's just a way to organize the flow so the loop knows where to continue, where to exit, and what to do in the "otherwise" case.

👉 Think of it like a board game:

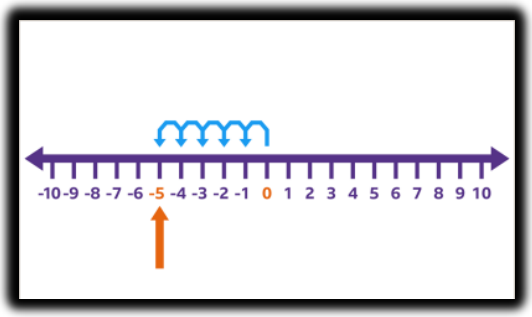
- **continue_while** = "Go back to Start."
- **end_while** = "Game Over."
- **else_decrement** = "Take the default move."

FINAL QUESTIONS FOR THIS TOPIC ON CONDITIONAL PROCESSING

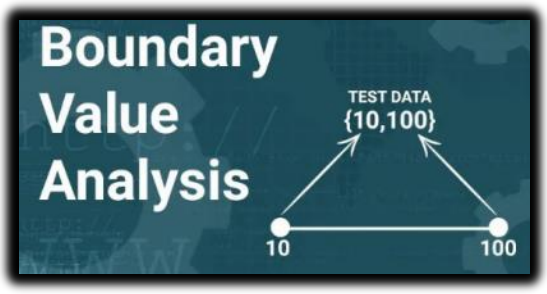
Testing Tips for Your Code

Use a Debugger: Debugging is key to finding issues in your code. A debugger lets you step through your code, check variables, and pinpoint where things go wrong. Most dev environments, like Visual Studio, come with one built in.

Test with Negative Values: If your code works with signed data, make sure to test with negative values. This helps cover all possible cases and ensures your code handles them well.

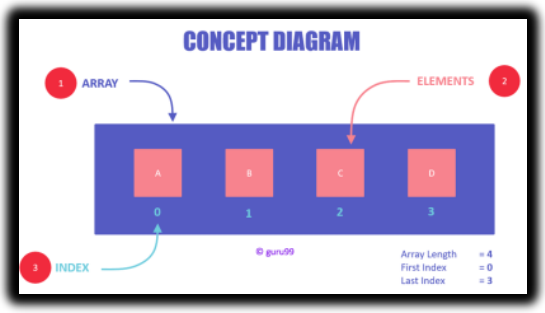


Test Boundaries: If you're working with a range of values, test with inputs just before, on, and after the boundaries. This ensures your code handles edge cases properly.



Multiple Test Cases: Don't just run one test case. Try a few different scenarios to make sure your code can handle a variety of inputs and conditions.

Debugging Array Operations: If you're working with arrays, the debugger's Memory window is super helpful. It shows you the array's contents, so you can check how your code is modifying the array, either in hexadecimal or decimal.



Check Register Preservation: If a procedure changes registers, call it twice in a row. This helps make sure the procedure properly preserves register values between calls.

Passing Multiple Arrays: When passing multiple arrays to a procedure, avoid using array names directly inside the procedure. Instead, set up registers (like ESI or EDI) to point to the arrays, and use indirect addressing (like [esi]) inside the procedure.

Local Variables in Procedures: For variables that only need to exist inside a procedure, declare them using the .data directive before the procedure, and the .code directive after. This ensures they're correctly initialized every time the procedure is called.

Testing at Boundaries: When a range of input values is specified, test your code with values that fall before, on, and after these boundaries. This helps verify how your code handles edge cases.

Exercise: Filling an Array

This exercise requires you to create a procedure that fills an array of doublewords with N random integers within the range [j, k]. You need to pass a pointer to the array, the value of N, and the values of j and k as parameters to the procedure. Additionally, you should preserve all register values between calls to the procedure.

Here's a sample assembly code for this exercise:

```
.data
array DWORD 10 DUP (?) ; Define an array to hold the random integers
.code
FillArray PROC
    ; Parameters:
    ;   edi = pointer to the array
    ;   ecx = N (number of elements)
    ;   ebx = j (lower bound)
    ;   edx = k (upper bound) || you can initialize random number generator (optional)
    call InitializeRandom
    ; Loop to fill the array with random numbers
    fill_loop:
        mov eax, ebx         ; Load lower bound (j) into eax
        sub eax, 1           ; Subtract 1 to make j inclusive
        add eax, edx         ; Calculate the range (k - j + 1)
        call GetRandom       ; Get a random number in [0, range)
        add eax, ebx         ; Add j to the random number to fit [j, k]
        mov [edi], eax       ; Store the random number in the array
        add edi, 4           ; Move to the next element
        loop fill_loop        ; Repeat for N elements
    ret
FillArray ENDP
main:
    ; Usage example:
    mov edi, OFFSET array ; Pointer to the array
    mov ecx, 10           ; N = 10 elements
    mov ebx, 1            ; Lower bound (j)
    mov edx, 100          ; Upper bound (k)
    call FillArray
    ; Call FillArray again with different j and k values if needed,Verify the results using a debugger
    ;(you can inspect the contents of the 'array' variable), and the rest of the program
```

This code defines a procedure called FillArray, which fills an array with random integers within the specified range. The main program demonstrates how to use this procedure with different values of j and k.

Exercise: Summing an Array

This exercise requires you to create a procedure that returns the sum of all array elements within the range [j, k]. You'll pass a pointer to the array, the size of the array, and the values of j and k as parameters to the procedure. The sum should be returned in the EAX register, and all other register values should be preserved between calls.

Here's a sample assembly code for this exercise:

```
.data
    array SDWORD 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ; Example array of signed doublewords
.code
SumInRange PROC
    ; Parameters:
    ;   edi = pointer to the array
    ;   ecx = size of the array
    ;   ebx = j (lower bound)
    ;   edx = k (upper bound)
    xor eax, eax             ; Clear EAX to store the sum
    sum_loop:
        mov esi, [edi]       ; Load the next element into ESI
        cmp esi, ebx          ; Compare with lower bound (j)
        jl not_in_range       ; Jump if less than j
        cmp esi, edx          ; Compare with upper bound (k)
        jg not_in_range       ; Jump if greater than k
        add eax, esi          ; Add to the sum
    not_in_range:
        add edi, 4            ; Move to the next element
        loop sum_loop         ; Repeat for all elements
    ret
SumInRange ENDP
main:                        ; Usage example:
    mov edi, OFFSET array    ; Pointer to the array
    mov ecx, 10              ; Size of the array
    mov ebx, 2               ; Lower bound (j)
    mov edx, 7               ; Upper bound (k)
    call SumInRange
    ; The sum will be in the EAX register
    ; Call SumInRange again with different j and k values if needed
    ; Rest of the program
```

This code defines a procedure called SumInRange, which calculates the sum of array elements within the specified range [j, k]. The main program demonstrates how to use this procedure with different values of j and k.

Exercise: TestScore Evaluation

This exercise requires you to create a procedure named CalcGrade that receives an integer value between 0 and 100 and returns a single capital letter grade in the AL register. The grade returned should be based on specified ranges.

Here's a sample assembly code for this exercise:

```
.data
    grade CHAR ? ; Variable to store the grade
.code
CalcGrade PROC
    ; Parameter:
    ;   eax = integer value between 0 and 100
    cmp eax, 0
    jl invalid_input     ; Input is less than 0, return 'F'
    cmp eax, 60
    jl grade_F           ; Input is less than 60, return 'F'
    cmp eax, 70
    jl grade_D           ; Input is less than 70, return 'D'
    cmp eax, 80
    jl grade_C           ; Input is less than 80, return 'C'
    cmp eax, 90
    jl grade_B           ; Input is less than 90, return 'B'
    grade_A:
        mov al, 'A'       ; Input is 90 or greater, return 'A'
        jmp done
    grade_B:
        mov al, 'B'       ; Input is between 80 and 89, return 'B'
        jmp done
    grade_C:
        mov al, 'C'       ; Input is between 70 and 79, return 'C'
        jmp done
    grade_D:
        mov al, 'D'       ; Input is between 60 and 69, return 'D'
        jmp done
    grade_F:
        mov al, 'F'       ; Input is between 0 and 59, return 'F'

    invalid_input:
         mov al, '?'       ; Invalid input, return '?'

    done:
        ret
CalcGrade ENDP

main:
    ; Usage example:
    mov eax, 85           ; Input value (test score)
    call CalcGrade

    ; The grade will be in the AL register

    ; Rest of the program
```

This code defines a procedure called CalcGrade, which returns a grade based on the specified ranges. The main program demonstrates how to use this procedure by passing a test score (integer value) and receiving the corresponding grade in the AL register.

Questions for you…

Exercise 4: Test Score Evaluation

Create a program that generates 10 random integers between 50 and 100 (inclusive). For each integer generated, pass it to the CalcGrade procedure, which will return a corresponding letter grade based on specified ranges. Display the integer and its corresponding letter grade. You can use the RandomRange procedure from the Irvine32 library to generate random integers.

Exercise 5: Boolean Calculator (1)

Create a program that acts as a simple boolean calculator for 32-bit integers. It displays a menu with options to perform logical operations (AND, OR, NOT, XOR) and allows the user to choose an operation. Implement this menu using Table-Driven Selection. When the user selects an operation, call a procedure to display the operation name. Implement this menu-driven program.

Exercise 6: Boolean Calculator (2)

Continuing from Exercise 5, implement procedures for each of the logical operations (AND, OR, NOT, XOR). Prompt the user for inputs (hexadecimal integers) as required by the chosen operation, perform the operation, and display the result in hexadecimal.

Exercise 7: Probabilities and Colors

Write a program that randomly selects one of three colors (white, blue, green) with specific probabilities (30%, 10%, 60%). Use a loop to display 20 lines of text, each with a randomly chosen color based on the given probabilities. You can generate a random integer between 0 and 9 and use it to select colors accordingly.

Exercise 8: Message Encryption

Revise an encryption program to encrypt and decrypt a message using an encryption key consisting of multiple characters. Implement encryption and decryption by XOR-ing each character of the key against a corresponding byte in the message. Repeat the key as necessary until all plaintext bytes are translated.

Plain text	T	h	i	s		i	s		a		P	l	a	i	n	t	e	x	t		m	e	s	s	a	g	e	(etc.)
Key	A	B	X	m	v	#	7	A	B	X	m	v	#	7	A	B	X	m	v	#	7	A	8	X	m	v	#	7	

(The key repeats until it equals the length of the plain text...)

Exercise 9: Validating a PIN

Create a procedure called Validate_PIN that checks the validity of a 5-digit PIN based on specified digit ranges. The procedure receives a pointer to an array containing the PIN and validates each digit. If any digit is outside its valid range, return the digit's position (1 to 5) in the EAX register; otherwise, return 0. Write a test program that calls Validate_PIN with valid and invalid PINs and verifies the return values.

Digit Number	Range
1	5 to 9
2	2 to 5
3	4 to 8
4	1 to 4
5	3 to 6

Exercise 10: Parity Checking

Implement a procedure that checks the parity (even or odd) of bytes in an array. The procedure returns True (1) in EAX if the bytes have even parity and False (0) if they have odd parity. Write a test program that calls the procedure with arrays having even and odd parity and verifies the return values.


You know what?
Imma finally do it!
Imma learn assembly!
assembly helio world
All
Images
Videos
News
About 67,300,000 results (0.49 seconds)
I know it's low-level,
but how bad can it be?
click
https://montcs.bloomu.edu › Assembly › hello-asm
"Hello, World" in x86 Assembly Language
"Hello, World" In x8S Assembly Language - A minimal-size version - Individual-character output along with string output - DOS2 length-delimited output - a Linux- ...
People also ask
How do you say hello world in assembly language?
Is assembly still used?
What are the examples of assembly languags?
; hello-DOS.asm - single-segment, 16-bit "hello world" program
;
; assemble w
-o hi.com hello-DOS.asm"
org 0x100
ways start 256 bytes into the segment
; int 21h is
mov dx, msg
es of or message in dx
mov ah, 9
nt string" sub-function
int 0x21
mov ah, 0x
rogram " sub
int 0x21
ces
msg db
0d, 0x0a,
ssage
You know what?
Imma finally do it!
Imma play Elden Ring!