

Contents

AND OPERATOR	2
BIT-MAPPED SETS	8
OR OPERATION.....	12
XOR OPERATION.....	16
NOT OPERATION.....	19
TEST OPERATION.....	19
CMP INSTRUCTION	23
SETTING AND CLEARING FLAGS	24
BOOLEANS AND 64-BIT MODE	26
CONDITIONAL JUMPS	29
CONDITIONAL LOOPS	46
CONDITIONAL STRUCTURES.....	52
WHITEBOX TESTING	56
SHORT CIRCUIT EVALUATION(AND)	62
SHORT CIRCUIT EVALUATION(OR)	64
WHILE LOOPS.....	66
IF STATEMENTS IN ASSEMBLY.....	72
TABLE DRIVEN SELECTION.....	76
FINITE STATE MACHINES.....	86
CONDITIONAL CONTROL FLOW DIRECTIVES	101
SIGNED AND UNSIGNED IN ASSEMBLY CODE	106
COMPARING REGISTERS	107
COMPOUND EXPRESSIONS	108
CREATING LOOPS WITH .REPEAT AND .WHILE	112
FINAL QUESTIONS FOR THIS TOPIC ON CONDITIONAL PROCESSING	121

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:

```

01 ; AND two registers
02 mov eax, 10h
03 mov ebx, 5h
04 and eax, ebx ; eax = 4h
05
06 ; OR two registers
07 mov eax, 10h
08 mov ebx, 5h
09 or eax, ebx ; eax = 15h
10
11 ; XOR two registers
12 mov eax, 10h
13 mov ebx, 5h
14 xor eax, ebx ; eax = 11h
15
16 ; NOT a register
17 mov eax, 10h
18 not eax ; eax = 11111110h
19
20 ; TEST two registers
21 mov eax, 10h
22 mov ebx, 5h
23 test eax, ebx ; CF flag is set to 0
24
25 ; TEST a register against a value
26 mov eax, 10h
27 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: $x \text{ AND } y$, where x and y are bits.

x	y	$x \wedge y$
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.

```
119 .data
120 SetX DWORD 10000000h ; SetX represented as a 32-bit value
121
122 .code
123 main PROC
124     mov eax, SetX      ; Load SetX into EAX
125     and eax, 00000010b ; Check if element[4] is a member
126     cmp eax, 0          ; Compare the result with 0
127     jz not_a_member    ; Jump if Zero flag is set (not a member)
128     ; If Zero flag is not set, element[4] is a member of SetX
129     ; Your code here for member case
130     jmp done
131
132 not_a_member:
133     ; Your code here for not a member case
134
135 done:
136     ; Exit your program
137     invoke ExitProcess, 0
138
139 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.

```
145 .data
146 SetX DWORD 10000000h ; SetX represented as a 32-bit value
147
148 .code
149 main PROC
150     mov eax, SetX      ; Load SetX into EAX
151     not eax          ; Complement SetX
152     ; EAX now contains the complement of SetX
153     ; Your code here to work with the complemented set
154     ; Exit your program
155     invoke ExitProcess, 0
156
157 main ENDP
158
```

♥ Set Intersection

Use AND to find elements common to both sets.

```
162 .data
163 SetX DWORD 10000000000000000000000000000111h
164 SetY DWORD 100000101010000000011101100011h
165
166 .code
167 main PROC
168     mov eax, SetX      ; Load SetX into EAX
169     and eax, SetY      ; Calculate the intersection of SetX and SetY
170     ; EAX now contains the intersection
171     ; Your code here to work with the intersection
172     ; Exit your program
173     invoke ExitProcess, 0
174
175 main ENDP
```

Set Union

Use OR to combine elements from both sets.

Example of SetX and SetY:

```
.data
SetX DWORD 10000000000000000000000000000000111h
SetY DWORD 1000001010100000000011101100011h

.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: $x \text{ OR } y$, where x and y are bits.

x	y	$x \vee y$
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

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

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:  
10000000000000000000000000000000111
```

```
SetY:  
100000101010000000011101100011
```

```
Union (SetX OR SetY):  
100000101010000000011101100111
```

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 ($x \oplus y$)

x	y	Result ($x \oplus y$)
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.)

```
251 Input value: 0 0 1 0 0 1 0 1
252 Test value: 0 0 0 0 1 0 0 1
253 Result: 0 0 0 0 0 0 0 1
254 Zero Flag (ZF) = 0 (Not set)
255
256
257 Input value: 0 0 1 0 0 1 0 0
258 Test value: 0 0 0 0 1 0 0 1
259 Result: 0 0 0 0 0 0 0 0
260 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 = FFFFFFFFFFFFFF  
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 = FFFFFFFFFFFFFF  
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.

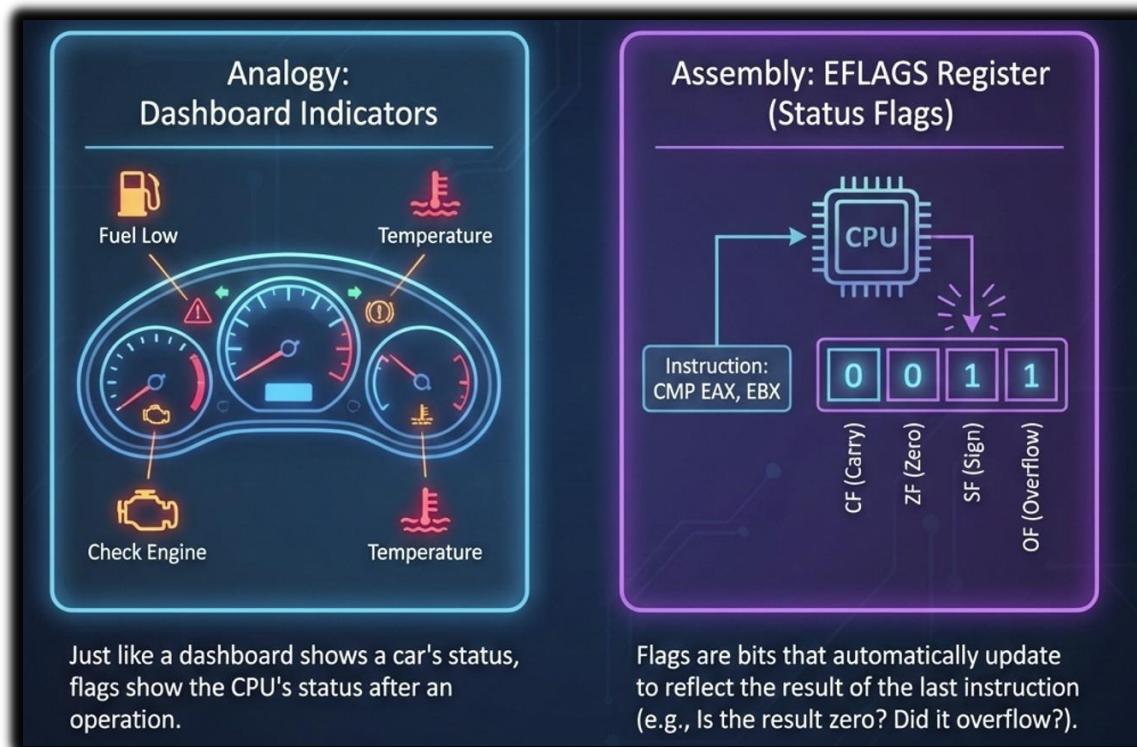
So, in simple terms, this code checks if the value in the EAX register is zero. If it is, it jumps to L1. If not, it continues executing the code below the jz instruction.

```
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 $leftOp \geq rightOp$)
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 $leftOp \leq rightOp$)
JNA	Jump if not above (same as JBE)

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

Signed Comparisons (Table Below)

Mnemonic	Description
JG	Jump if greater (if $leftOp > rightOp$)
JNLE	Jump if not less than or equal (same as JG)
JGE	Jump if greater than or equal (if $leftOp \geq rightOp$)
JNL	Jump if not less (same as JGE)
JL	Jump if less (if $leftOp < rightOp$)
JNGE	Jump if not greater than or equal (same as JL)
JLE	Jump if less than or equal (if $leftOp \leq rightOp$)
JNG	Jump if not greater (same as JLE)

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

Example 1:

```
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.

1. Conditional Jump Applications:

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

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

In your provided example:

```
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
```

Here's a breakdown of what's happening:

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

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

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

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

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

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

1. 2.

2. Larger of Two Integers:

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

Certainly, let's delve deeper into the code snippet that compares two unsigned integers (EAX and EBX) and moves the larger value to EDX. This is a great example of conditional branching based on integer comparisons in assembly language:

```
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
```

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

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

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

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

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

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

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

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

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

1. 3.

3. Smallest of Three Integers:

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

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

```
104 .data
105     V1 WORD ?
106     V2 WORD ?
107     V3 WORD ?
108
109 .code
110     mov ax, V1      ; Assume V1 is the smallest
111     cmp ax, V2      ; Compare AX and V2
112     jbe L1          ; Jump to L1 if AX is less than or equal to V2
113     mov ax, V2      ; Move V2 to AX (V1 is not the smallest)
114     L1:
115     cmp ax, V3      ; Compare AX and V3
116     jbe L2          ; Jump to L2 if AX is less than or equal to V3
117     mov ax, V3      ; Move V3 to AX (V2 or V1 is not the smallest)
118     L2:
119     ; AX now contains the smallest integer among V1, V2, and V3
```

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

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

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

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

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

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

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

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

1. 4.

4. Loop until Key Pressed:

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

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

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

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

```
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
```

Here's how this code works step by step:

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

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

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

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

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

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

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

```

135 ; Scanning an Array (ArrayScan.asm)
136 ; Scan an array for the first nonzero value.
137 INCLUDE Irvine32.inc
138 .data
139     intArray SWORD 0,0,0,0,1,20,35,-12,66,4,0
140     noneMsg BYTE "A non-zero value was not found",0
141 .code
142     main PROC
143         ; Initialize registers and variables
144         mov esi, 0           ; Index for array traversal
145         mov ecx, LENGTHOF intArray ; Length of the array
146         mov ebx, ADDR intArray    ; Address of the array
147         mov al, 0            ; Clear AL register to store the result (found or not)
148     searchLoop:
149         cmp word ptr [ebx + esi * 2], 0 ; Compare the current element with zero
150         jnz foundNonZero      ; Jump if not zero
151         inc esi              ; Increment index
152         loop searchLoop       ; Continue loop until ecx is zero
153         mov al, 1            ; Set AL to 1 if no nonzero value found
154         jmp done
155     foundNonZero:
156         mov al, 0            ; Set AL to 0 if a nonzero value is found
157     done:
158         ; Display appropriate message based on AL value
159         cmp al, 0
160         je noNonZeroFound
161         mov edx, OFFSET noneMsg
162         call WriteString
163         jmp endProgram

164     noNonZeroFound:
165         ; Display the first nonzero value found
166         mov edx, [ebx + esi * 2]
167         call WriteInt
168
169     endProgram:
170         call Crlf
171         exit
172     main ENDP
173
174 END main

```

Explanation:

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

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

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

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

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

The program then ends by calling Crlf and exiting.

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

=====

Encryption Program Overview

=====

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

- • **User Input:** The user enters a plain text message.
- • **Encryption:** The program encrypts the plain text by XORing each character with a single character key and displays the cipher text.
- • **Decryption:** It then decrypts the cipher text using the same key and displays the original plain text.

```
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 and LOOPE Instructions

The **LOOPZ (Loop if Zero)** and **LOOPE (Loop if Equal)** instructions are conditional loop instructions used in assembly language programming.

They both share the same opcode and have identical behavior based on the condition of the Zero Flag (ZF). Here's a simplified explanation:

ECX (or RCX in 64-bit mode) is the loop counter register. ECX is decremented by 1 in each iteration of the loop.

The **loop continues if ECX is greater than 0 and the Zero Flag (ZF) is set** (indicating the result of the previous operation was zero).

If the condition is met, the program jumps to the specified destination label. If the condition is not met (ECX not greater than 0 or ZF not set), the loop exits, and control proceeds to the next instruction.

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

```
256 ; sum the elements of the array `array`
257 mov ecx, array_size
258 loopz sum_loop
259
260 ; sum_loop:
261 add eax, [array + ecx - 1]
262 dec ecx
263 jnz sum_loop
```

In this example, the loop counter register ECX is initialized with the size of the array.

The LOOPZ instruction then decrements ECX and adds the element at array + ECX - 1 to the accumulator register EAX. If ECX is still greater than 0 and the Zero flag is set, the loop will jump back to the sum_loop label.

Otherwise, execution will fall through to the next instruction, which will be the end of the loop.

LOOPZ (Loop if Zero): This instruction is used to create a loop with an additional condition. The condition is that the Zero Flag (ZF) must be set for the loop to continue. Here's the syntax:

LOOPZ destination

These instructions do not affect any other status flags. LOOPE is essentially the same as LOOPZ, and they **share the same opcode**. They both have the same behavior and conditions as described above.

These instructions are often used for implementing loops in assembly language, where you want to repeat a block of code a specific number of times while a certain condition is met (in this case, the Zero Flag being set). Loop a specific number of times based on the value in the loop counter (ECX) and a condition (Zero Flag set).

The LOOPZ and LOOPE instructions can be used in a variety of other ways as well. For example, they can be used to implement nested loops, to search for a value in an array, or to reverse a string.

LOOPNZ (Loop if Not Zero)

LOOPNE (Loop if Not Equal) Instructions

The **LOOPNZ (Loop if Not Zero)** and **LOOPNE (Loop if Not Equal)** instructions are used in assembly language programming 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:

```
270 .data
271     array SWORD -3,-6,-1,-10,10,30,40,4
272     sentinel SWORD 0
273 .code
274     mov esi, OFFSET array
275     mov ecx, LENGTHOF array
276     L1:
277     test WORD PTR [esi], 8000h ; Test sign bit
278     pushfd
279     add esi, TYPE array
280     popfd
281     loopnz L1
282     jnz quit
283     sub esi, TYPE array
284     quit:
```

This code iterates through the array, testing the sign bit of each element, and continues the loop until a nonnegative value is found. If none is found, it stops when ECX becomes zero and jumps to the quit label, where ESI points to the sentinel value (0) located after the array.

Now, let's break down the code inside the .code section:

mov esi, OFFSET array: This instruction initializes the ESI register with the memory address of the array variable, effectively pointing to the beginning of the array.

mov ecx, LENGTHOF array: It loads the ECX register with the length of the array, which is the number of elements in the array.

L1:: This is a label for the beginning of a loop.

test WORD PTR [esi], 8000h: The test instruction checks the sign bit of the current array element by bitwise ANDing it with 8000h (hexadecimal representation of a signed word with the sign bit set).

pushfd: It pushes the processor flags onto the stack. This is done to save the state of the Zero Flag (ZF) because the add instruction that follows modifies the flags.

add esi, TYPE array: ESI is incremented by the size of a single array element (TYPE array), effectively moving to the next position in the array.

popfd: The flags saved by pushfd are popped from the stack, restoring the previous state.

loopnz L1: The LOOPNZ instruction decrements ECX by 1 and checks if ECX > 0 and ZF = 0. If both conditions are met, it jumps to the L1 label, continuing the loop. Otherwise, if ECX becomes zero or ZF is set, the loop terminates.

jnz quit: If the loop completes without finding a nonnegative value, it jumps to the quit label.

sub esi, TYPE array: If a nonnegative value is found, ESI is left pointing to that value.

The code efficiently iterates through the array, testing each element's sign bit. If it finds a nonnegative value, ESI points to that value; otherwise, it points to the sentinel value (0) after the array. This logic allows you to handle different cases depending on the outcome of the loop.

(True/False): The LOOPE instruction jumps to a label when (and only when) the Zero flag is clear.

True. The LOOPE (Loop If Equal) instruction jumps to a label when the Zero flag (ZF) is clear and the ECX register is greater than zero.

(True/False): In 32-bit mode, the LOOPNZ instruction jumps to a label when ECX is greater than zero and the Zero flag is clear.

True. The LOOPNZ (Loop If Not Zero) instruction in 32-bit mode jumps to a label when the ECX register is greater than zero and the Zero flag (ZF) is clear.

(True/False): The destination label of a LOOPZ instruction must be no farther than ±128 or ±127 bytes from the instruction immediately following LOOPZ.

False. The destination label of a LOOPZ instruction must be no farther than ±128 bytes from the instruction immediately following LOOPZ. This is because the offset for a short jump (like the one used by LOOPZ) is limited to a signed 8-bit value, which covers a range of -128 to +127 bytes. Modify the LOOPNZ example in Section 6.4.2 so that it scans for the first negative value in the array. Change the array initializers so they begin with positive values.

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:

```

288 int values[] = { 1, 2, 3, -4, 5, 6 };
289 int array_length = sizeof(values) / sizeof(values[0]);
290 int found_negative = 0;
291
292 __asm {
293     mov ecx, array_length
294     mov esi, 0 ; Index for accessing the array
295
296     start_loop:
297         cmp [values + esi * 4], 0 ; Compare the current element with zero
298         jns not_negative ; Jump if not negative
299         ; Code to handle negative value goes here (set found_negative flag, etc.)
300         jmp loop_end
301
302     not_negative:
303         inc esi ; Increment the array index
304         loop start_loop
305
306     loop_end:
307 }

```

Challenge: The LOOPNZ example in Section 6.4.2 relies on a sentinel value to handle the possibility that a positive value might not be found. What might happen if you removed the sentinel?

If you remove the sentinel value from the LOOPNZ example, it means that there is no clear indicator to stop the loop when a positive value is not found in the array. Without a sentinel value, the loop would continue running indefinitely, potentially causing an infinite loop in your program. To avoid this, it's essential to have some mechanism, like a sentinel value or a counter, to terminate the loop when a specific condition is not met.

Certainly, 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:

```

312 int values[] = { -1, -2, -3, -4, -5 }; // Array with only negative values
313 int array_length = sizeof(values) / sizeof(values[0]);
314 int iterations = 0; // Counter for loop iterations
315
316 __asm {
317     mov ecx, array_length
318     mov esi, 0 ; Index for accessing the array
319
320     start_loop:
321         cmp [values + esi * 4], 0 ; Compare the current element with zero
322         jns not_negative ; Jump if not negative
323         ; Code to handle negative value goes here (set found_negative flag, etc.)
324         jmp loop_end
325
326     not_negative:
327         inc esi ; Increment the array index
328         loop_start:
329             inc iterations
330             cmp iterations, array_length ; Check if we have iterated through the entire array
331             jge loop_end ; If iterations >= array_length, exit the loop
332             jmp start_loop ; Continue looping
333
334     loop_end:
335 }

```

In this code, we've introduced an iterations counter, which increments with each iteration of the loop. If iterations becomes greater than or equal to array_length, it means we've iterated through the entire array without finding a positive value, and we exit the loop. This prevents an infinite loop from occurring when there are no positive values in the array.

CONDITIONAL STRUCTURES

Conditional structures in programming allow you to make decisions based on conditions. You've likely used these in high-level languages like C++. These structures involve evaluating a condition and executing different sets of instructions based on whether the condition is true or false.

Block-Structured IF Statements: In the context of C++ and similar languages, an IF statement consists of a boolean expression followed by two sets of statements. One set of statements executes when the expression is true, and the other set when the expression is false. The "else" part is optional, meaning you can have just an "if" without an "else."

Translation to Assembly: When you translate high-level code into assembly language, you need to perform two main steps:

Evaluate the Boolean Expression: This step involves comparing values in registers or memory locations in such a way that it affects a CPU status flag. In your example, they use the CMP instruction to compare op1 and op2. This sets flags in the CPU.

Conditional Jumps: Based on the CPU status flags set by the comparison, you decide which branch of the code to execute. In your example, they use JNE (Jump if Not Equal) to skip ahead if op1 is not equal to op2.

Code Example: The code example you provided in C++ checks if op1 is equal to op2. If they are equal, it sets X to 1 and Y to 2.

Now, let's look at the assembly code they provided:

```
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:
```

mov eax, op1: This line moves the value of op1 into the EAX register for comparison.

cmp eax, op2: It then compares the values in EAX (which now holds op1) and op2.

jne L1: If the comparison shows that op1 is not equal to op2, it jumps to the label L1.

mov X, 1 and mov Y, 2: Inside L1, it assigns 1 to X and 2 to Y. This code allows for efficient execution by letting the code "fall through" when the boolean condition is true. It avoids unnecessary jumps, making it more compact.

The alternative implementation using JE (Jump if Equal) would be slightly less compact but functionally equivalent.

..it jumps to the label L1. why not say it jumps to the procedure L1?

ChatGPT

You're correct; the term "procedure" is often used in programming to refer to a block of code that has a specific purpose or function. In the context of assembly language, especially in the example you provided, "L1" is more commonly referred to as a "label."

In assembly language, labels are used to mark specific locations in the code, typically for branching purposes. So, when I mentioned "jumps to the label L1," it means that the program's execution flow will transfer to the point in the code marked by the label "L1."

While the term "procedure" is indeed used in programming, it's usually associated with higher-level languages like C++ or C# and refers to a named block of code that can be called or invoked with specific parameters. In assembly language, labels are more commonly used to mark positions in the code for control flow.

So, you could also say "jumps to the procedure L1" if you were discussing this in a higher-level language context, but in the context of assembly language, "jumps to the label L1" is the more accurate terminology.

In high-level programming languages, labels and procedures are distinct concepts:

- **Labels:** Labels are not typically used in high-level languages like C++, C#, Python, etc. In these languages, you don't typically define labels explicitly. Instead, control flow is managed using constructs like functions, loops, and conditional statements. Labels are more commonly associated with assembly language or lower-level languages.
- **Procedures (Functions):** Procedures and functions are an integral part of high-level programming languages. They are named blocks of code that can be called or invoked with specific parameters. Functions in high-level languages serve various purposes, such as performing a specific task, returning values, or modifying data. They encapsulate a set of instructions and can be reused throughout the program.

So, in summary:

- In high-level languages, you use functions (or procedures) to encapsulate code with a specific purpose and make your code more modular and readable.
 - Labels are more commonly associated with low-level languages like assembly, where they are used as markers for control flow and branching.
 - The terminology and usage can vary between different programming languages, but this distinction is generally applicable.
-

We want to implement these conditional structures in real life:

Example 2: Setting Cluster Size Based on Volume Capacity

In this example, the pseudocode is used to set the cluster size of a disk volume in the NTFS file system. The cluster size depends on the volume's capacity, specifically, whether it's less than 16 terabytes.

Here's the assembly language implementation:

```
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:
```

mov clusterSize, 8192: Initially, the code assumes a larger cluster size of 8192.

cmp terrabytes, 16: It compares the variable terrabytes with 16.

jae next: If terrabytes is greater than or equal to 16, it jumps to the next label.

mov clusterSize, 4096: Inside the next label, it sets clusterSize to 4096, indicating a smaller cluster size.

The code effectively changes the cluster size based on the volume size, as described in the pseudocode.

Example 3: Conditional Routine Calls

In this example, the pseudocode involves calling different routines based on a condition, specifically, whether op1 is greater than op2.

Here's the assembly language implementation:

```
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:
```

mov eax, op1: It moves the value of op1 into the EAX register.

cmp eax, op2: It compares the values in EAX (which now holds op1) and op2.

jg A1: If op1 is greater than op2, it jumps to A1, which calls Routine1.

call Routine2: If the comparison is false, it calls Routine2.

jmp A2: After executing either Routine1 or Routine2, it jumps to A2, which marks the exit point for the IF statement.

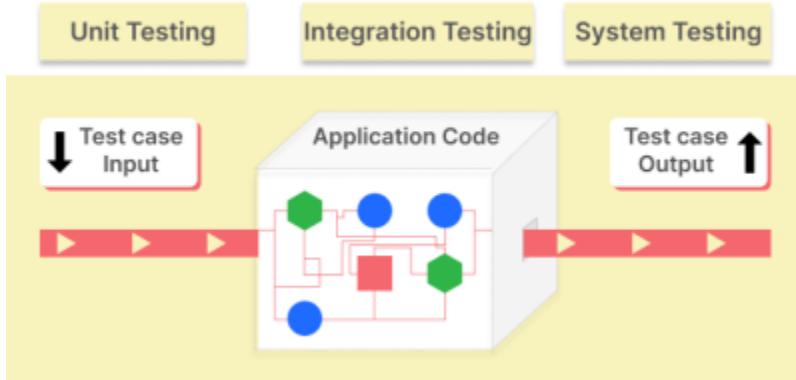
The code effectively calls different routines based on the condition provided in the pseudocode.

WHITEBOX TESTING

White box testing, also known as clear box testing, glass box testing, transparent box testing, or structural testing, is a software testing method that examines the internal structure, design, and coding of an application to verify input-output flow and improve design, usability, and security. It is one of two parts of the box testing approach to software testing, the other being **black box testing**.

White box testing involves testing the internal logic and execution paths of a subroutine by examining the source code.

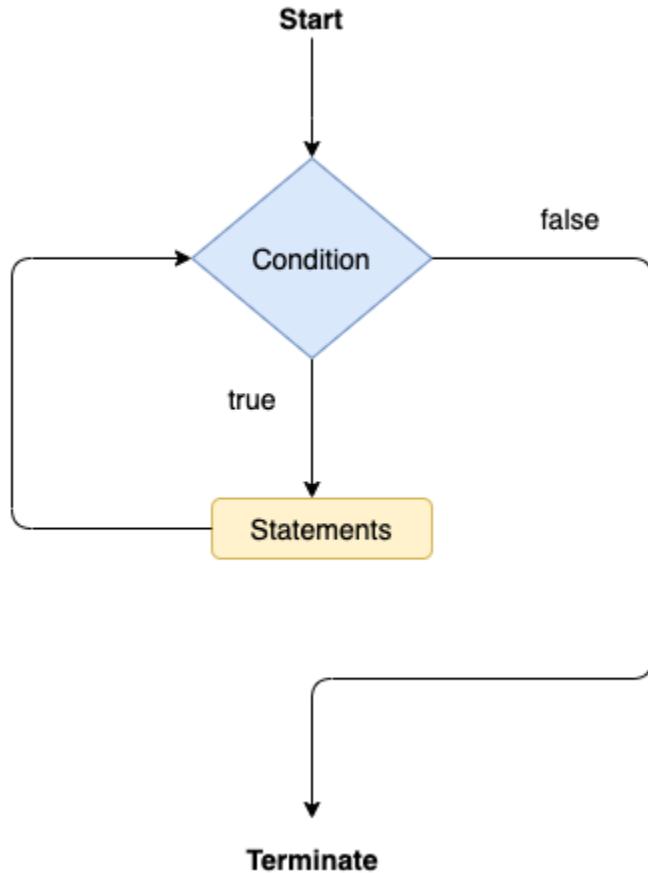
White Box Testing



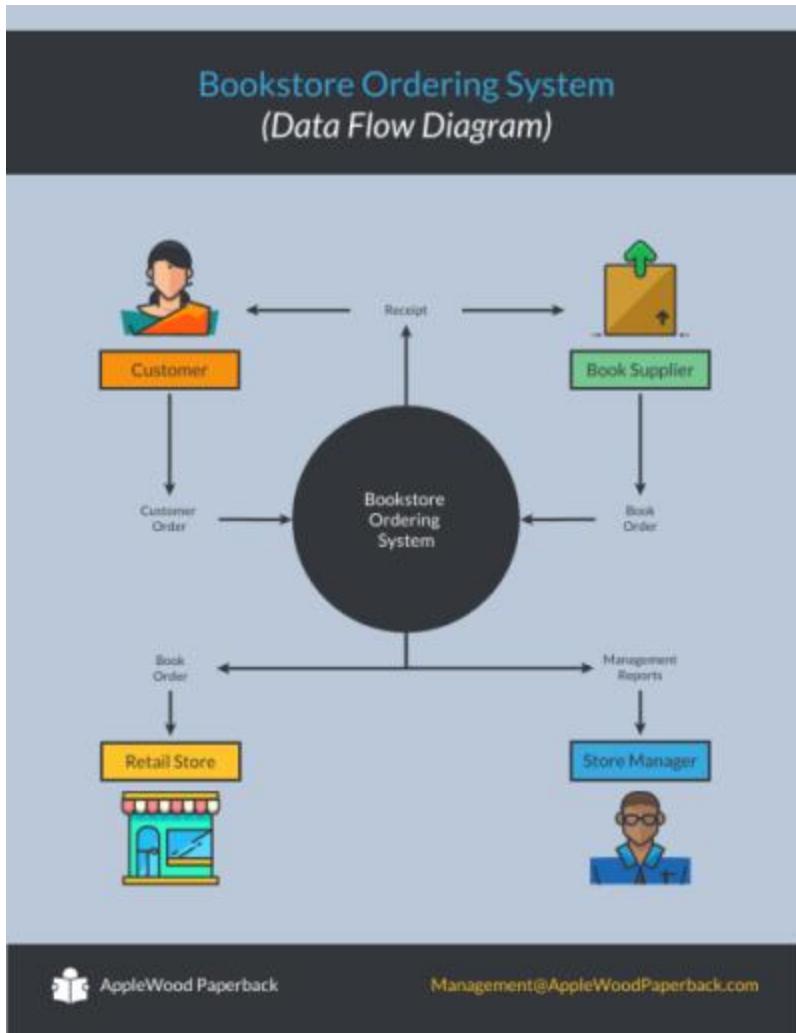
In **white box testing**, testers have access to the source code of the application and use this knowledge to design test cases that can **verify the correctness of the software at the code level**. White box testing is often used to test the following aspects of a software application:

```
$function(){cards();},  
$(window).on('resize', function(){cards();});  
function cards(){  
    var width = $(window).width();  
    if(width < 750){  
        cardssmallscreen();  
    } else{  
        cardsbigscreen();  
    }  
}  
function cardssmallscreen(){  
    var cards = $('.card');  
    height = 0;  
    i = 0;  
    cards.each(function(index){  
        cards.eq(index).css('height', height);  
        height += 100;  
    });  
}  
function cardsbigscreen(){  
    var cards = $('.card');  
    height = 0;  
    i = 0;  
    cards.each(function(index){  
        cards.eq(index).css('height', height);  
        height += 100;  
    });  
}
```

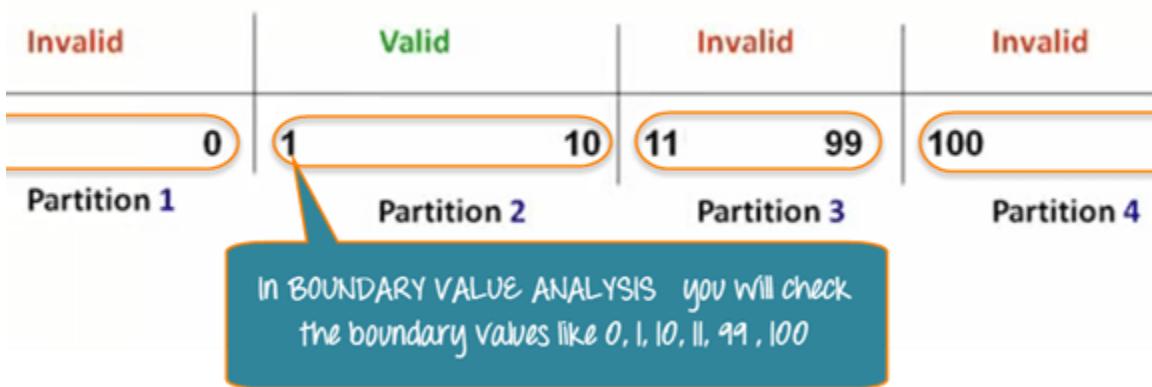
Control flow: White box testing can be used to ensure that all of the possible execution paths through an application are tested. This can be done by using techniques such as control flow analysis and path testing.



Data flow: White box testing can be used to ensure that all of the possible data flows through an application are tested. This can be done by using techniques such as data flow analysis and taint analysis.



Boundary values: White box testing can be used to ensure that the application behaves correctly at the boundaries of its input and output values. This can be done by using techniques such as equivalence partitioning and boundary value analysis.



Error handling: White box testing can be used to ensure that the application handles errors correctly. This can be done by designing test cases that trigger different types of errors.

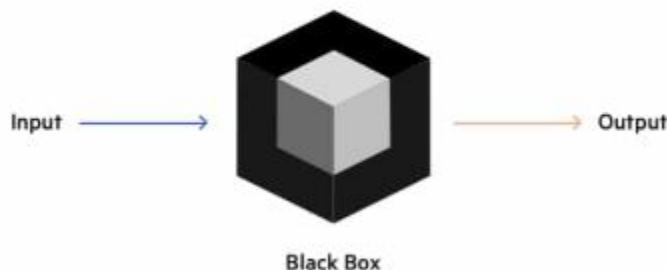


White box testing is a powerful tool for ensuring the quality of software applications.

However, it is important to note that white box testing alone cannot guarantee that an application is bug-free.

Black box testing is also necessary to test the application from a user's perspective and to ensure that it meets all of its functional requirements.

Black Box Testing



Benefits of white box testing:

- Identify bugs early in the development process.
- Improve code quality by identifying potential design problems and inefficiencies.
- Improve security by identifying potential vulnerabilities

Drawbacks of white box testing:

- Time-consuming and expensive.

- Difficult to perform if testers do not have a good understanding of the source code.
- Cannot guarantee that an application is bug-free.

In your provided assembly language code example, you're implementing a nested IF-ELSE statement and conducting white box testing by assigning different values to the variables and tracing the execution paths.

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 show the test values assigned to op1, op2, X, and Y. The fifth column shows the execution path that is taken, based on the test values. The sixth column shows the output that is produced, based on the execution path.

For example, in the first test case, the values of op1 and op2 are both 10, and the values of X and Y are 20 and 30, respectively. Since op1 is equal to op2, the execution path will follow the first branch of the IF statement, and Routine1 will be called. The output of Routine1 is unspecified, but it is likely to return a value that indicates that the condition $op1 == op2$ is true.

In the second test case, the values of op1 and op2 are both 10, and the values of X and Y are 30 and 20, respectively. Again, since op1 is equal to op2, the execution path will follow the first branch of the IF statement. However, this time, the condition $X > Y$ is also true, so the execution path will follow the first branch of the nested IF statement. This will result in Routine1 being called.

In the third test case, the values of op1 and op2 are 10 and 20, respectively, and the values of X and Y are 30 and 20, respectively. Since op1 is not equal to op2, the execution path will follow the second branch of the IF statement. The condition $X > Y$ is false, so the execution path will follow the second branch of the nested IF statement. This will result in Routine2 being called.

In the fourth test case, the values of op1 and op2 are 10 and 20, respectively, and the values of X and Y are 20 and 30, respectively. Since op1 is not equal to op2, the execution path will follow the second branch of the IF statement. The condition $X > Y$ is also false, so the execution path will fall through to the ELSE clause of the nested IF statement. This will result in Routine3 being called.

White box testing is a valuable tool for ensuring that complex conditional statements are working as expected. By testing all possible combinations of input values, programmers can be confident that their code will handle all possible scenarios correctly.

SHORT CIRCUIT EVALUATION(AND)

Short-circuit evaluation is a technique used by compilers and interpreters to optimize the evaluation of Boolean expressions.

With short-circuit evaluation, the second operand of an AND expression is only evaluated if the first operand is true.

This is because if the first operand is false, then the overall expression must be false, regardless of the value of the second operand.

The following assembly language code implements short-circuit evaluation 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 only evaluated if the first operand is false.

This is because if the first operand is true, then the overall expression must be true, regardless of the value of the second operand.

The following assembly language code implements short-circuit evaluation 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 first compares the values of the registers al and bl. If al is greater than bl, then the second operand of the OR expression is not evaluated, and the program jumps to the L1 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 OR expression evaluated to true or false.

The following table shows the difference 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 a ja instruction instead of a jbe instruction. The ja instruction can be implemented as a single machine instruction, while the jbe 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 ja instruction or a jbe 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.

As you mentioned, there are multiple ways to implement a compound expression containing OR operators in assembly language. For example, the following code is also 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 first compares the values of the registers al and bl. If al is equal to bl, then the program jumps to the L1 label. Otherwise, the program compares the values of the registers bl and cl.

If bl is equal to cl, then the program jumps to the L1 label. Otherwise, the program stores the value 1 in the register X and jumps to the next label.

The L1 label is used to indicate that the overall expression is true. The next label is used to exit the code, regardless of whether the OR expression evaluated to true or false.

Ultimately, the best way to implement a compound expression containing OR operators in assembly language will depend on the specific needs of the program.

WHILE LOOPS

While loop

The notes you provided for WHILE loops in assembly language are a bit confusing and unexplained, so I will try to rewrite them in a more clear and concise way.

WHILE loops in assembly language work in a similar way to WHILE loops in high-level languages. The loop first checks a condition. If the condition is true, the loop body is executed.

Then, the condition is checked again. If the condition is still true, the loop body is executed again. This process continues until the condition becomes false.

To implement a WHILE loop in assembly language, you can use the following steps:

Initialize a register to store the loop condition. Check the loop condition. If the condition is false, jump to the end of the loop. Execute the loop body. Update the loop condition. Jump back to step 2.

The following assembly code shows 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 the notes you provided mention, it is often convenient to reverse the loop condition in assembly language. This means that the loop will continue to iterate as long as the condition is false.

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

The jnl instruction jumps to the specified label if the condition is not less than or equal to zero.

For example, the following assembly code is equivalent to the previous example, but it reverses the 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 the loop variable is used inside the loop body, you need to copy it to a register before the loop starts.

Then, you need to restore the value of the loop variable at the end of the loop.

This is necessary because assembly language is a stack-based language.

This means that all variables are stored on the stack. When a function is called, the parameters are pushed onto the stack.

When the function returns, the parameters are popped off the stack.

If you use a loop variable inside the loop body, the loop variable will be pushed onto the stack when you call the loop body.

When the loop body returns, the loop variable will be popped off the stack. This means that the loop variable will be modified by the loop body.

To avoid this problem, you need to copy the loop variable to a register before the loop starts. Then, you need to restore the value of the loop variable at the end of the loop.

For example, the following assembly code shows how to copy and restore the 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
```

The first instruction copies the value of the variable val1 to the register eax. This is done because the loop will be operating on eax, so it is important to have a copy of val1 in a register.

The next instruction is a cmp instruction that compares the values of eax and val2. If eax is not less than val2, then the loop condition is false and the program will jump to the endwhile label.

If the loop condition is true, then the program will execute the following instructions:

Increment the value of eax by 1. This corresponds to the val1++ statement in the C++ code. Decrement the value of val2 by 1. This corresponds to the val2-- statement in the C++ code.

Jump to the beginwhile label to repeat the loop. The endwhile label is used to mark the end of the loop. When the program reaches the endwhile label, it will exit the loop and continue with the rest of the program.

The last instruction copies the value of eax to the variable val1. This is done because we need to save the new value of val1 in the variable before exiting the loop.

The JNL instruction is used to jump to the endwhile label if the loop condition is not true. This instruction is used because val1 and val2 are signed integers. If val1 is greater than val2, then the loop condition is false and we need to exit the loop.

It is important to note that the eax register is used as a proxy for the variable val1 inside the loop. This means that all references to val1 must be through the eax register. This is because the loop will be operating on eax, not val1.

In this code:

The mov instruction copies the value of val1 to the EAX register. The beginwhile label marks the beginning of the loop. The cmp instruction compares the values in EAX and val2.

The jnl instruction jumps to the endwhile label if not (EAX < val2), effectively ending the loop. inc eax increments the value in EAX, representing val1++. dec val2 decrements the value in val2, representing val2--.

The jmp beginwhile instruction jumps back to the beginwhile label to repeat the loop until the condition is no longer met. Finally, the endwhile label marks the end of the loop, and mov val1, eax saves the updated value of val1 back to the variable val1.

```
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 }
```

It defines an integer array array with ten elements. It initializes a variable sample with the value 50. It calculates the size of the array using sizeof(array) divided by sizeof(sample) to get the number of elements in the array, and stores it in ArraySize.

It initializes an index variable index to 0 to keep track of the current position in the array. It initializes a sum variable to 0 to accumulate the sum of elements greater than sample. The while loop iterates through the elements of the array using the index variable. For each element, it checks if it is greater than sample.

If it is, it adds that element's value to the sum. The loop continues until index is less than ArraySize. Finally, it prints the sum of elements in the array that are greater than the sample. If you have any further questions or need additional explanations, please let me know

The C++ code you provided is a good example of a nested IF statement inside a WHILE loop. The logic of the code is as follows:

```
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 }
```

Initialize the variables sum, sample, ArraySize, index, and sum. Enter the WHILE loop. Compare the value of index to the value of ArraySize.

If index is less than ArraySize, proceed to step 4. Otherwise, exit the loop. Compare the value of array[index] to the value of sample.

If array[index] is greater than sample, add the value of array[index] to the variable sum. Increment the value of index.

Repeat steps 3-5 until the WHILE loop condition is no longer met. Exit the WHILE loop and store the value of sum in the variable sum.

The assembly code you provided is a good translation of the C++ code. It uses registers to store the values of the variables sum, sample, array, ArraySize, and index. It also uses labels to mark the different parts of the code flow.

Here is a brief explanation of the assembly code:

```
535 ; sum_of_elements_greater_than_sample  
536 ; rdi: array  
537 ; rsi: sample  
538 ; rdx: size  
539 ; rax: sum  
540 mov rax, 0  
541 cmp rsi, [rdi]  
542 jl done  
543 add rax, [rdi]  
544 inc rdi  
545 jmp sum_of_elements_greater_than_sample  
546 done:  
547 ret
```

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

IF STATEMENTS IN ASSEMBLY

```
605 int array[] = {10, 60, 20, 33, 72, 89, 45, 65, 72, 18};  
606 int sample = 50;  
607 int ArraySize = sizeof array / sizeof sample;  
608 int index = 0;  
609 int sum = 0;  
610  
611 while (index < ArraySize) {  
612     if (array[index] > sample) {  
613         sum += array[index];  
614     }  
615     index++;  
616 }
```

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:

```
619 .data
620     sum DWORD 0
621     sample DWORD 50
622     array DWORD 10, 60, 20, 33, 72, 89, 45, 65, 72, 18
623     ArraySize = ($ - Array) / TYPE array
624 .code
625     main PROC
626         mov eax, 0 ; sum
627         mov edx, sample
628         mov esi, 0 ; index
629         mov ecx, ArraySize
630
631     L1:
632         cmp esi, ecx ; if esi < ecx
633         jl L2
634         jmp L5
635
636     L2:
637         cmp array[esi * 4], edx ; if array[esi] > edx
638         jg L3
639         jmp L4
640
641     L3:
642         add eax, array[esi * 4]
643
644     L4:
645         inc esi
646         jmp L1
647
648     L5:
649         mov sum, eax
```

This code works in the following way:

The loop starts by initializing the eax register to 0. This register will be used to store the sum of all array elements greater than the value in sample. The loop then compares the esi register to the ecx register.

If the esi register is less than the ecx register, the loop jumps to the L1 label. This means that the loop will continue to iterate until it has examined all of the array elements. If the

loop jumps to the L1 label, it then compares the value of the array element at array[esi * 4] to the value in the edx register.

If the value of the array element is greater than the value in the edx register, the loop jumps to the L3 label. If the loop jumps to the L3 label, it then adds the value of the array element at array[esi * 4] to the eax register.

The loop then increments the esi register and jumps back to the L1 label. This process continues until the loop has examined all of the array elements. Once the loop has examined all of the array elements, it jumps to the L5 label.

This label marks the end of the loop. At the end of the loop, the value of the eax register is stored in the sum variable.

Improvements

:

There are a few things that could be done to improve the assembly language code above:

The cmp instruction in the L1 label could be replaced with a test instruction.

The test instruction is faster than the cmp instruction because it does not set the condition flags. The jmp instruction in the L1 label could be replaced with a loop instruction.

The loop instruction is faster than the jmp instruction because it does not push the return address onto the stack. The cmp instruction in the L2 label could be replaced with a sub instruction.

The sub instruction is faster than the cmp instruction because it does not set the condition flags. The jmp instruction in the L2 label could be replaced with a jbe instruction.

The jbe instruction is faster than the jmp instruction because it does not push the return address onto the stack.

FIGURE 6–1 Loop containing IF statement.

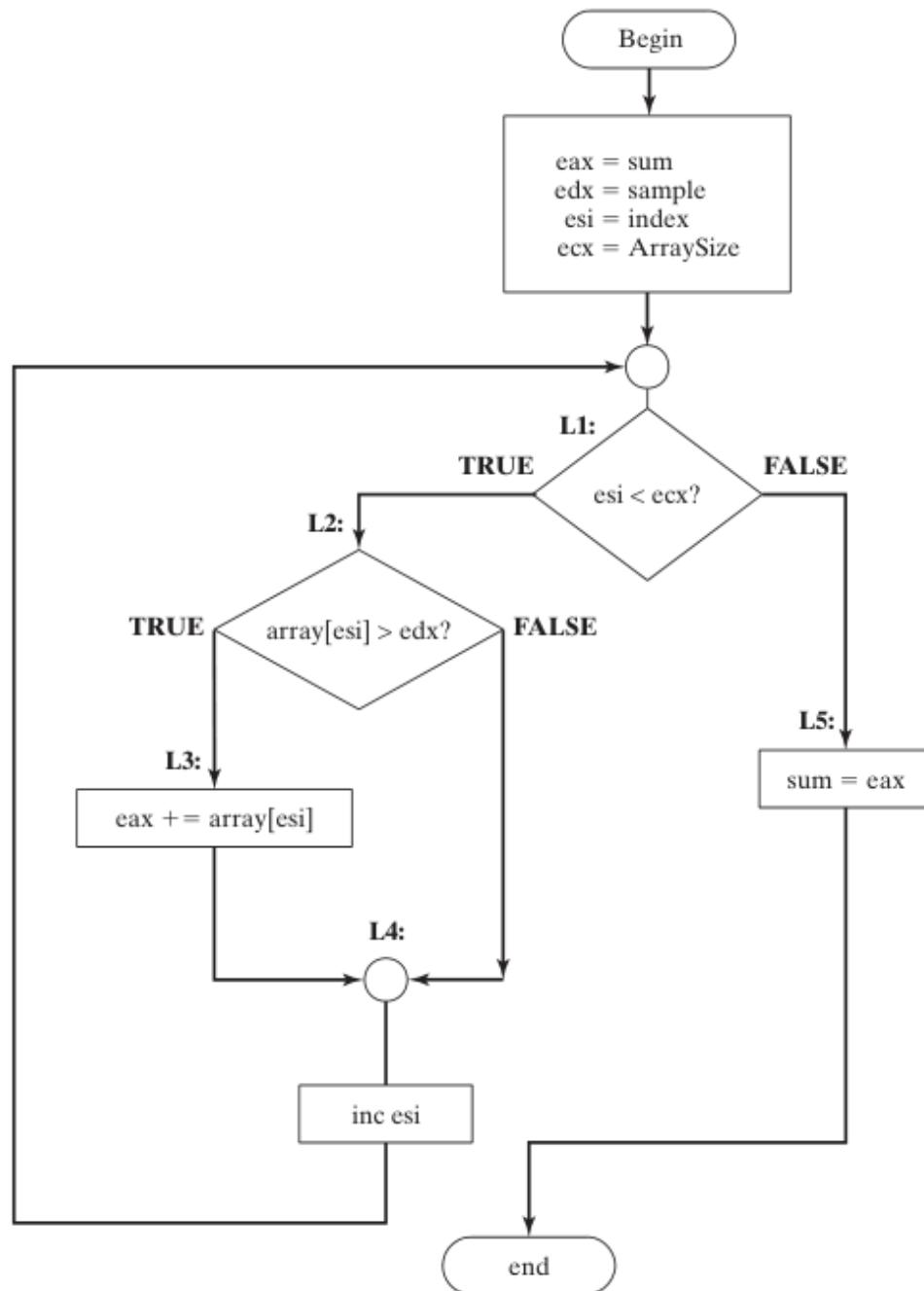


TABLE DRIVEN SELECTION

Table-driven selection is a way of using a table lookup to replace a multiway selection structure.

This can be useful when there are a large number of possible values to compare, as it can avoid the need to write a series of nested IF statements.

To use table-driven selection, you first need to create a table of lookup values and the addresses of the corresponding procedures.

Then, you need to write a loop to search the table and call the appropriate procedure based on the lookup value.

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

```
654 .data
655     CaseTable BYTE 'A'
656     ; lookup value
657     DWORD Process_A
658     ; address of procedure
659     BYTE 'B'
660     DWORD Process_B
661     (etc.)
662
663 .code
664     mov eax, [esi] ; get the lookup value
665     cmp eax, CaseTable ; compare to first lookup value
666     je Process_A ; if equal, call the corresponding procedure
667     cmp eax, CaseTable + 1 ; compare to second lookup value
668     je Process_B ; if equal, call the corresponding procedure
669     (etc.)
670
671     ; if no match is found, do something else
```

The loop in this example iterates over the table of lookup values and compares each value to the value in the eax register. If a match is found, the corresponding procedure is called. If no match is found, the loop terminates and the program can do something else.

The table-driven selection example in the image you provided shows a table of lookup values and the addresses of corresponding procedures for a simple calculator. The table contains the following lookup values:

- 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 code will compare the value in the eax register to the first lookup value in the table. If the two values are equal, the Add procedure is called.

Otherwise, the loop counter is incremented and the loop is repeated. The loop continues to iterate until the loop counter is greater than or equal to the size of the table.

When the loop terminates, the sum of the two numbers is stored in the eax register.

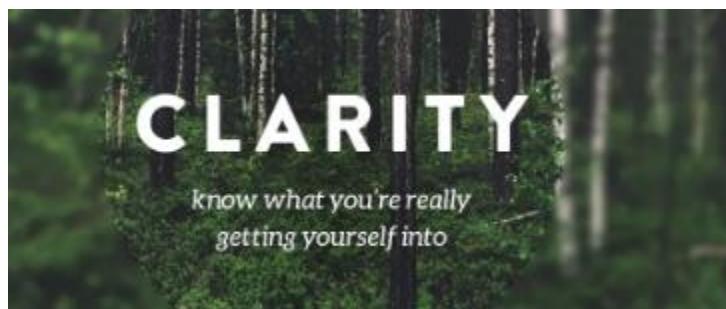
Advantages of table-driven selection

Table-driven selection can offer a number of advantages over other methods of implementing multiway selection structures, such as nested IF statements. Some of the advantages of table-driven selection include:

Efficiency: Table-driven selection can be more efficient than other methods of implementing multiway selection structures, as it can avoid the need to write a series of nested IF statements.



Clarity: Table-driven selection can make code more readable and maintainable, as it can simplify the implementation of complex multiway selection structures.



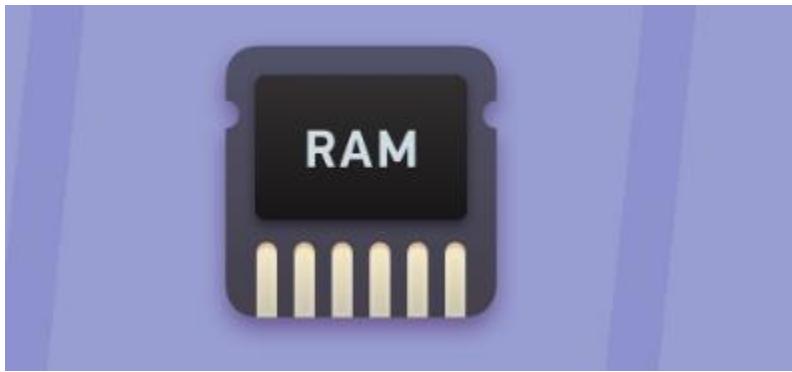
Flexibility: Table-driven selection can be more flexible than other methods of implementing multiway selection structures, as it can be easily extended to support new lookup values and procedures.



Disadvantages of table-driven selection

Table-driven selection also has some disadvantages, such as:

Memory usage: Table-driven selection can require more memory than other methods of implementing multiway selection structures, as it requires a table to be stored in memory.



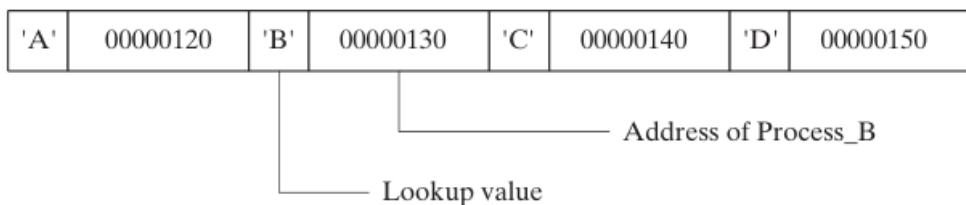
Speed: Table-driven selection can be slower than other methods of implementing multiway selection structures, as it requires a loop to search the table.



Overall, table-driven selection is a useful technique for implementing multiway selection structures, especially when there are a large number of possible values to compare.

However, it is important to be aware of the advantages and disadvantages of table-driven selection before using it in your code.

Example 1:



Program written in assembly language (x86) 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 calls the corresponding procedure to display a message. Here's a breakdown of the program with explanations:

```
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
```

```
777 .data
778 CaseTable BYTE 'A'
779             DWORD Process_A
780 EntrySize = ($ - CaseTable)
781 BYTE 'B'
782             DWORD Process_B
783 BYTE 'C'
784             DWORD Process_C
785 BYTE 'D'
786             DWORD Process_D
787 NumberOfEntries = ($ - CaseTable) / EntrySize
```

In this section, we define the data for our program:

CaseTable is a table that contains characters ('A', 'B', 'C', 'D') and the addresses of corresponding procedures (Process_A, Process_B, Process_C, Process_D).

EntrySize is calculated as the difference between the current memory position (\$) and CaseTable. This represents the size of each entry in the table.

NumberOfEntries calculates the number of entries in CaseTable by dividing the size of the table by EntrySize.

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.

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, we perform the following tasks:

mov edx, OFFSET prompt: Load the address of the prompt message into the edx register, displaying the prompt.

call WriteString: Call a procedure to print the prompt.

call ReadChar: Call a procedure to read a character from the user and store it in the al register.

mov ebx, OFFSET CaseTable: Load the address of CaseTable into the ebx register.

mov ecx, NumberOfEntries: Load the number of entries in the table into the ecx register.

```
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 this part of the main procedure:

L1 is a label marking the start of a loop.

cmp al, [ebx] compares the user input character (al) with the character in the current entry of CaseTable.

jne L2 jumps to L2 if there's no match (continue searching).

call NEAR PTR [ebx + 1] calls the procedure stored in the table.

call WriteString displays the corresponding message.

call Crlf adds a line break.

jmp L3 jumps to L3 (exit). The loop continues until a match is found or all entries have been checked.

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 procedures (Process_A to Process_D) that set the edx register with the address of the corresponding message string and return.

This section marks the end of the main program.

In summary, the code defines a lookup table, messages, and procedures. The main procedure reads user input, searches the table for a match, and calls the corresponding procedure to display a message.

The table-driven approach makes it easy to extend and modify the program for different cases.

=====

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(long one), it's better to let the assembler calculate NumberOfEntries rather than assigning a constant because it makes the code more flexible and maintainable.

If you hardcode a constant like NumberOfEntries - 4, you would need to manually update it if the size of the entries changes in the future.

By letting the assembler calculate it, you ensure that it always reflects the actual size, reducing the risk of errors and making your code more adaptable.

To rewrite the code from Section above with fewer instructions while maintaining functionality, you can use conditional move (CMOV) instructions. Here's an example using CMOV:

```
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 cmoveb 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.

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:

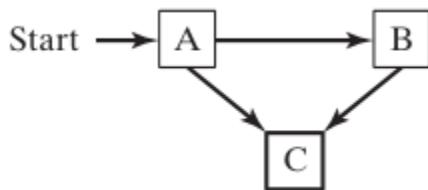
```

895 Initial state: Start
896 Terminal state: Exit
897
898 Transitions:
899 Start -> A on input "a"
900 Start -> B on input "b"
901 Start -> C on input "c"
902 A -> B on input "a"
903 A -> C on input "b"
904 A -> A on input "c"
905 B -> C on input "a"
906 B -> B on input "b"
907 B -> A on input "c"
908 C -> A on input "a"
909 C -> B on input "b"
910 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.



Here is a more detailed explanation of the diagram:

The initial state is the Start state.

The three possible states are A, B, and C.

The arrows show the possible transitions between states. The terminal state is the Exit state. The FSM can be described in words as follows:

- The FSM starts in the Start state.
- If the FSM receives the input "a", it transitions to the A state.
- If the FSM receives the input "b", it transitions to the B state.
- If the FSM receives the input "c", it transitions to the C state.
- If the FSM is in the A state and receives the input "a", it transitions to the B state.
- If the FSM is in the A state and receives the input "b", it transitions to the C state.
- If the FSM is in the A state and receives the input "c", it remains in the A state.
- If the FSM is in the B state and receives the input "a", it transitions to the C state.
- If the FSM is in the B state and receives the input "b", it remains in the B state.
- If the FSM is in the B state and receives the input "c", it transitions to the A state.
- If the FSM is in the C state and receives the input "a", it transitions to the A state.
- If the FSM is in the C state and receives the input "b", it transitions to the B state.
- If the FSM is in the C state and receives the input "c", it remains in the C state.

The FSM will continue to transition between states until it reaches the terminal state, the Exit state.

=====

Validating an Input String Programs

=====

Here is a more detailed explanation of the example FSM in Figure 6-4:

Start state: **A**

Terminal state: **C**

Transitions: **A -> B on input "x"**

- **B -> B on input any letter in the range {a, b, ..., y}** **B -> C on input "z"** This FSM can be used to validate an input string according to the following two rules:

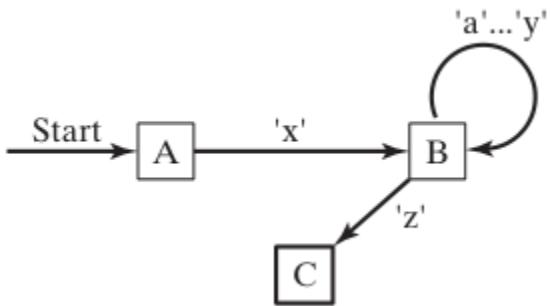
- The string must begin with the letter "x" and end with the letter "z". Between the first and last characters, there can be zero or more letters within the range {a, b, ..., y}.
- To validate an input string, the FSM starts in the A state.
- It then reads the input string character by character.
- If the next character is "x", the FSM transitions to the B state.
- If the next character is any letter in the range {a, b, ..., y}, the FSM remains in the B state.
- If the next character is "z", the FSM transitions to the C state.
- If the FSM reaches the C state, it means that the input string is valid.
- If the FSM reaches the end of the input string before it reaches the C state, it means that the input string is invalid.

Here is an example of a valid input string:

- xyz The FSM will start in the A state. It will then read the first character, which is "x". This will transition the FSM to the B state.
- The FSM will then read the second character, which is "y". This will cause the FSM to remain in the B state. The FSM will then read the third character, which is "z".
- This will transition the FSM to the C state. Since the FSM has reached the C state, it means that the input string is valid.

Here is another example of an invalid input string:

- xab The FSM will start in the A state. It will then read the first character, which is "x". This will transition the FSM to the B state.
- The FSM will then read the second character, which is "a". This will cause the FSM to remain in the B state.
- The FSM will then read the third character, which is "b". This will cause the FSM to remain in the B state.
- However, the FSM will not be able to transition to the C state, because the next input character is not "z". This means that the input string is invalid.



I have reviewed the notes again, and I understand them better now. The notes are describing how to use a FSM to validate an input string according to the following two rules:

The string must begin with the letter "x" and end with the letter "z". Between the first and last characters, there can be zero or more letters within the range {a, b, ..., y}.

The FSM diagram in the image shows the possible states and transitions of the FSM. The FSM starts in the A state.

If the next character in the input string is "x", the FSM transitions to the B state. If the next character in the input string is any letter in the range {a, b, ..., y}, the FSM remains in the B state.

If the next character in the input string is "z", the FSM transitions to the C state.

If the FSM reaches the C state, it means that the input string is valid. If the FSM reaches the end of the input string before it reaches the C state, it means that the input string is invalid.

Here is an example of a valid input string:

xyz The FSM will start in the A state. It will then read the first character, which is "x". This will transition the FSM to the B state.

The FSM will then read the second character, which is "y". This will cause the FSM to remain in the B state. The FSM will then read the third character, which is "z".

This will transition the FSM to the C state. Since the FSM has reached the C state, it means that the input string is valid.

Here is an example of an invalid input string:

xab The FSM will start in the A state. It will then read the first character, which is "x". This will transition the FSM to the B state.

The FSM will then read the second character, which is "a". This will cause the FSM to remain in the B state. The FSM will then read the third character, which is "b".

This will cause the FSM to remain in the B state. However, the FSM will not be able to transition to the C state, because the next input character is not "z". This means that the input string is invalid.

- If the end of the input stream is reached while the program is in state A or B, an error condition results because only state C is marked as a terminal state. This means that the input string must end with the letter "z" in order for it to be valid.
- The following input strings would be recognized by this FSM:

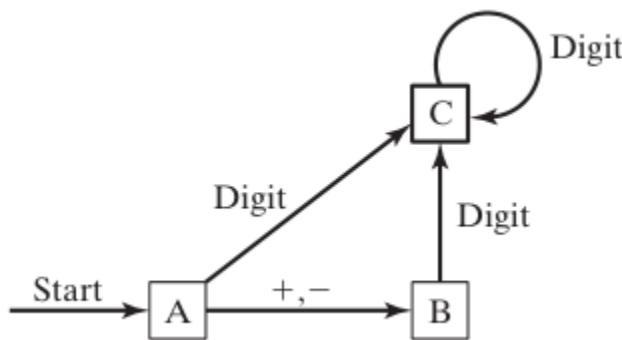
xaabcdefgz xz xyyqqrrstuvwxyz

- All of these input strings begin with the letter "x" and end with the letter "z". There may be any number of letters in the range {a, b, ..., y} in between.
- Here is an example of an input string that would not be recognized by this FSM:

xab

- This input string begins with the letter "x" but does not end with the letter "z". Therefore, it is invalid.

Validating Integers in Programs



The FSM diagram in Figure 6-5 shows how to validate a signed integer. The FSM starts in the Start state.

If the next character in the input stream is a plus sign (+) or a minus sign (-), the FSM transitions to the Sign state.

If the next character in the input stream is a digit, the FSM transitions to the Digits state.

If the FSM is in the Sign state and the next character in the input stream is a digit, the FSM remains in the Sign state.

This is because the sign can be followed by any number of digits.

If the FSM is in the Digits state and the next character in the input stream is a digit, the FSM remains in the Digits state.

This is because the sequence of digits can be any length.

If the FSM is in the Digits state and the next character in the input stream is not a digit, the FSM transitions to the End state.

This is because the sequence of digits must end with a non-digit character.

The End state is a terminal state. This means that the input string is valid if the FSM reaches the End state.

Here is an example of a valid input string:

-123456

-123456 The FSM will start in the Start state. It will then read the first character, which is a minus sign (-).

This will transition the FSM to the Sign state. The FSM will then read the second character, which is the digit 1.

This will cause the FSM to remain in the Sign state. The FSM will then read the third character, which is the digit 2.

This will cause the FSM to remain in the Sign state. The FSM will then read the fourth character, which is the digit 3.

This will cause the FSM to remain in the Sign state. The FSM will then read the fifth character, which is the digit 4.

This will cause the FSM to remain in the Sign state. The FSM will then read the sixth character, which is the digit 5.

This will cause the FSM to remain in the Sign state. The FSM will then read the seventh character, which is the digit 6.

This will cause the FSM to transition to the Digits state. The FSM will then read the eighth character, which is not a digit.

This will cause the FSM to transition to the End state. Since the FSM has reached the End state, it means that the input string is valid.

Here is another example of an invalid input string:

-123456.78 ;invalid

-123456.78 The FSM will start in the Start state. It will then read the first character, which is a minus sign (-).

This will transition the FSM to the Sign state.

The FSM will then read the second character, which is the digit 1. This will cause the FSM to remain in the Sign state.

The FSM will then read the third character, which is the digit 2. This will cause the FSM to remain in the Sign state.

The FSM will then read the fourth character, which is the digit 3. This will cause the FSM to remain in the Sign state.

The FSM will then read the fifth character, which is the digit 4. This will cause the FSM to remain in the Sign state.

The FSM will then read the sixth character, which is the digit 5. This will cause the FSM to remain in the Sign state.

The FSM will then read the seventh character, which is the digit 6. This will cause the FSM to transition to the Digits state.

The FSM will then read the eighth character, which is a period (.). This is not a digit, so the FSM will transition to the End state.

However, the End state is not a terminal state. This means that the input string is invalid.

FSMs are a powerful tool for validating input strings and other types of data. They are used in a wide variety of applications, including programming language compilers, text editors, and network protocols.

=====

Validating Integers in Programs

=====

FSM for parsing a signed integer:

- This FSM diagram shows how to validate a signed integer. The FSM starts in the Start state. If the next character in the input stream is a plus sign (+) or a minus sign

(-), the FSM transitions to the Sign state. If the next character in the input stream is a digit, the FSM transitions to the Digits state.

- If the FSM is in the Sign state and the next character in the input stream is a digit, the FSM remains in the Sign state. This is because the sign can be followed by any number of digits.
- If the FSM is in the Digits state and the next character in the input stream is a digit, the FSM remains in the Digits state. This is because the sequence of digits can be any length.
- If the FSM is in the Digits state and the next character in the input stream is not a digit, the FSM transitions to the End state. This is because the sequence of digits must end with a non-digit character.
- The End state is a terminal state. This means that the input string is valid if the FSM reaches the End state.

The following assembly language code implements the FSM diagram above:

```
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
```

This code works as follows:

- The label StateA marks the start of the FSM.
- The call to the Getnext procedure reads the next character from the console input into the AL register.
- The code checks for a leading plus (+) or minus (-) sign.
- If a leading sign is found, the code jumps to the label StateB.
- If a leading sign is not found, the code calls the IsDigit procedure to check if the character in AL is a digit. If the character is a digit, the code jumps to the label StateC.
- If the character in AL is not a digit or a leading sign, the code calls the DisplayErrorMsg procedure to display an error message on the console and then jumps to the label Quit.
- The label StateB marks the state where the FSM is after reading a leading sign.
- The code in StateB will check for the other possible transitions away from this state and take the appropriate action. The label StateC marks the state where the FSM is after reading a digit.
- The code in StateC will check for the other possible transitions away from this state and take the appropriate action. The label End marks the terminal state of the FSM.
- The code in End will perform any necessary cleanup and then exit the program. The main procedure simply calls the StateA procedure to start the FSM.
- This is just a basic example of how to implement a FSM in assembly language. More complex FSMs can be implemented using the same basic principles.

```
; Finite State Machine (Finite.asm)
INCLUDE Irvine32.inc

ENTER_KEY = 13

.data
InvalidInputMsg: db "Invalid input", 13, 10, 0

.code
main:
; Clear screen
call Clrscr
```

```
; Start state
StateA:
    ; Read next character into AL
    call Getnext

    ; Check for leading + or - sign
    cmp al, '+'
    je StateB
    cmp al, '-'
    je StateB

    ; Check if AL contains a digit
    call IsDigit
    jz StateC

    ; Invalid input
    call DisplayErrorMsg
     Quit

StateB:
    ; Read next character into AL
    call Getnext

    ; Check if AL contains a digit
    call IsDigit
    jz StateC

    ; Invalid input
    call DisplayErrorMsg
     Quit

StateC:
    ; Read next character into AL
    call Getnext

    ; Check if AL contains a digit
    call IsDigit
    jz StateC

    ; Check if Enter key pressed
    cmp al, ENTER_KEY
```

je Quit

```
; Invalid input  
call DisplayErrorMsg  
 Quit
```

Quit:

```
; Call Crlf to print a newline  
call Crlf  
exit
```

; Getnext procedure
; Reads a character from standard input
; Receives: nothing
; Returns: AL contains the character

Getnext:

```
; Input from keyboard  
call ReadChar
```

```
; Echo on screen  
call WriteChar
```

ret

; DisplayErrorMsg procedure
; Displays an error message indicating that
; the input stream contains illegal input
; Receives: nothing
; Returns: nothing

DisplayErrorMsg:

```
; Push EDX onto the stack  
push edx
```

```
; Move the offset of the error message to EDX  
mov edx, OFFSET InvalidInputMsg
```

```
; Call WriteString to print the error message  
call WriteString
```

```
; Pop EDX from the stack  
pop edx
```

ret

main procedure:

The main procedure is the entry point for the program. It starts by clearing the screen and then entering the StateA state.

StateA state:

The StateA state is the start state for the FSM. It reads the next character from the input stream and checks for a leading + or - sign. If a leading sign is found, the FSM transitions to the StateB state. If a leading sign is not found, the FSM checks if the character is a digit. If the character is a digit, the FSM transitions to the StateC state. Otherwise, the FSM calls the DisplayErrorMsg procedure to display an error message and then jumps to the Quit label to exit the program.

StateB state:

The StateB state is the state where the FSM is after reading a leading + or - sign. It reads the next character from the input stream and checks if it is a digit. If the character is a digit, the FSM transitions to the StateC state. Otherwise, the FSM calls the DisplayErrorMsg procedure to display an error message and then jumps to the Quit label to exit the program.

StateC state:

The StateC state is the state where the FSM is after reading a digit. It reads the next character from the input stream and checks if it is a digit. If the character is a digit, the FSM remains in the StateC state. Otherwise, the FSM checks if the Enter key was pressed. If the Enter key was pressed, the FSM transitions to the Quit label to exit the program. Otherwise, the FSM calls the DisplayErrorMsg procedure to display an error message and then jumps to the Quit label to exit the program.

Quit label:

The Quit label is the exit point for the program. The main procedure jumps to the Quit label to exit the program when the Enter key is pressed or when an invalid character is encountered.

Getnext procedure:

The Getnext procedure reads a character from standard input and echoes it to the screen. It returns the character in the AL register.

DisplayErrorMsg procedure:

The DisplayErrorMsg procedure displays an error message indicating that the input stream contains illegal input. It receives nothing and returns nothing. This is a basic example of how to implement a FSM in assembly language.

-
- The IsDigit procedure determines whether the character in the AL register is a valid decimal digit. It returns the setting of the Zero flag, which is 1 if the character is a valid decimal digit and 0 otherwise.
 - The IsDigit procedure works by first comparing the character in AL to the ASCII code for the digit 0. If the character is less than 0, then it is not a valid decimal digit and the Zero flag is cleared.
 - Next, the IsDigit procedure compares the character in AL to the ASCII code for the digit 9. If the character is greater than 9, then it is not a valid decimal digit and the Zero flag is cleared.
 - Finally, the IsDigit procedure sets the Zero flag.

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

The IsDigit procedure uses the CMP instruction to compare the character in the AL register to the ASCII codes for the digits 0 and 9. If the character is less than 0 or greater than 9, then the JB or JA instruction will jump to the label ID1, respectively.

The JB instruction jumps to a label when the Carry flag (CF) is set and the Zero flag (ZF) is clear. The JA instruction jumps to a label when the Carry flag (CF) is clear and the Zero flag (ZF) is clear.

If neither the JB nor the JA instruction jumps to the label ID1, then the TEST instruction is executed. The TEST instruction sets the Zero flag if the result of the AND operation is zero.

Since the JB and JA instructions jump to the label ID1 if the Zero flag is clear, the TEST instruction will only be executed if the character in the AL register is a digit.

Therefore, the IsDigit procedure returns the setting of the Zero flag, which is 1 if the character in the AL register is a digit and 0 otherwise.

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 are used to control the flow of execution of a program depending on the result of a condition. These directives are used to implement conditional statements such as if, else, and elseif.

The following are the most common conditional control flow directives in MASM:

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 <i>condition</i>	Generates code that tests <i>condition</i> 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 <i>condition</i>	Generates code that executes the block of statements if <i>condition</i> is true.
.REPEAT	Generates code that repeats execution of the block of statements until <i>condition</i> becomes true
.UNTIL <i>condition</i>	Generates code that repeats the block of statements between .REPEAT and .UNTIL until <i>condition</i> becomes true
.UNTILCXZ	Generates code that repeats the block of statements between .REPEAT and .UNTILCXZ until CX equals zero
.WHILE <i>condition</i>	Generates code that executes the block of statements between .WHILE and .ENDW as long as <i>condition</i> is true

Here are some additional things to keep in mind:

Conditions can be complex, but they must evaluate to a single Boolean value (true or false).

If the condition in the .IF directive is true, the assembler will assemble all of the statements between the .IF and .ELSEIF (or .ENDIF) directives.

If the condition in the .IF directive is false, the assembler will skip all of the statements between the .IF and .ELSEIF (or .ENDIF) directives.

If the .ELSEIF directive is present, the assembler will evaluate the condition in the .ELSEIF directive.

If the condition is true, the assembler will assemble all of the statements between the .ELSEIF and .ELSE (or .ENDIF) directives.

If the .ELSEIF directive is present and the condition is false, the assembler will skip all of the statements between the .ELSEIF and .ELSE (or .ENDIF) directives.

.ELSE directive is optional. If it is present, the assembler will assemble all of the statements between the .ELSE and .ENDIF directives if all of the previous conditions were false.

The .ENDIF directive is required. It tells the assembler the end of the conditional statement. Here is an example of a more complex conditional statement using the .IF, .ELSEIF, and .ELSE directives:

```
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.

Table 6-8 Runtime Relational and Logical Operators.

Operator	Description
<i>expr1</i> == <i>expr2</i>	Returns true when <i>expr1</i> is equal to <i>expr2</i> .
<i>expr1</i> != <i>expr2</i>	Returns true when <i>expr1</i> is not equal to <i>expr2</i> .
<i>expr1</i> > <i>expr2</i>	Returns true when <i>expr1</i> is greater than <i>expr2</i> .
<i>expr1</i> ≥ <i>expr2</i>	Returns true when <i>expr1</i> is greater than or equal to <i>expr2</i> .
<i>expr1</i> < <i>expr2</i>	Returns true when <i>expr1</i> is less than <i>expr2</i> .
<i>expr1</i> ≤ <i>expr2</i>	Returns true when <i>expr1</i> is less than or equal to <i>expr2</i> .
! <i>expr</i>	Returns true when <i>expr</i> is false.
<i>expr1</i> && <i>expr2</i>	Performs logical AND between <i>expr1</i> and <i>expr2</i> .
<i>expr1</i> <i>expr2</i>	Performs logical OR between <i>expr1</i> and <i>expr2</i> .
<i>expr1</i> & <i>expr2</i>	Performs bitwise AND between <i>expr1</i> and <i>expr2</i> .
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.

The table you sent shows the relational and logical operators in MASM. These operators are used to compare two values and return a Boolean value (true or false). The Boolean value can then be used to control the flow of execution of a program using conditional control flow directives such as .IF, .ELSE, and .ELSEIF.

The following is a detailed explanation of each of the operators in the table:

== (equal to): Returns true if the two values are equal.

!= (not equal to): Returns true if the two values are not equal.

> (greater than): Returns true if the first value is greater than the second value.

>= (greater than or equal to): Returns true if the first value is greater than or equal to the second value.

< (less than): Returns true if the first value is less than the second value.

<= (less than or equal to): Returns true if the first value is less than or equal to the second value.

&& (logical AND): Returns true if both operands are true.

|| (logical OR): Returns true if either operand is true.

! (logical NOT): Returns true if the operand 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
```

Here is a simpler explanation of the notes you provided:

Before using MASM conditional directives, be sure you thoroughly understand how to implement conditional branching instructions in pure assembly language.

This means that you should understand how to use the following assembly language instructions to implement conditional branching:

- • • **CMP (compare)**
- **JBE (jump if below or equal)**
- **JA (jump if above)**
- **JE (jump if equal)**
- **JNE (jump if not equal)**

Once you understand how to implement conditional branching in pure assembly language, you can use MASM conditional directives to make your code more concise and readable.

In addition, when a program containing decision directives is assembled, inspect the listing file to make sure the code generated by MASM is what you intended.

MASM conditional directives are translated into assembly language instructions by the assembler. It is a good idea to inspect the listing file to make sure that the assembler generated the code that you expected. This can help you to identify any errors in your code.

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

You can control whether or not MASM-generated code appears in the source listing file by setting the Enable Assembly Generated Code Listing property in the Visual Studio Project Properties dialog box.

To do this, follow these steps: 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.

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

If you compare two registers using the .IF directive, the assembler cannot determine whether the values are signed or unsigned. Therefore, the assembler will default to an unsigned comparison.

Example:

```
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:
```

Conclusion: It is important to be aware of whether the values you are comparing are signed or unsigned when using the .IF directive. This will help you to ensure that the assembler generates the correct conditional jump instruction.

COMPOUND EXPRESSIONS

Compound Boolean expressions allow you to combine two or more Boolean expressions using the logical OR and AND operators.

The **logical OR operator (||)** returns true if either of the Boolean expressions is true. The logical AND operator (&&) returns true if both of the Boolean expressions are true. Using Compound Expressions with the .IF Directive

You can use compound expressions with the .IF directive to control the flow of execution of your program.

For example, the following .IF directive uses the logical OR operator to compare the values of the eax and ebx registers:

```
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 Example

=====

The SetCursorPosition procedure sets the cursor position to the specified coordinates. It receives two input parameters: DL (X-coordinate) and DH (Y-coordinate).

The procedure first checks if the X-coordinate and Y-coordinate are within the valid ranges. If either coordinate is out of range, the procedure displays an error message and exits.

The following code shows the range checking code 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 is used to check if the X-coordinate or Y-coordinate is out of range. The logical OR operator (||) is used to combine the two conditions.

If either condition is true, the procedure displays an error message and exits.

If the X-coordinate and Y-coordinate are within the valid ranges, the procedure calls the Gotoxy procedure to set the cursor position.

The following code shows the code that sets the cursor position:

```
call Gotoxy
```

The Gotoxy procedure is a built-in MASM procedure that sets the cursor position to the specified coordinates.

The SetCursorPosition procedure is an example of how to use the .IF directive to range check input parameters. This can be helpful for preventing errors in your program.

=====

College Registration Example

=====

The college registration example you provided 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 is used to check the first condition: whether the student's grade average is greater than 350. If the condition is true, the mov instruction sets the OkToRegister variable to TRUE.

The .ELSEIF directive is used to check the second condition: whether the student's grade average is greater than 250 and the number of credits the student wants to take is less than or equal to 16. If both conditions are true, the mov instruction sets the OkToRegister variable to TRUE.

The .ELSEIF directive is used to check the third condition: whether the number of credits the student wants to take is less than or equal to 12. If the condition is true, the mov instruction sets the OkToRegister variable to TRUE.

If none of the conditions are true, the OkToRegister variable will remain set to FALSE.

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

Here is a more complete explanation of the .REPEAT and .WHILE directives:

=====

.Repeat directive

=====

The .REPEAT directive creates a loop that executes the statements in the loop body once before testing the condition following the .UNTIL directive.

If the condition is false, the loop body is executed again. The loop continues to execute until the condition becomes 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
```

This code will display the values 1 through 10 to the console.

=====

.While directive

=====

The .WHILE directive creates a loop that tests the condition before executing the loop body. If the condition is false, the loop body is skipped. The loop continues to execute until the condition 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
```

This code will also display the values 1 through 10 to the console.

Differences Between .REPEAT and .WHILE

The main difference between the .REPEAT and .WHILE directives is that the .REPEAT directive executes the loop body at least once, even if the condition is false. The .WHILE directive, on the other hand, will skip the loop body if the condition is false.

Which Directive to Use?

In general, you should use the .WHILE directive for loops where you need to test the condition before executing the loop body. This is because the .WHILE directive is more efficient than the .REPEAT directive, as it avoids executing the loop body if the condition is false.

However, there are some cases where you may want to use the .REPEAT directive. For example, you may want to use the .REPEAT directive if you need to initialize a variable before executing the loop body.

Conclusion

The .REPEAT and .WHILE directives are two powerful tools for creating loops in MASM. By understanding the differences between the two directives, you can choose the right directive for your needs.

The notes you provided are unclear and incomplete because they do not provide a complete explanation of how to implement the pseudocode using the .WHILE and .IF directives.

Here is a more complete explanation:

```

1184 .data
1185     X DWORD 0
1186     op1 DWORD 2
1187     ; test data
1188     op2 DWORD 4
1189     ; test data
1190     op3 DWORD 5
1191     ; test data
1192 .code
1193     mov eax, op1
1194     mov ebx, op2
1195     mov ecx, op3
1196
1197     .WHILE eax < ebx
1198         inc eax
1199
1200         .IF eax == ecx
1201             mov X, 2
1202         .ELSE
1203             mov X, 3
1204         .ENDIF
1205     .ENDW

```

This code will loop from the value of op1 to the value of op2, incrementing op1 on each iteration. Within the loop, the code uses the .IF directive to check if op1 is equal to op3.

If it is, the code moves the value 2 to X. Otherwise, the code moves the value 3 to X.

The .WHILE directive will continue to loop until op1 is greater than or equal to op2.

Here is a breakdown of the code:

The code you provided is a loop that executes the following steps:

1. **Moves the values of the variables op1, op2, and op3 to the registers eax, ebx, and ecx, respectively.**
2. **Starts a .WHILE loop that will continue to execute until eax is greater than or equal to ebx.**
3. **Increments the eax register by 1.**
4. **Uses the .IF directive to check if eax is equal to ecx.**
 - • If eax is equal to ecx, the code moves the value 2 to the variable X.

- Otherwise, the code moves the value 3 to the variable X.
- Ends the .WHILE loop.

This loop will essentially iterate from the value of op1 to the value of op2, incrementing op1 on each iteration. Within the loop, the code checks if op1 is equal to op3. If it is, the code moves the value 2 to X. Otherwise, the code moves the value 3 to X.

Here is a simpler explanation:

- Input: Three variables: op1, op2, and op3.
- Output: The variable X.
- Algorithm: 1. Initialize X to 0.

Iterate from op1 to op2, incrementing op1 on each iteration. • If op1 is equal to op3, set X to 2. Otherwise, set X to 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:

```

1232 ; Assuming SetX and SetY are two memory operands of the same size (e.g., 32 bits)
1233 mov eax, SetX      ; Load SetX into EAX
1234 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:

```
1238 cmp dx, cx ; Compare DX and CX
1239 jbe L1 ; Jump to L1 if DX <= CX
1240
1241 cmp ax, cx ; Compare AX and CX (signed comparison)
1242 jg L2 ; Jump to L2 if AX > CX
1243
1244 and al, 0xFC ; Clear bits 0 and 1 in AL
1245 test al, al ; Test if AL is zero
1246 jz L3 ; Jump to L3 if AL is zero
1247 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:

```
1251 if( val1 > ecx ) AND ( ecx > edx )
1252     X = 1
1253 else
1254     X = 2;
```

Here's the corresponding assembly code:

```
1258 ; Assuming val1, ecx, edx, and X are 32-bit variables
1259 ; Also, assuming the condition is checked within a function
1260
1261 cmp dword [val1], ecx ; Compare val1 with ecx
1262 jle else_condition ; Jump to else_condition if val1 <= ecx
1263
1264 cmp ecx, edx ; Compare ecx with edx
1265 jle else_condition ; Jump to else_condition if ecx <= edx
1266
1267 mov dword [X], 1 ; Set X to 1 if both conditions are met
1268 jmp done ; Jump to done to skip the else block
1269
1270 else_condition:
1271 mov dword [X], 2 ; Set X to 2 if conditions are not met
1272
1273 done:
1274 ; 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 ($\text{val1} > \text{ecx}$ and $\text{ecx} > \text{edx}$), 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 8:

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.

Exercise 9:

Implement the following pseudocode using short-circuit evaluation:

```
1307 if( ebx > ecx AND ebx > edx) OR ( edx > eax )
1308     X = 1
1309 else
1310     X = 2
```

Here's the corresponding assembly code:

```
1314 ; Assuming ebx, ecx, edx, eax, and X are 32-bit variables
1315 ; Also, assuming the condition is checked within a function
1316
1317 ; Check the first condition: ebx > ecx AND ebx > edx
1318 cmp ebx, ecx          ; Compare ebx with ecx
1319 jle check_second_condition ; If ebx <= ecx, skip the second condition
1320
1321 cmp ebx, edx          ; Compare ebx with edx
1322 jle check_second_condition ; If ebx <= edx, skip the second condition
1323
1324 ; If we reach here, both conditions are met, so set X to 1
1325 mov dword [X], 1
1326 jmp done
1327
1328 check_second_condition:
1329 ; Check the second condition: edx > eax
1330 cmp edx, eax          ; Compare edx with eax
1331 jle set_X_to_2          ; If edx <= eax, set X to 2 and skip to done
1332
1333 ; If we reach here, the second condition is met, so set X to 1
1334 mov dword [X], 1
1335 jmp done
1336
1337 set_X_to_2:
1338 ; If neither condition is met, set X to 2
1339 mov dword [X], 2
1340
1341 done:
1342 ; 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.

Now, we have one more exercise remaining:

Exercise 10:

Implement the following pseudocode using short-circuit evaluation:

```
1347 while N > 0
1348     if N != 3 AND (N < A OR N > B)
1349         N = N - 2
1350     else
1351         N = N - 1
```

Here's the corresponding assembly code:

```
1356 ; Assuming N, A, B are 32-bit signed integers
1357 ; Also, assuming this code is part of a larger program
1358 while_loop:
1359     cmp dword [N], 0      ; Compare N with 0
1360     jle end_while        ; If N <= 0, exit the loop
1361     cmp dword [N], 3      ; Compare N with 3
1362     je skip_decrement    ; If N == 3, skip to else (decrement by 1)
1363     ; Check the second condition: N < A OR N > B
1364     cmp dword [N], [A]      ; Compare N with A
1365     jge else_decrement    ; If N >= A, skip to else (decrement by 1)
1366     cmp dword [N], [B]      ; Compare N with B
1367     jle else_decrement    ; If N <= B, skip to else (decrement by 1)
1368     ; If neither condition is met, decrement by 2
1369     sub dword [N], 2
1370     jmp continue_while    ; Continue the loop
1371 skip_decrement:
1372     ; N == 3, decrement by 1
1373     sub dword [N], 1
1374     jmp continue_while
1375 else_decrement:
1376     ; If any condition is met, decrement by 1
1377     sub dword [N], 1
1378 continue_while:
1379     ; Loop back to the beginning of the while loop
1380     jmp while_loop
1381 end_while:
1382 ; 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.

In the provided assembly code, `continue_while`, `end_while`, and `else_decrement` are not procedures or labels that you explicitly call; they are labels used for control flow within the while loop. Let me clarify their roles:

continue_while: This is a label used to mark the point in the code where the loop should continue if none of the conditions for decrementing N are met. It is not a procedure or function that you explicitly call; instead, it's a reference point for the code to jump back to the beginning of the while loop.

end_while: This is also a label used to mark the end of the while loop. When the condition $N \leq 0$ is met, the code jumps to this label to exit the loop and continue with the rest of the program.

else_decrement: This label is used to mark the point in the code where N is decremented by 1 when the condition $N \neq 3$ AND $(N < A \text{ OR } N > B)$ is not met. It's used to skip the decrement by 2 and directly decrement by 1.

These labels are simply reference points in the code to control the flow of execution within the while loop. You don't explicitly call them as procedures or functions. Instead, the program flow jumps to these labels based on conditional branches (`jmp`) to implement the logic of the while loop.

I hope this helps!

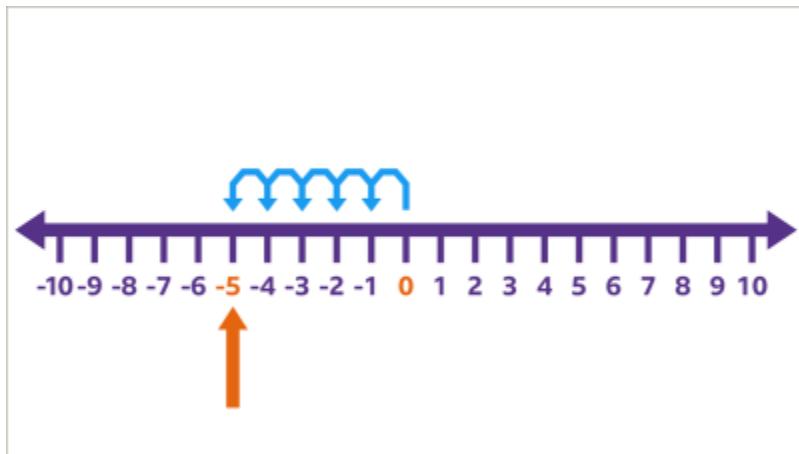
FINAL QUESTIONS FOR THIS TOPIC ON CONDITIONAL PROCESSING

I can explain the suggestions provided for testing your code:

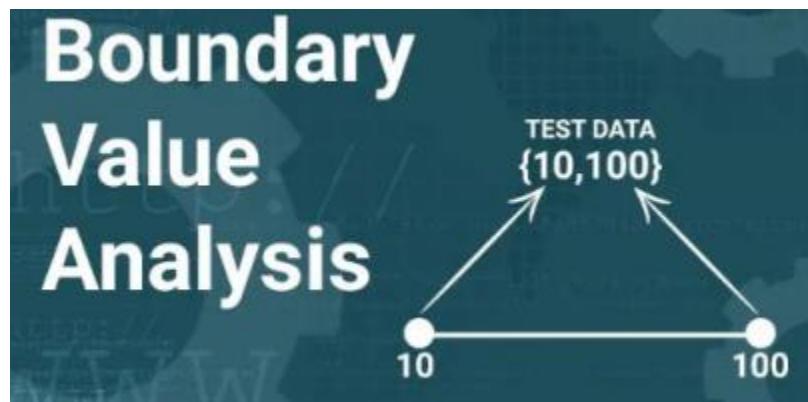
Using a Debugger: Debugging your code using a debugger is a crucial step in ensuring its correctness. Debuggers allow you to step through your code line by line, inspect variables, and identify issues. You can use debugging tools provided by your development environment (e.g., Visual Studio Debugger).



Testing with Negative Values: If your code deals with signed data, it's essential to include negative values in your test cases to cover all possible scenarios.



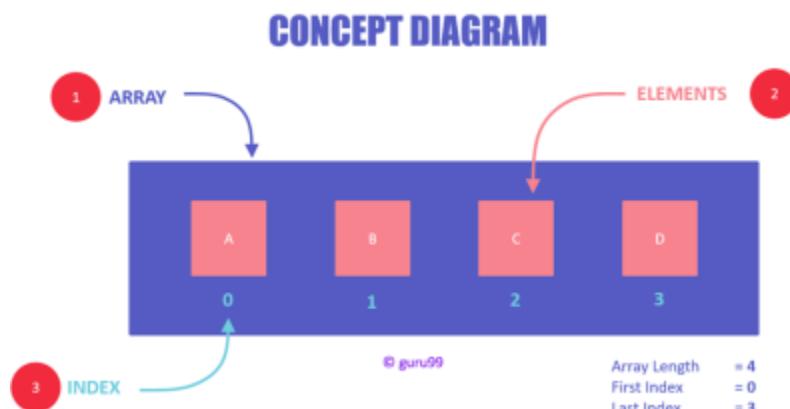
Testing at Boundaries: When a range of input values is specified, test your code with values that fall before, on, and after these boundaries. This helps verify how your code handles edge cases.



Multiple Test Cases: Create multiple test cases with different inputs and conditions. This ensures that your code is robust and can handle a variety of scenarios.

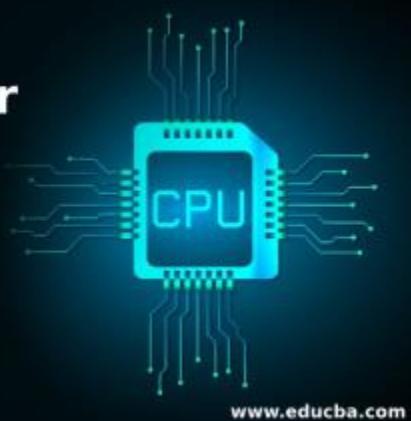


Using a Debugger for Array Operations: When working with arrays, especially when modifying them, a debugger's Memory window can be very useful. It allows you to inspect the array's contents in hexadecimal or decimal representation.

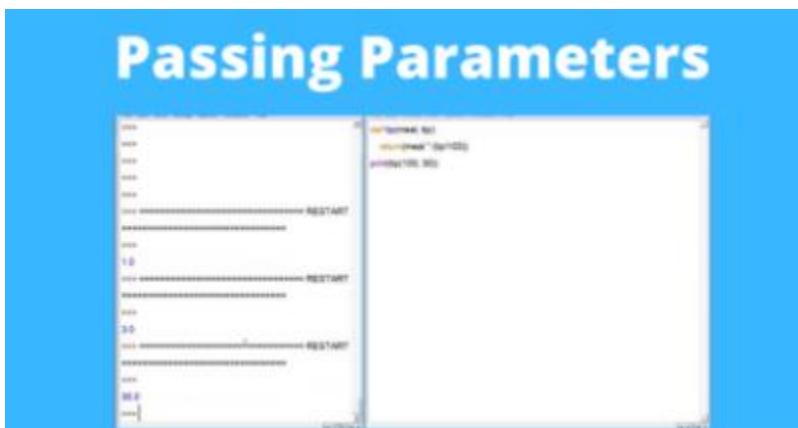


Checking Register Preservation: If you have a procedure that modifies registers, consider calling it twice in a row. This helps verify that the procedure correctly preserves register values between calls.

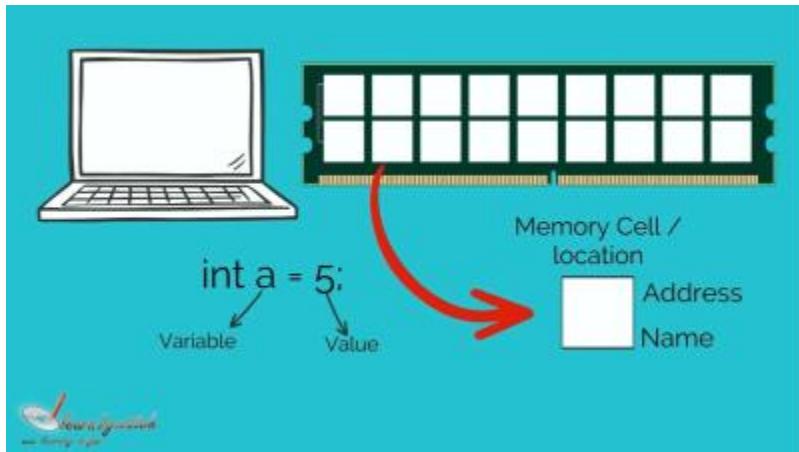
What is CPU Register



Parameter Passing for Multiple Arrays: When passing multiple arrays to a procedure, it's a good practice not to refer to arrays by name inside the procedure. Instead, set registers like ESI or EDI to the offsets of the arrays before calling the procedure. Use indirect addressing ([esi] or [edi]) inside the procedure to access array elements.



Local Variables in Procedures: If you need to create variables for use only within a procedure, you can declare them using the .data directive before the variable and the .code directive afterward. Initialize these variables within the procedure to ensure they start with the correct values when the procedure is called multiple times.



=====

Exercise 1: 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:

```

1386 .data
1387 array DWORD 10 DUP (?) ; Define an array to hold the random integers
1388 .code
1389 FillArray PROC
1390     ; Parameters:
1391     ; edi = pointer to the array
1392     ; ecx = N (number of elements)
1393     ; ebx = j (lower bound)
1394     ; edx = k (upper bound) || you can initialize random number generator (optional)
1395     call InitializeRandom
1396     ; Loop to fill the array with random numbers
1397     fill_loop:
1398         mov eax, ebx          ; Load lower bound (j) into eax
1399         sub eax, 1           ; Subtract 1 to make j inclusive
1400         add eax, edx         ; Calculate the range (k - j + 1)
1401         call GetRandom       ; Get a random number in [0, range)
1402         add eax, ebx         ; Add j to the random number to fit [j, k]
1403         mov [edi], eax       ; Store the random number in the array
1404         add edi, 4            ; Move to the next element
1405         loop fill_loop       ; Repeat for N elements
1406     ret
1407 FillArray ENDP
1408 main:
1409     ; Usage example:
1410     mov edi, OFFSET array ; Pointer to the array
1411     mov ecx, 10             ; N = 10 elements
1412     mov ebx, 1               ; Lower bound (j)
1413     mov edx, 100             ; Upper bound (k)
1414     call FillArray
1415     ; Call FillArray again with different j and k values if needed, Verify the results using a debugger
1416     ; (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 1: 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:

```
1421 .data
1422     array SDWORD 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ; Example array of signed doublewords
1423 .code
1424 SumInRange PROC
1425     ; Parameters:
1426     ;    edi = pointer to the array
1427     ;    ecx = size of the array
1428     ;    ebx = j (lower bound)
1429     ;    edx = k (upper bound)
1430     xor eax, eax           ; Clear EAX to store the sum
1431     sum_loop:
1432         mov esi, [edi]      ; Load the next element into ESI
1433         cmp esi, ebx        ; Compare with lower bound (j)
1434         jl not_in_range    ; Jump if less than j
1435         cmp esi, edx        ; Compare with upper bound (k)
1436         jg not_in_range    ; Jump if greater than k
1437         add eax, esi        ; Add to the sum
1438     not_in_range:
1439         add edi, 4          ; Move to the next element
1440         loop sum_loop      ; Repeat for all elements
1441     ret
1442 SumInRange ENDP
1443 main:                   ; Usage example:
1444     mov edi, OFFSET array ; Pointer to the array
1445     mov ecx, 10            ; Size of the array
1446     mov ebx, 2              ; Lower bound (j)
1447     mov edx, 7              ; Upper bound (k)
1448     call SumInRange
1449     ; The sum will be in the EAX register
1450     ; Call SumInRange again with different j and k values if needed
1451     ; 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 1: 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:

```
1455 .data
1456     grade CHAR ? ; Variable to store the grade
1457 .code
1458 CalcGrade PROC
1459     ; Parameter:
1460     ;    eax = integer value between 0 and 100
1461     cmp eax, 0
1462     jl invalid_input      ; Input is less than 0, return 'F'
1463     cmp eax, 60
1464     jl grade_F            ; Input is less than 60, return 'F'
1465     cmp eax, 70
1466     jl grade_D            ; Input is less than 70, return 'D'
1467     cmp eax, 80
1468     jl grade_C            ; Input is less than 80, return 'C'
1469     cmp eax, 90
1470     jl grade_B            ; Input is less than 90, return 'B'
1471     grade_A:
1472         mov al, 'A'        ; Input is 90 or greater, return 'A'
1473         jmp done
1474     grade_B:
1475         mov al, 'B'        ; Input is between 80 and 89, return 'B'
1476         jmp done
1477     grade_C:
1478         mov al, 'C'        ; Input is between 70 and 79, return 'C'
1479         jmp done
1480     grade_D:
1481         mov al, 'D'        ; Input is between 60 and 69, return 'D'
1482         jmp done
1483     grade_F:
1484         mov al, 'F'        ; Input is between 0 and 59, return 'F'
```

```

1485
1486     invalid_input:
1487         mov al, '?'           ; Invalid input, return '?'
1488
1489     done:
1490         ret
1491 CalcGrade ENDP
1492
1493 main:
1494     ; Usage example:
1495     mov eax, 85            ; Input value (test score)
1496     call CalcGrade
1497
1498     ; The grade will be in the AL register
1499
1500    ; 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.

Now it's time for you to do your own practice:

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.