**CHAPTER 6 NOTES**

**Conditional Branching (Simplified)**

* **Conditional Branching** allows programs to make decisions based on certain conditions, changing how the program runs.
* This is done with **IF** statements, **switch** statements, and loops in high-level languages. Assembly also supports this by letting you manipulate control flow.
* Important for tasks like **bit manipulation** in hardware control, **data encryption**, and **compression**.
* Key Questions Covered:
  + How to use **AND**, **OR**, **NOT** (from Chapter 1)?
  + How to write **IF** statements in assembly?
  + How do **nested IF** statements work in machine code?
  + How to change individual **bits** in numbers?
  + How to perform **binary data encryption**?
  + How are **signed** vs **unsigned** numbers handled in logic?
* This chapter goes from basics (binary logic) to how the **CPU** processes these instructions using **CMP** (comparison) and flags.

**6.2 Boolean and Comparison Instructions (Simplified)**

**Boolean Operations**:

* Basic operations **AND**, **OR**, **XOR**, **NOT** work at the **bit level** in assembly language. These are used to manage hardware, communication, and encryption.
* Flags help track the result of operations:
  + **Zero Flag**: Set when result is 0.
  + **Carry Flag**: Set when there’s a carry-out from the highest bit.
  + **Sign Flag**: Indicates if the number is negative or positive.
  + **Overflow Flag**: Indicates overflow in the result.
  + **Parity Flag**: Indicates even or odd number of 1 bits.

**6.2.2 AND Instruction (Simplified)**

* **AND** performs a bitwise AND operation between two operands (e.g., numbers or memory).
* **Operation**: Result is 1 only if both bits are 1.
* Example: **AND** is used to clear specific bits in a number without changing others (like masking).
* **Flags**: **AND** clears **Overflow** and **Carry** flags, and changes **Sign**, **Zero**, and **Parity** flags based on the result.

**Example**:

mov al, 10101110b

and al, 11110110b ; Clears bits 0 and 3 in AL

**Converting Characters to Uppercase:**

* **AND** is used to convert lowercase letters to uppercase by clearing bit 5 in ASCII values.

**6.2.3 OR Instruction (Simplified)**

* **OR** performs a bitwise OR operation, setting a bit to 1 if any of the bits is 1.
* **Operation**: If at least one bit is 1, the result is 1.
* Example: **OR** is used to set specific bits without affecting others.

**Flags**: **OR** clears **Carry** and **Overflow** flags, and modifies **Sign**, **Zero**, and **Parity** flags depending on the result.

**Example**:

or al, 00000100b ; Sets bit 2 in AL

**6.2.4 Bit-Mapped Sets**

* **Bit Maps**: Use binary bits to represent whether an item belongs to a set.
  + Example: A number like 10000000 00000000 00000000 00000111 shows which items are in a set.
* **Set Operations**:
  + **Complement**: **NOT** reverses all bits to get the opposite set.
  + **Intersection**: **AND** finds common elements between two sets.
  + **Union**: **OR** combines two sets.

**Example** of Intersection:

mov eax, SetX

and eax, SetY ; Finds the intersection of SetX and SetY

**6.2.5 XOR Instruction (Simplified)**

* **XOR** (Exclusive-OR) compares two bits: if they are the same (both 0 or both 1), the result is 0; if they are different, the result is 1.
* **Reversible**: Applying **XOR** twice on the same bits brings back the original values, which is useful for encryption.

**Flags**: **XOR** clears **Overflow** and **Carry** flags, and modifies **Sign**, **Zero**, and **Parity** flags based on the result.

**6.2.8 CMP Instruction**

* **CMP** is used to **compare integers** (works with character codes too) and performs an **implied subtraction** between the destination and source operands without changing their values.
* The **CMP** instruction updates various **flags** (Overflow, Sign, Zero, Carry, Auxiliary Carry, Parity) based on the result of this subtraction:
  + For **unsigned** values:
    - **ZF (Zero Flag)** = 1 if both operands are equal.
    - **CF (Carry Flag)** = 1 if the destination is less than the source.
  + For **signed** values:
    - **SF (Sign Flag)** ≠ **OF (Overflow Flag)** means the destination is smaller than the source.
    - **ZF (Zero Flag)** = 1 if the operands are equal.
* **CMP** is often used before **conditional jump instructions**, similar to **IF** statements in higher-level programming.

**Examples:**

* Comparing AX = 5 and 10 sets **CF** (because 5 < 10).
* Comparing 1000 and 1000 sets **ZF** (because they are equal).
* Comparing 105 and 0 clears both **ZF** and **CF** (because 105 > 0).

**6.2.9 Setting and Clearing CPU Flags**

You can **set** or **clear** the flags (Zero, Sign, Carry, Overflow) in different ways:

* **Zero Flag (ZF)**:
  + Set by TEST or AND with 0.
  + Clear by OR with 1.
* **Sign Flag (SF)**:
  + Set by OR with 0x80 (highest bit).
  + Clear by AND with 0x7F (clearing highest bit).
* **Carry Flag (CF)**:
  + Set by STC (Set Carry).
  + Clear by CLC (Clear Carry).
* **Overflow Flag (OF)**:
  + Set by adding two positive numbers that give a negative result.
  + Clear by OR with 0.

**6.2.10 Boolean Instructions in 64-Bit Mode**

* In **64-bit mode**, instructions often work similarly to 32-bit mode, with some differences:
  + If a constant is smaller than 32 bits and you’re working with a 64-bit operand, all bits in the destination are affected.
  + However, if the operand is 32 bits, only the lower 32 bits of the destination register are affected.

**Example in 64-bit mode:**

* mov rax, allones loads a 64-bit value into rax.
* Using AND on rax with different values affects rax depending on whether the operand is 64-bit or 32-bit.

In short, the CMP instruction is crucial for comparison and conditional logic in assembly, and flags can be controlled to guide program flow.

**Conditional Jumps in x86 Assembly**

**6.3 Conditional Jumps**

Conditional jumps are used to make decisions based on flags that are set during operations like comparisons. These jumps either happen or don’t based on the status of certain CPU flags.

**6.3.1 Conditional Structures**

In x86 assembly, there are no direct high-level logic structures like if or while. Instead, you combine comparison operations (like CMP, AND, or SUB) with jump instructions to achieve conditional logic.

There are two steps involved:

1. **Comparison Operation:** An operation like CMP sets flags in the CPU.
2. **Conditional Jump:** A jump is triggered based on those flags.

**Example 1:**

cmp eax, 0 ; Compare EAX with 0

jz L1 ; Jump to L1 if Zero Flag (ZF) is set

* The CMP compares EAX to 0.
* If EAX is zero, the Zero Flag (ZF) is set, and jz jumps to L1.

**Example 2:**

and dl, 10110000b ; Perform bitwise AND on DL

jnz L2 ; Jump to L2 if Zero Flag (ZF) is not set

* The AND operation may clear or set the Zero Flag.
* If ZF is clear, meaning the result is not zero, the jnz (jump if not zero) instruction jumps to L2.

**6.3.2 Jcond Instruction**

Conditional jump instructions, such as JZ, JNZ, JC, JNC, etc., allow branching to a destination label based on the condition of a CPU flag. If the condition is true, the jump happens; if false, the program continues executing the next instruction.

**Syntax:**

Jcond destination

Where cond refers to the condition (like ZF, CF, etc.). Examples include:

* JC: Jump if carry (Carry flag is set)
* JZ: Jump if zero (Zero flag is set)
* JNZ: Jump if not zero (Zero flag is clear)

**Example:**

cmp eax, 5

je L1 ; Jump if Equal (Zero flag set)

* If EAX equals 5, the CMP instruction sets the Zero flag, and JE (Jump if Equal) jumps to L1.

**6.3.3 Types of Conditional Jump Instructions**

Conditional jumps can be categorized into four groups:

1. **Jumps based on specific flag values:** These checks include flags like Zero (ZF), Carry (CF), Overflow (OF), etc.
2. **Jumps based on equality comparisons:** For example, JE (Jump if Equal) and JNE (Jump if Not Equal).
3. **Jumps based on unsigned comparisons:** These compare operands without considering signs.
4. **Jumps based on signed comparisons:** These are based on whether the operands are greater or less, considering their sign.

**Jumps Based on Specific Flag Values:**

| **Mnemonic** | **Description** | **Flag Condition** |
| --- | --- | --- |
| JZ | Jump if Zero | ZF = 1 |
| JNZ | Jump if Not Zero | ZF = 0 |
| JC | Jump if Carry | CF = 1 |
| JNC | Jump if No Carry | CF = 0 |
| JO | Jump if Overflow | OF = 1 |
| JNO | Jump if No Overflow | OF = 0 |

**Equality Comparisons:**

* **JE**: Jump if Equal (Zero flag is set)
* **JNE**: Jump if Not Equal (Zero flag is clear)

**Example:**

mov edx, 0A523h

cmp edx, 0A523h

jne L5 ; Jump if not equal (Zero flag set, no jump)

je L1 ; Jump if equal (Zero flag set, jump)

**6.3.4 Conditional Jump Applications**

**Testing Status Bits**

One common use of conditional jumps is testing specific bits in a register. For example, checking if bit 5 of a status byte is set, you can use the TEST instruction to examine the bit without modifying the original value:

mov al, status

test al, 00100000b

jnz DeviceOffline ; Jump if bit 5 is set (device offline)

**Finding the Larger of Two Integers:**

To compare two integers in registers EAX and EBX and store the larger one in EDX:

mov edx, eax

cmp eax, ebx

jae L1 ; Jump if EAX >= EBX

mov edx, ebx ; If not, move EBX to EDX

L1:

**Finding the Smallest of Three Integers:**To compare three integers (V1, V2, V3) and store the smallest in AX:

mov ax, V1

cmp ax, V2

jbe L1 ; Jump if V1 <= V2

mov ax, V2

L1:

cmp ax, V3

jbe L2 ; Jump if V2 <= V3

mov ax, V3

L2:

**Loop Until Key Pressed:**

To loop continuously until a key is pressed, you can use a flag to check for a key in the buffer:

L1:

mov eax, 10 ; Delay

call Delay

call ReadKey

jz L1 ; Repeat if no key is pressed

mov char, AL ; Save the key pressed

**Section Review**

 **Which jump instructions follow unsigned integer comparisons?**

* Jump instructions that follow unsigned integer comparisons include:
  + **JC** (Jump if Carry)
  + **JNC** (Jump if No Carry)
  + **JZ** (Jump if Zero)
  + **JNZ** (Jump if Not Zero)
  + **JA** (Jump if Above)
  + **JAE** (Jump if Above or Equal)
  + **JB** (Jump if Below)
  + **JBE** (Jump if Below or Equal)

 **Which jump instructions follow signed integer comparisons?**

* Jump instructions that follow signed integer comparisons include:
  + **JS** (Jump if Sign)
  + **JNS** (Jump if Not Sign)
  + **JE** (Jump if Equal)
  + **JNE** (Jump if Not Equal)
  + **JG** (Jump if Greater)
  + **JGE** (Jump if Greater or Equal)
  + **JL** (Jump if Less)
  + **JLE** (Jump if Less or Equal)

 **Which conditional jump instruction is equivalent to JNAE?**

* The **JNAE** (Jump if Not Above or Equal) instruction is equivalent to **JB** (Jump if Below) for unsigned comparisons.

 **Which conditional jump instruction is equivalent to the JNA instruction?**

* The **JNA** (Jump if Not Above) instruction is equivalent to **JBE** (Jump if Below or Equal) for unsigned comparisons.

 **Which conditional jump instruction is equivalent to the JNGE instruction?**

* The **JNGE** (Jump if Not Greater or Equal) instruction is equivalent to **JL** (Jump if Less) for signed comparisons.

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

* **Code:**

mov ax, 8109h

cmp ax, 26h

jg Target

* **Answer:** **No**
  + Explanation: The comparison between ax (8109h) and 26h results in a greater condition when interpreted as unsigned integers, but in signed integers, 8109h is a negative value (because it’s greater than 7FFFh, which marks the sign bit for signed integers), so the condition would not be met for a signed comparison.

**6.4 Conditional Loop Instructions**

**6.4.1 LOOPZ and LOOPE Instructions**

* **LOOPZ** (Loop if Zero) works like the regular **LOOP** instruction but with an extra condition: it only loops if the Zero Flag (ZF) is set (meaning the previous operation resulted in zero).
* **Syntax:** LOOPZ destination
* **LOOPE** is the same as **LOOPZ**, and they share the same opcode.

**What they do:**

* ECX is the counter. After each iteration, ECX is decreased by 1.
* If **ECX > 0** and **ZF = 1**, the loop jumps to the specified destination.
* If not, the loop stops, and the next instruction is executed.

**Important Notes:**

* They **do not change any status flags**.
* In **32-bit mode**, **ECX** is used as the counter.
* In **64-bit mode**, **RCX** is the counter.

**6.4.2 LOOPNZ and LOOPNE Instructions**

* **LOOPNZ** (Loop if Not Zero) is the opposite of **LOOPZ**. The loop will continue if **ECX > 0** (after it’s decremented) and **ZF = 0** (meaning the result of the last operation was not zero).
* **Syntax:** LOOPNZ destination
* **LOOPNE** is the same as **LOOPNZ**, and they share the same opcode.

**What they do:**

* ECX is decremented by 1 each time.
* If **ECX > 0** and **ZF = 0**, it jumps to the specified destination.
* If not, the loop stops, and control passes to the next instruction.

**Example (LOOPNZ in Action):**

The example demonstrates how **LOOPNZ** can be used to scan through an array of numbers, looking for a non-negative value (when the sign bit is clear).

.data

array SWORD -3,-6,-1,-10,10,30,40,4

sentinel SWORD 0

.code

mov esi, OFFSET array ; Load address of array into ESI

mov ecx, LENGTHOF array ; Set the loop counter (ECX) to the length of the array

L1:

test WORD PTR [esi], 8000h ; Check the sign bit (if it's set, the number is negative)

pushfd ; Push flags to stack before modifying them

add ; Modify flags (add operation)

popfd ; Restore flags from stack

esi, TYPE array ; Move ESI to the next element in the array

loopnz L1 ; Continue looping if ECX > 0 and ZF = 0 (non-zero result)

; If a non-negative value is found, ESI will point to it. If none found, the loop ends when ECX reaches zero.

**How It Works:**

* The loop goes through each element of the array.
* It checks if the number is non-negative by testing the **sign bit**.
* If a non-negative number is found, **ESI** points to it.
* If no positive number is found, the loop ends when **ECX** reaches zero.
* After finishing, if no positive number is found, a **JNZ** (Jump if Not Zero) instruction will jump to **quit** and **ESI** will point to the sentinel value (0), which marks the end of the array.

**Section Review**

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

**Answer: False.**  
The **LOOPE** (Loop if Equal) instruction jumps when the Zero flag is **set**, not cleared. It’s the **LOOPZ** instruction that requires the Zero flag to be set.

1. **(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.**

**Answer: True.**  
**LOOPNZ** (Loop if Not Zero) will jump when **ECX > 0** and the **Zero flag (ZF) is clear**.

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

**Answer: True.**  
In **x86 assembly**, the destination of the **LOOPZ** (or **LOOPE**) instruction must be within a signed 8-bit offset, which limits the distance to 128 or 127 bytes.

1. **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.**

**Modified Code:**

.data

array SWORD 5, 6, 3, 10, -10, -30, -40, -4 ; Now starts with positive values

sentinel SWORD 0

.code

mov esi, OFFSET array ; Load address of array into ESI

mov ecx, LENGTHOF array ; Set the loop counter (ECX) to the length of the array

L1:

test WORD PTR [esi], 8000h ; Check the sign bit (if it's set, the number is negative)

pushfd ; Push flags to stack before modifying them

add ; Modify flags (add operation)

popfd ; Restore flags from stack

esi, TYPE array ; Move ESI to the next element in the array

loopnz L1 ; Continue looping if ECX > 0 and ZF = 0 (non-zero result)

; Now ESI will point to the first negative value in the array.

**Explanation:**  
The array now starts with positive values, and we modified the code so that it scans for the first negative number. This is done by checking the sign bit in the **test** instruction, which tells if the number is negative (if the sign bit is set).

1. **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?**

**Answer:**  
Without a sentinel value, if the array contains no positive numbers, the loop will continue to decrement **ECX** until it becomes zero. The program would eventually exit the loop, but **ESI** would point to the last element in the array, which might not be useful or intended. Additionally, there would be no clear "end" of the search, making it harder to determine whether the loop completed successfully or found a positive number. Removing the sentinel would also risk running out of bounds if not properly handled, which could lead to incorrect results or memory access errors.

**Conditional Structures**

**Conditional structures** are used to make decisions in code. They test conditions and execute different instructions based on whether the condition is true or false.

**6.5.1 Block-Structured IF Statements**

An **IF** statement checks a boolean expression and then executes one set of instructions if the condition is true, and another if it's false. For example:

if( op1 == op2 )

{

X = 1;

Y = 2;

}

In assembly language, this is done with a **CMP** instruction to compare values, followed by conditional **JUMP** instructions to choose which set of instructions to execute.

Example translation:

mov eax, op1 ; Move op1 into register eax

cmp eax, op2 ; Compare op1 and op2

jne L1 ; Jump if not equal

mov X, 1 ; If op1 == op2, set X to 1

mov Y, 2 ; Set Y to 2

L1: ; Label for the jump

**Example 2: Setting Cluster Size**

In this case, we set the cluster size based on the volume size:

c++

CopyEdit

clusterSize = 8192;

if (terabytes < 16)

clusterSize = 4096;

**Assembly translation:**

mov eax, terabytes

cmp eax, 16

jae next

mov clusterSize, 4096

next:

mov clusterSize, 8192

**Example 3: Conditional Branching with Calls**

This example calls different routines depending on a condition:

c++

if (op1 > op2)

call Routine1;

else

call Routine2;

**Assembly translation:**

mov eax, op1

cmp eax, op2

jg A1

call Routine2

jmp A2

A1: call Routine1

A2:

**White Box Testing**

**White box testing** is used to verify code behavior by manually testing all possible input combinations and verifying the execution path.

**Example of Nested IF**

Here is an example of a nested IF statement:

**mov eax, op1**

**cmp eax, op2**

**jne L2**

**mov eax, X**

**cmp eax, Y**

**jg L1**

**call Routine2**

**jmp L3**

**L1: call Routine1**

**L3: jmp L3**

**L2: call Routine3**

**if (op1 == op2)**

**if (X > Y)**

**call Routine1;**

**else**

**call Routine2;**

**else**

**call Routine3;**

**Assembly translation:------------------------🡪**

**6.5.2 Compound Expressions**

**Logical AND Operator**

For compound boolean expressions, **AND** is used to check if both conditions are true.

Example:

c++

if (al > bl) AND (bl > cl)

X = 1;

**Assembly translation using short-circuit evaluation:**

**cmp al, bl**

**ja L1**

**jmp next**

**L1: cmp bl, cl**

**ja L2**

**jmp next**

**L2: mov X, 1**

**Logical OR Operator**

For **OR**, the expression is true if any condition is true.

Example:

**Cmp al,bl**

**Ja L1**

**Cmp bl,cl**

**Jb next**

**L1:movx,1**

**Next:**

**Movx,1**

c++

if (al > bl) OR (bl > cl)

X = 1;

Assembly translation:

assembly

CopyEdit

cmp al, bl

ja L1

cmp bl, cl

jbe next

L1: mov X, 1

next:

**6.5.3 WHILE Loops**

A **WHILE loop** repeatedly executes a block of code as long as the condition is true.

Example:

c++

**Mov eax,var1**

**While:**

**Cmp eax,var2**

**Jnl end**

**Inc eax**

**Dec val2**

**Jmp while**

**End:**

**Mov va1,eax**

while (val1 < val2)

{

val1++;

val2--;

}

Assembly translation:

mov eax, val1

beginwhile:

cmp eax, val2

jnl endwhile

inc eax

dec val2

jmp beginwhile

endwhile:

mov val1, eax

**IF Nested in a WHILE Loop**

An **IF** statement inside a **WHILE loop** can be translated into assembly as well. Here's a loop that sums array elements greater than a given value:

c++

while (index < ArraySize)

{

if (array[index] > sample)

sum += array[index];

index++;

}

**Assembly translation:**

mov eax, 0 ; Initialize sum

mov edx, sample ; Set sample value

mov esi, 0 ; Initialize index

mov ecx, ArraySize ; Set array size

L1:

cmp esi, ecx ; Check if index < ArraySize

jl L2

jmp L5

L2:

cmp array[esi\*4], edx ; Check if array[index] > sample

jg L3

jmp L4

L3:

add eax, array[esi\*4] ; Add to sum

L4:

inc esi ; Increment index

jmp L1

L5:

mov sum, eax ; Store result in sum

**Table-Driven Selection**

**Table-driven selection** is a method where a table lookup is used instead of multiple if-else or switch-case conditions. The idea is to create a table with values and corresponding actions (like procedure addresses). Then, you loop through this table to perform the correct action based on a condition.

**How it works:**

1. **Create the table**: The table contains lookup values (like characters or numbers) and the addresses of the corresponding procedures.
2. **Search the table**: Use a loop to search through the table. For each entry, compare the current value to the lookup value.
3. **Call the procedure**: Once a match is found, call the procedure linked to the corresponding address in the table.

This method is most useful when there are many comparisons, as it can simplify your code and reduce its size.

**Example of a Lookup Table:**

Let's take a table with single-character lookup values and corresponding procedure addresses:

CaseTable BYTE 'A'

DWORD Process\_A ; address of Procedure A

BYTE 'B'

DWORD Process\_B ; address of Procedure B

BYTE 'C'

DWORD Process\_C ; address of Procedure C

BYTE 'D'

DWORD Process\_D ; address of Procedure D

Each character ('A', 'B', 'C', 'D') in the table maps to a procedure's address. The table is stored in memory, and the procedures are located at specific addresses (like 120h, 130h, 140h, 150h for Process\_A, Process\_B, Process\_C, and Process\_D respectively).

**Example Program:**

In the example program, the user inputs a character (like 'A', 'B', etc.), and the program looks through the table to find the corresponding procedure to call.

; Table-driven selection program

INCLUDE Irvine32.inc

.data

CaseTable BYTE 'A', DWORD Process\_A

BYTE 'B', DWORD Process\_B

BYTE 'C', DWORD Process\_C

BYTE 'D', DWORD Process\_D

prompt BYTE "Press capital A,B,C,or D: ", 0

; Procedure messages

msgA BYTE "Process\_A", 0

msgB BYTE "Process\_B", 0

msgC BYTE "Process\_C", 0

msgD BYTE "Process\_D", 0

.code

main PROC

mov edx, OFFSET prompt

call WriteString

call ReadChar

mov ebx, OFFSET CaseTable

mov ecx, NumberOfEntries ; loop counter

L1:

cmp al, [ebx] ; compare user input (al) with table value

jne L2 ; if no match, continue checking the next entry

call NEAR PTR [ebx + 1] ; call the corresponding procedure

jmp L3

L2:

add ebx, EntrySize ; move to the next entry in the table

loop L1 ; repeat until we find a match or reach the end of the table

L3:

exit

main ENDP

; Procedures for each process

Process\_A PROC

mov edx, OFFSET msgA

ret

Process\_A ENDP

Process\_B PROC

mov edx, OFFSET msgB

ret

Process\_B ENDP

Process\_C PROC

mov edx, OFFSET msgC

ret

Process\_C ENDP

Process\_D PROC

mov edx, OFFSET msgD

ret

Process\_D ENDP

END main

**How It Works:**

1. The program prints a prompt asking the user to input a character (A, B, C, or D).
2. The program compares the user input with the entries in the CaseTable.
3. Once a match is found, it calls the procedure linked to that entry.
4. Each procedure sets the EDX register to the address of a message (like "Process\_A", "Process\_B", etc.), and the program displays the message.

**Advantages of Table-Driven Selection:**

* **Less code**: You don't need multiple if, switch, or case statements.
* **Scalability**: This approach works well when there are a large number of comparisons.
* **Flexibility**: The table can be modified or reconfigured at runtime.
* **Easier maintenance**: Modifying the table is easier than rewriting long comparison chains.

**6.5.5 Section Review Notes**

**Important Concept:**

* **Short-circuit evaluation** should be used in compound expressions. This means that the second condition is only evaluated if necessary, improving performance.

Assuming that val1 and X are 32-bit variables, here are the solutions to the review questions.

**1. Implement the following pseudocode in assembly language:**

**Pseudocode:**

if ebx > ecx

X = 1

**Assembly Code:**

cmp ebx, ecx ; Compare ebx and ecx

jle skip ; Jump to skip if ebx <= ecx (less than or equal)

mov X, 1 ; Set X = 1 if ebx > ecx

skip:

* cmp ebx, ecx compares the two registers.
* jle skip jumps over the mov instruction if the condition ebx <= ecx is true.
* mov X, 1 sets X to 1 only if ebx > ecx.

**2. Implement the following pseudocode in assembly language:**

**Pseudocode:**

if edx <= ecx

X = 1

else

X = 2

**Assembly Code:**

assembly

CopyEdit

cmp edx, ecx ; Compare edx and ecx

jg set\_X\_2 ; Jump to set\_X\_2 if edx > ecx

mov X, 1 ; Set X = 1 if edx <= ecx

jmp done ; Skip setting X = 2

set\_X\_2:

mov X, 2 ; Set X = 2 if edx > ecx

done:

* cmp edx, ecx compares the two registers.
* jg set\_X\_2 jumps to set X to 2 if edx > ecx (greater than).
* If edx <= ecx, X is set to 1.
* The program then jumps over the second part to avoid redundant checks.

**3. In the program from Section 6.5.4, why is it better to let the assembler calculate NumberOfEntries rather than assigning a constant such as NumberOfEntries = 4?**

* **Reasoning:**
  + Calculating NumberOfEntries allows the program to be more **flexible** and **scalable**. If the table is modified (e.g., more entries are added), the calculation will automatically update the number of entries without the need to manually change a constant value like 4.
  + It also ensures the program is less error-prone because the calculation reflects the actual size of the data, rather than requiring a manual update when changes are made.
  + The assembler can calculate the value dynamically, ensuring that the program works with any size table, even if the number of entries changes during runtime.

**4. Challenge: Rewrite the code from Section 6.5.3 so it is functionally equivalent, but uses fewer instructions.**

Let's simplify the code from Section 6.5.3. We will aim to optimize the loop and the comparison logic.

Original code:

main PROC

mov edx, OFFSET prompt

call WriteString

call ReadChar

mov ebx, OFFSET CaseTable

mov ecx, NumberOfEntries ; loop counter

L1:

cmp al, [ebx] ; compare user input (al) with table value

jne L2 ; if no match, continue checking the next entry

call NEAR PTR [ebx + 1] ; call the corresponding procedure

jmp L3

L2:

add ebx, EntrySize ; move to the next entry in the table

loop L1 ; repeat until we find a match or reach the end of the table

L3:

exit

main ENDP

Optimized version:

main PROC

mov edx, OFFSET prompt

call WriteString

call ReadChar

mov ebx, OFFSET CaseTable

mov ecx, NumberOfEntries ; loop counter

L1:

cmp al, [ebx] ; compare user input (al) with table value

je call\_proc ; jump if matched

add ebx, EntrySize ; move to the next entry

loop L1

jmp done

call\_proc:

call NEAR PTR [ebx + 1] ; call the corresponding procedure

done:

exit

main ENDP

**Explanation of Changes:**

* Instead of using a jne (jump if not equal) and then jumping to L3, we directly use je (jump if equal) to go to call\_proc when a match is found. This reduces one unnecessary jump.
* We simplify the code by removing the redundant jmp L3 and use jmp done at the end of the program for a cleaner exit path.

**6.6 Application: Finite-State Machines (FSM)**

A **Finite-State Machine (FSM)** is a system that changes its state based on input. It's like a flowchart with circles (nodes) and arrows (edges) showing how the system moves from one state to another.

* **States**: Represent different situations in the program.
* **Transitions**: Show how the system moves from one state to another based on input.

**Key Components of an FSM:**

* **Initial State**: The starting point, marked with an incoming arrow.
* **Terminal States**: States where the program might stop without errors, shown with a thick border.
* **Valid Input**: For example, a string must start with "x" and end with "z". Between these, it can have any letter from 'a' to 'y'. The FSM checks the input and transitions between states.

**6.6.1 Validating an Input String**

When a program checks if an input string is valid, an FSM reads the string one character at a time. It follows certain rules for valid transitions between states.

**Example:**

To validate a string:

* Start with "x".
* End with "z".
* Between them, any letter except "z" is allowed.

If the input string does not follow the rules, the FSM will detect an error.

**6.6.2 Validating a Signed Integer**

FSM can also validate numbers, like signed integers (e.g., +123 or -456). The FSM reads the sign (+ or -) and digits in sequence. If the input doesn’t match the expected pattern, an error is shown.

**FSM in Assembly Language**

FSMs are easy to convert into assembly language. Here's how:

1. Each state in the FSM is represented by a label.
2. The program reads the input character and checks which state to go to.
3. The program uses comparisons and conditional jumps to transition between states.

**Example Code Walkthrough:**

In assembly language, the FSM code follows this pattern:

1. **Get Input**: Read the next character.
2. **Check Transitions**: Compare the character to see if it matches the expected input for the current state.
3. **Error Handling**: If the input is not valid, show an error message and stop the program.

**Example FSM Program for a Signed Integer:**

The FSM program checks if a number has a sign (+ or -) and digits. If the input is valid, it moves to the next state; otherwise, it shows an error message.

* **State A**: Checks for a sign or a digit.
* **State B**: Checks if the next input is a digit.
* **State C**: Checks if the input is valid.

**IsDigit Procedure**

This function checks if a character is a digit (0-9). It compares the ASCII value of the character to the values for '0' and '9'. If the character is not a digit, the program jumps to an error state

**Section Review**

**1. A finite-state machine is a specific application of what type of data structure?**  
A finite-state machine (FSM) is a specific application of a **graph** data structure. It uses nodes (states) and edges (transitions between states).

**2. In a finite-state machine diagram, what do the nodes represent?**  
In a finite-state machine diagram, the **nodes** represent the different **states** that the machine can be in. Each state shows a specific condition or position in the process.

**3. In a finite-state machine diagram, what do the edges represent?**  
In a finite-state machine diagram, the **edges** represent the **transitions** between states. They show how the machine moves from one state to another based on input.

**6.7 Conditional Control Flow Directives**

**4. In the signed integer finite-state machine (Section 6.6.2), which state is reached when the input consists of “5”?**  
In the signed integer finite-state machine, when the input is “5”, the machine reaches the **state for processing digits**, where it expects more digits to form the full integer.

**5. In the signed integer finite-state machine (Section 6.6.2), how many digits can occur after a minus sign?**  
In the signed integer finite-state machine, **unlimited digits** can occur after a minus sign. The machine allows any number of digits to follow the sign.

**6. What happens in a finite-state machine when no more input is available and the current state is a nonterminal state?**  
When no more input is available and the current state is a **nonterminal state**, the finite-state machine is in an **invalid state**. This means the input doesn't lead to a valid output or final state, and the process might be considered incomplete or incorrect.

**6.7 Conditional Control Flow Directives (MASM)**

MASM provides conditional control flow directives to make coding easier, but they are only available in 32-bit mode, not 64-bit. These directives are processed before your code is assembled.

**6.7.1 Creating IF Statements**

MASM makes it simple to create conditional statements using .IF, .ELSE, .ELSEIF, and .ENDIF. These directives help generate comparison and jump instructions automatically. The basic syntax is:

sql

.IF condition

statements

[.ELSEIF condition

statements ]

[.ELSE

statements ]

.ENDIF

* .IF and .ENDIF are required, but .ELSEIF and .ELSE are optional.
* Conditions use boolean expressions like eax > 10000h, val1 <= 100, or val2 == eax.
* MASM generates the equivalent assembly code (e.g., cmp, jbe for unsigned comparison).

Example:

asm

CopyEdit

mov eax,6

.IF eax > val1

mov result,1

.ENDIF

This gets expanded to:

asm

CopyEdit

mov eax,6

cmp eax,val1

jbe @C0001

mov result,1

@C0001:

**6.7.2 Signed and Unsigned Comparisons**

MASM generates different jump instructions depending on whether the comparison is signed or unsigned.

* For **unsigned** variables, it uses instructions like jbe (jump if below or equal).
* For **signed** variables, it uses jle (jump if less or equal).

If two registers are compared, MASM assumes **unsigned** comparison unless otherwise specified.

**6.7.3 Compound Expressions**

You can use logical operators like || (OR) and && (AND) in .IF conditions.

Example:

asm

CopyEdit

.IF expression1 || expression2

statements

.ENDIF

**Example: SetCursorPosition**

A function to check if cursor coordinates (DH and DL) are within a valid range:

asm

CopyEdit

.IF (dl < 0) || (dl > 79)

mov edx, OFFSET BadXCoordMsg

call WriteString

jmp quit

.ENDIF

.IF (dh < 0) || (dh > 24)

mov edx, OFFSET BadYCoordMsg

call WriteString

jmp quit

.ENDIF

call Gotoxy

quit:

ret

**Example: College Registration**

A college registration check based on grade average and credits:

asm

CopyEdit

.IF gradeAverage > 350

mov OkToRegister, TRUE

.ELSEIF (gradeAverage > 250) && (credits <= 16)

mov OkToRegister, TRUE

.ELSEIF (credits <= 12)

mov OkToRegister, TRUE

.ENDIF

The assembler generates the appropriate code, checking conditions in sequence.

**6.7.4 Creating Loops with .REPEAT and .WHILE**

* **.REPEAT** executes the loop body first, then checks the condition:

asm

CopyEdit

.REPEAT

statements

.UNTIL condition

* **.WHILE** checks the condition first before executing the loop:

asm

CopyEdit

.WHILE condition

statements

.ENDW

**Example: Loop with .WHILE**

asm

CopyEdit

mov eax, 0

.WHILE eax < 10

inc eax

call WriteDec

call Crlf

.ENDW

**Example: Loop with .REPEAT**

asm

CopyEdit

mov eax, 0

.REPEAT

inc eax

call WriteDec

call Crlf

.UNTIL eax == 10

**Loop Containing an IF Statement**

Example of a while loop with an embedded if statement:

asm

CopyEdit

.data

X DWORD 0

op1 DWORD 2

op2 DWORD 4

op3 DWORD 5

.code

mov eax, op1

mov ebx, op2

mov ecx, op3

.WHILE eax < ebx

inc eax

.IF eax == ecx

mov X, 2

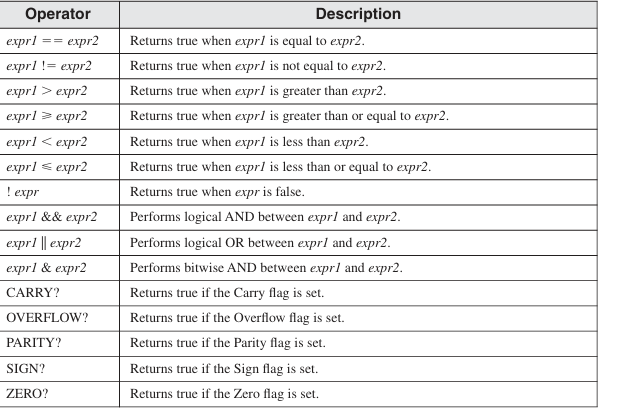
.ELSE

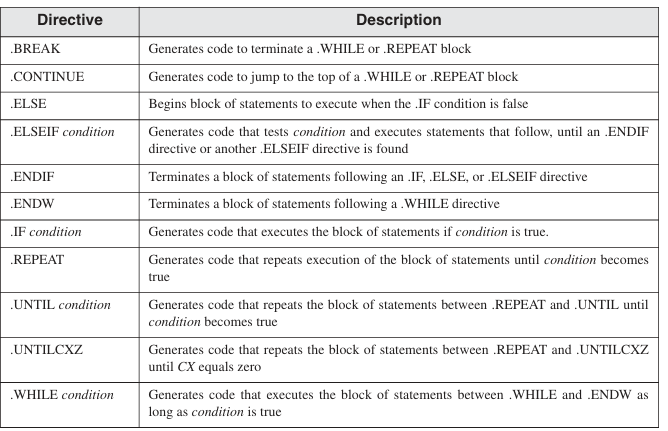
mov X, 3

.ENDIF

.ENDW

This example checks the value of eax and updates X depending on the condition.





**Review Questions:**

**1. mov bx, 0FFFFh; and bx, 6Bh**

* 0FFFFh in binary: 1111 1111 1111 1111
* 6Bh in binary: 0110 1011
* Performing AND operation:

yaml

CopyEdit

1111 1111 1111 1111

AND 0110 1011

---------------

0110 1011 (6Bh)

* **Result:** BX = 6Bh

**2. mov bx, 91BAh; and bx, 92h**

* 91BAh in binary: 1001 0001 1011 1010
* 92h in binary: 1001 0010
* Performing AND operation:

1001 0001 1011 1010

AND 0000 0000 1001 0010

---------------

0000 0000 1001 0010 (92h)

* **Result:** BX = 92h

**3. mov bx, 0649Bh; or bx, 3Ah**

* 0649Bh in binary: 0000 0110 0100 1001 1011
* 3Ah in binary: 0011 1010
* Performing OR operation:

0000 0110 0100 1001 1011

OR 0000 0000 0011 1010

---------------

0000 0110 0111 1011 (64B7h)

* **Result:** BX = 64B7h

**4. mov bx, 029D6h; xor bx, 8181h**

* 029D6h in binary: 0000 0010 1001 1101 0110
* 8181h in binary: 1000 0001 1000 0001
* Performing XOR operation:

yaml

CopyEdit

0000 0010 1001 1101 0110

XOR 1000 0001 1000 0001

---------------

1000 0011 0001 1100 0111 (8387h)

* **Result:** **BX = A857**

**5. mov ebx, 0AFAF649Bh; or ebx, 3A219604h**

* 0AFAF649Bh in binary: 0000 1010 1111 1010 1111 0110 0100 1001 1011
* 3A219604h in binary: 0011 1010 0010 0001 1001 0110 0000 0100
* Performing OR operation:

yaml

CopyEdit

0000 1010 1111 1010 1111 0110 0100 1001 1011

OR 0011 1010 0010 0001 1001 0110 0000 0100

---------------

0011 1110 1111 1011 1111 1110 0100 1101 (AFAF659Fh)

* **Result:** EBX = AFAF659Fh

**6. mov rbx, 0AFAF649Bh; xor rbx, 0FFFFFFFFh**

* 0AFAF649Bh in binary: 0000 1010 1111 1010 1111 0110 0100 1001 1011
* 0FFFFFFFFh in binary: 1111 1111 1111 1111 1111 1111 1111 1111
* Performing XOR operation:

yaml

CopyEdit

0000 1010 1111 1010 1111 0110 0100 1001 1011

XOR 1111 1111 1111 1111 1111 1111 1111 1111

---------------

1111 0100 0000 0000 0000 0000 0000 0000 (F5000000h)

* **Result:** RBX = F5000000h

**7. Showing the value of AL in binary after each instruction:**

* a. mov al, 01101111b; and al, 00101101b → Result: 00101101b (AL = 2Dh)
* b. mov al, 6Dh; and al, 4Ah → Result: 4Ah (AL = 4Ah)
* c. mov al, 00001111b; or al, 61h → Result: 01111111b (AL = 7Fh)
* d. mov al, 94h; xor al, 37h → Result: ABh (AL = ABh)

**8. Showing the value of AL in hexadecimal after each instruction:**

* a. mov al, 7Ah; not al → Result: 85h (AL = 85h)
* b. mov al, 3Dh; and al, 74h → Result: 34h (AL = 34h)
* c. mov al, 9Bh; or al, 35h → Result: BFh (AL = BFh)
* d. mov al, 72h; xor al, 0DCh → Result: 4Eh (AL = 4Eh)

**9. Showing the values of flags:**

* a. mov al, 00001111b; test al, 00000010b → CF = 0, ZF = 0, SF = 0
* b. mov al, 00000110b; cmp al, 00000101b → CF = 0, ZF = 0, SF = 0
* c. mov al, 00000101b; cmp al, 00000111b → CF = 1, ZF = 0, SF = 1

**10. Conditional jump instruction based on ECX:**

* **Answer:** JECXZ (Jump if ECX is zero)

**11. Effect of JA and JNBE on Zero and Carry flags:**

* **JA (Jump if Above)** and **JNBE (Jump if Not Below or Equal)** both check:
  + **Carry Flag (CF)** should be 0.
  + **Zero Flag (ZF)** should be 0.

**12. Final value of EDX:**

* EDX = 0 (Since eax is greater than or equal to 8000h)

**13. Final value of EDX:**

* EDX = 1 (Since eax is less than 8000h)

**14. Final value of EDX:**

* EDX = 0 (Since eax is less than FFFF8000h)

**15. True/False: Will jump to Target?**

* **Answer:** True (Since eax is greater than -50)

**16. True/False: Will jump to Target?**

* **Answer:** False (Since eax is not greater than 26)

**17. mov rbx, 0FFFFFFFFFFFFFFFFh; and rbx, 80h**

* **Result:** RBX = 80h

**18. mov rbx, 0FFFFFFFFFFFFFFFFh; and rbx, 808080h**

* **Result:** RBX = 808080h

**19. mov rbx, 0FFFFFFFFFFFFFFFFh; and rbx, 80808080h**

* **Result:** RBX = 80808080h