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:

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

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

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

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:

```
A - Add
B - Subtract
C - Multiply
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:

```
; mov eax, 1 ; add 1
; mov ebx, 2 ; add 2
; mov ecx, OFFSET CaseTable ; set the loop counter
; start the loop
L1:
cmp eax, CaseTable        ; compare the value in eax to the first lookup value in the table
je Add                    ; if equal, call the Add procedure
inc ecx                   ; increment the loop counter
cmp ecx, CaseTable + 4    ; check if the loop counter is greater than the size of the table
jge Done                  ; if greater than or equal, the loop is finished

jmp L1                    ; jump back to the beginning of the loop

Add:                      ; Add procedure
add eax, ebx
ret


Done:                     ; Done label
; 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.

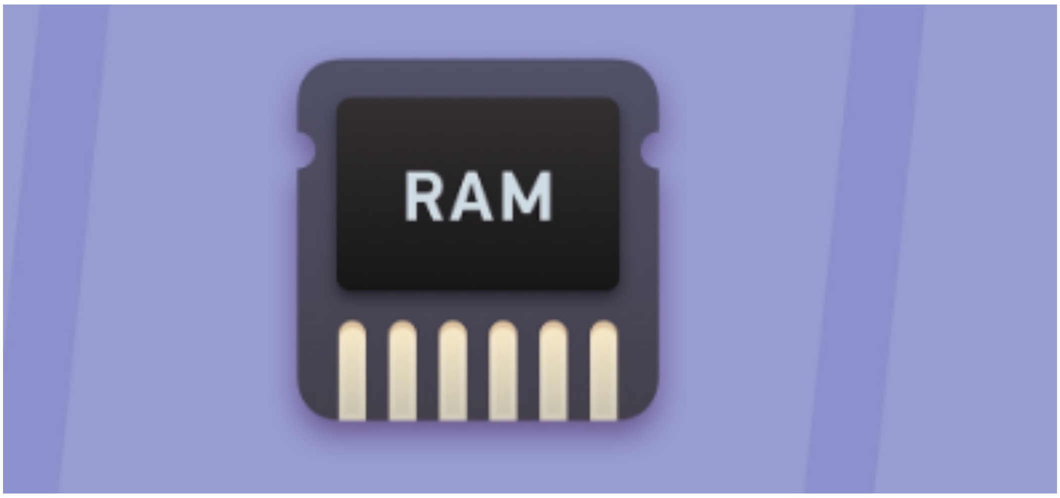
Clarity: Table-driven selection can make code more readable and maintainable, as it can simplify the implementation of complex multiway selection structures.

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.

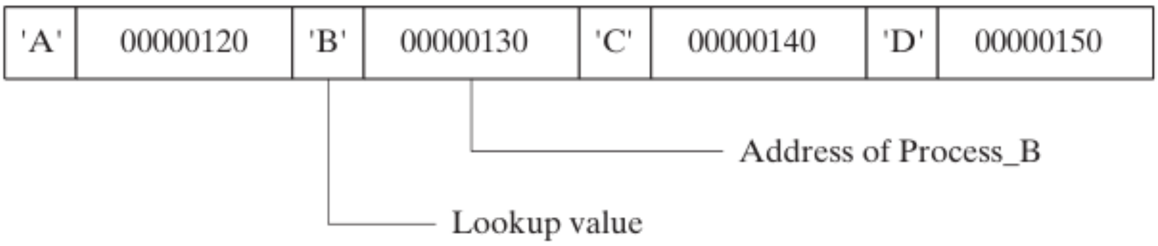


Speed: Table-driven selection can be slower than other methods of implementing multiway selection structures, as it requires a loop to search the table.

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:

```
INCLUDE Irvine32.inc

.data
CaseTable BYTE 'A'          ; Lookup value
          DWORD Process_A   ; Address of procedure
EntrySize = ($ - CaseTable)  ; Calculate the size of each entry in the table
BYTE 'B'
          DWORD Process_B
BYTE 'C'
          DWORD Process_C
BYTE 'D'
          DWORD Process_D
NumberOfEntries = ($ - CaseTable) / EntrySize

prompt BYTE "Press capital A, B, C, or D: ",0
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   ; Ask the user for input
    call WriteString
    call ReadChar           ; Read character into AL
    mov ebx, OFFSET CaseTable ; Point EBX to the table
    mov ecx, NumberOfEntries ; Loop counter

```

```

L1:
    cmp al, [ebx]                ; Match found?
    jne L2                       ; No: continue
    call NEAR PTR [ebx + 1]      ; Yes: call the procedure
    call WriteString             ; Display message
    call Crlf
    jmp L3                       ; Exit the search

L2:
    add ebx, EntrySize           ; Point to the next entry
    loop L1                      ; Repeat until ECX = 0

L3:
    exit

main ENDP

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
```

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

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)

```
prompt BYTE "Press capital A, B, C, or D: ",0
msgA BYTE "Process_A",0
msgB BYTE "Process_B",0
msgC BYTE "Process_C",0
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

```
main PROC
    mov edx, OFFSET prompt
    call WriteString
    call ReadChar
    mov ebx, OFFSET CaseTable
    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.

```
L1:
    cmp al, [ebx]
    jne L2
    call NEAR PTR [ebx + 1]
    call WriteString
    call Crlf
    jmp L3

L2:
    add ebx, EntrySize
    loop L1

L3:
    exit

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

```
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
```

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:

```
; Assuming ebx and ecx are 32-bit variables
; Short-circuit evaluation: if ebx > ecx, set X = 1, else X remains unchanged

cmp ebx, ecx        ; Compare ebx and ecx
jg ebx_greater      ; Jump if ebx > ecx
mov eax, 0          ; If not greater, set eax to 0 (X = 0)
jmp done            ; Jump to done

ebx_greater:
mov eax, 1          ; If ebx > ecx, set eax to 1 (X = 1)

done:
mov X, eax          ; Store the result in X
```

Implementing the pseudocode with short-circuit evaluation:

```
; Assuming edx and ecx are 32-bit variables
; Short-circuit evaluation: if edx <= ecx, set X = 1, else X = 2

cmp edx, ecx        ; Compare edx and ecx
jle edx_less        ; Jump if edx <= ecx
mov eax, 2          ; If not less or equal, set eax to 2 (X = 2)
jmp done            ; Jump to done

edx_less:
mov eax, 1          ; If edx <= ecx, set eax to 1 (X = 1)

done:
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:

```
; Original code (pseudo-code):
; if (eax > ebx) ebx = eax

; Rewritten code using CMOV:
cmp eax, ebx        ; Compare eax and ebx
cmovg ebx, eax      ; If eax > ebx, move eax to ebx (conditional move)

; 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.