Programación en ensamblador i8086

Tema: Algoritmo de búsqueda binaria

Asignatura: Arquitectura de computadoras

Nombre: Lautaro Galante

Profesor: Víctor Teppaz

Fecha: 13 de noviembre de 2023

Como proyecto final de la materia de Arquitectura de computadoras, se nos pidió crear un programa en ensamblador 8086, yo decidí implementar el algoritmo de búsqueda binaria, ya que este me haria pensar un poco sobre como hacerlo y asi poder aprender sobre las distintas instrucciones de la arquitectura.

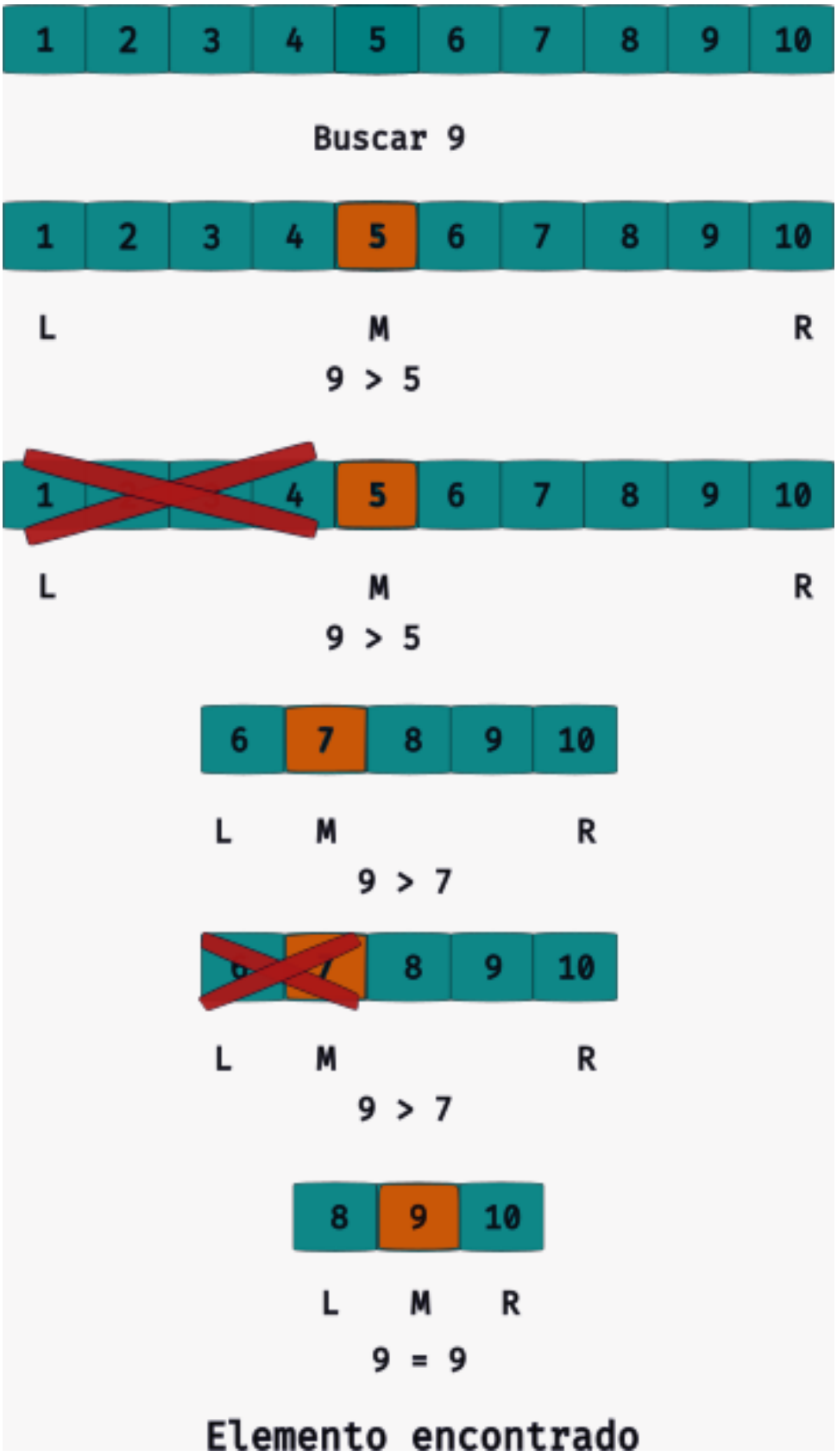
En ciencias de la computación y matemáticas, el algoritmo de búsqueda binaria o binary search en inglés, es un algoritmo de búsqueda que encuentra la posición de un valor en un array ordenado. Compara el valor con el elemento en el medio del array, si no son iguales, la mitad en la cual el valor no puede estar es eliminada y la búsqueda continúa en la mitad restante hasta que el valor se encuentre.

La búsqueda binaria es computada en el peor de los casos en un tiempo logarítmico, realizando $O(\log n)$ comparaciones, donde **n** es el número de elementos del arreglo y **log** es el logaritmo. La búsqueda binaria requiere solamente $O(1)$ en espacio, es decir, que el espacio requerido por el algoritmo es el mismo para cualquier cantidad de elementos en el array.

Para que se entienda mejor dejo expresado el algoritmo en pseudocódigo:

```
Function busqueda_binaria(Array, n, T):
    Izquierda := 0
    Derecha := n − 1
    while Izquierda ≤ Derecha do
        medio := (Izquierda + Derecha)/2
        if Array[medio] < T then
            Izquierda := medio + 1
        else
            if Array[medio] > T then
                Derecha := medio − 1
            else
                return medio
            end
        end
    end
    return no se encontro
```

Aqui hay una representación gráfica de los pasos que realiza el algoritmo



A continuación explicare que hace cada parte del código en ensamblador

Bloque 1

```
1       ORG 100h
2
3       .DATA
4           numeros db 1, 2, 3, 4, 5, 6, 7, 8, 9
5           numero db ?
6           mensaje db "Ingrese el numero a buscar: ", "$"
7           resultado db "El indice del valor ingresado es: ", "$"
8           left db 0
9           right db 8
10          middle db ?
11          value db ?
12
```

Al inicio del código en la linea **1** declaro **ORG 100h** que es una directiva del compilador, luego en la linea **3** defino un sector de datos **.DATA** en el cuál tengo un array y distintas variables, el primero es un array **numeros** que es de tipo **db** que tiene una capacidad de **8 bits** es decir **1 byte**, este contiene números del **1** al **9**, en la linea **5** una variable **numero** del mismo tipo la cuál la inicialize con el simbolo **?** que significa que aún no se le a asignado un valor, en la linea **6** una variable string **mensaje** y en la **7** **resultado** para mostrar los mensajes para que se ingrese el número, y para indicar el valor de salida. Luego hay cuatro variables, **left, rigth, middle y value**, **left** guarda el primer indice del array que es **0** y **right** guarda el inidice **8** que es el ultimo elemento del array en este caso es el **9**, en la variable **middle** y **value** aún no le asigne ningún valor, luego mas adelante en el codigo se explicara para que son cada variable.

Bloque 2

```
1       .CODE
2         MAIN:
3           mov ax, numeros
4           mov dx, offset mensaje
5
6           mov ah, 9h
7           int 21h
8
9           mov ah, 01h
10          int 21h
11          sub al, '0'
12          mov numero, al
13
14          call BINARYSEARCH
15          call PRINT
16
```

En el bloque **2** de codigo dentro del sector **.CODE** se encuentra el procedimiento principal **.MAIN** que tiene las primeras instrucciones y las llamadas a otros procedimientos, en la linea **3** se copian los valores del array **numeros** a el registro entero de 16 bits **ax** usando la instrucción **mov**, en la linea **4** se copia la dirección de memoria de la variable **mensaje** utilizando el operador **offset** en el registro **dx**, en la linea **6** se copia el valor hexadecimal **9h** al registro **ah** que es la parte alta de **ax** esta es una función para imprimir una cadena por consola, en la linea **7** se ejecuta la interrupción **int 21h** el **bios** busca el código de función que coincide con la función cargada en el registro **ah** y procede a imprimir la cadena **Ingrese el numero a buscar:** que se encuentra guardada en la variable **mensaje**.

En la linea **9** se copia el valor de la función **01h** en el registro **ah** esta función es para leer un carácter desde el teclado, en la linea **10** se ejecuta la interrupción, en la linea **11** se convierte el valor numérico **ascci** a decimal y se almacena en el registro **al**, luego en la linea **12** se copia el valor del registro **al** en la variable **numero**, en la linea **14** se utiliza el operador **call**

para llamar al procedimiento **BINARYSEARCH** que es donde ocurre toda la lógica del algoritmo, y en la linea **15** se llama al procedimiento **PRINT** que lo utilizo para imprimir los valores.

Bloque 3

```
1       BINARYSEARCH PROC
2           mov ax, 0
3           mov bl, left
4           mov bh, right
5
6           SEARCHLOOP:
7               cmp bl, bh
8               jg SEARCHEND
9
```

En este bloque **3** de código definimos el procedimiento **BINARYSEARCH**, donde dentro en la linea **2** limpiamos el registro de **ax** copiando el valor **0**, en la linea **3** copiamos el valor almacenado en la variable **left** a **bl** la parte baja de **bx**, en la linea **4** copiamos el valor del la variable **right** a **bh** su parte alta.

Luego en la linea **6** definimos un ciclo **SEARCHLOOP**, en el utilizamos la instrucción **cmp** su función es equivalente a la de una resta, ya que resta el segundo operando al el primer operando, pero no guarda el resultado de la resta, sino que cambia las banderas de estado en el registro de banderas, **CF:** se establece en 1 si el primer operando es menor al segundo, En este caso como el segundo operando es mayor al primero, la bandera **CF** se setea en **1**.

En la linea **8** la instrucción **jg** corre la ejecución del programa a donde se encuentre la definición de la etiqueta asignada a su derecha, que en este caso es **SEARCHEND**, esto sucede si el primer operando es mayor al segundo, lo que llevaria a terminar el ciclo ya que no tiene mucho sentido seguir con la búsqueda por que cuando se llega a esta instancia significa que se encontro el indice del valor ingresado.

Bloque 4

```
1            mov al, bh
2            add al, bl
3            mov cl, 2
4            div cl
5
6            mov cx, 0
7            mov cl, numero
8
```

Este bloque **4** de código es una continuación de las instrucciónes que se encuentran dentro del ciclo **SEARCHLOOP**, en la linea **1** copiamos el valor de **bh** en **al**, en la linea **2** con la instrucción **add** sumamos el valor de **bl** a **al**. En la linea **3** copiamos el valor **2** al registro bajo **cl**, y en la linea **4** usamos la intrucción **div** para dividir en dos el valor que se encuentra en **al**, por lo que ahora tendremos dos valores en el registro **ax**, en **ah** se ecuentra el residuo y en **al** se encuentra el cociente.

En la linea **6** limpiamos el valor del registro **cx**, y en la linea **7** copiamos el valor del la variable **numero**, que es el valor que ingresamos teclado.

Bloque 5

```
1            mov ah, 0
2            mov si, ax
3
4            mov middle, al
5            mov al, numeros[si]
6            mov value, al
7
8            cmp value, cl
9            jne second
10           jmp SEARCHEND
11
```

En el bloque **5** de código, en la linea número **1** se sobreescribe con cero el valor del registro alto **ah** para que en el registro entero **ax** solo quede en su parte baja **al** el resultado de haber dividido la sumatoria de los valores **left** y **right** entre **2**, entonces en la linea **2** copiamos el valor de **ax** en **si** (**source index**) que es el índice de la mitad del array **numeros**.

Igualmente necesitamos guardar el valor del índice medio en la variable **middle**, eso ocurre en la linea **4** donde copiamos el valor de el registro **al** ya que necesitamos reescribir ese registro para que en la linea **5**, utilicemos el **si** para iterar dentro del array **numeros** de la siguiente manera **numeros**[**si**], entonces con el índice medio dentro del array accedemos al número que se encuentra en esa posición media y lo copiamos en el registro **al**, luego en la linea **6** copiamos ese número en la variable **value**.

En la linea **8** comparamos ese número medio con el número que se ingreso por teclado que ya lo tenemos almacenado en el registro **cl** que explique en el bloque **4**, por lo que si los números no son iguales la instrucción **jne** salta al procedimiento **second**, sino la instrucción **jmp SEARCHEND** termina el bucle y se retorna ese número ya que son iguales.

Bloque 6

```
1           second:
2               cmp value, cl
3               jg third
4
5               mov dx, 0
6               mov dl, middle
7               inc dl
8               mov bl, dl
9
10              jmp SEARCHLOOP
11
```

En el bloque **6** en la linea **1** se encuentra la definición del procedimiento **second**, dentro del mismo en la linea **2** comparamos el valor medio del array, es decir el número que se encuentra en la mitad con el número ingresado por teclado, en la linea **3** si el primer operando es mayor al segundo se salta al procedimiento **third**, sino se procede con las operaciones que comienzan en la linea **5**, se copia el valor **0** en el registro **dx** para limpiarlo, luego en la linea **6** se copia el indice del valor medio que esta guardado en la variable **middle** en **dl**, luego en la linea **7** se usa la instrucción **inc** para incrementar en uno el registro **dl**.

En la linea **8** se copia el valor incrementado de **dl** en el registro **bl** ya que este corresponde a la parte izquierda del array **numeros**, para recortar la parte de array en donde no se encuentra valor que se ingreso por teclado. Luego en la linea **10** se salta al principio del loop utilizando la instrucción **jmp SEARCHLOOP** para que este se vuelva a iterar.

Bloque 7

```
1              third:
2                  mov dx, 0
3                  mov dl, middle
4                  dec dl
5                  mov bh, dl
6
7                  jmp SEARCHLOOP
8
9          SEARCHEND:
10         RET
11
12     BINARYSEARCH ENDP
13
14 END MAIN
15
```

En el bloque **7** de código en la linea **1** se define el procedimiento **third**, donde en la linea **2** limpiamos el registro **dx**, en la linea **3** copiamos el valor de la variable **middle** en **dl**, y luego en la linea **4** utilizamos la instrucción **dec** para decrementar en **1** el valor que se encuentra en **dl**, ya que en este caso no necesitamos hacer ninguna comparación, por que el programa llego hasta este procedimiento por que la comparación del procedimiento anterior **second** dio que el primer operando era mayor al segundo, entonces en este procedimiento solo se realizan las operaciones necesarias, para que en la linea **5** se copie el nuevo valor de **dl** en **bh** que es la parte derecha del array **numeros**, luego en la linea **7** con la instrucción **jmp SEARCHLOOP** volvemos a iterar hasta que se encuentra el valor que buscamos, cuando la ejecución llega nuevamente a la comparación que ocurre en el bloque de código número **3** y el valor del registro **bl** es mayor que **bh** el bucle **SEARCHLOOP** termina y se retorna en la linea **10** con la instrucción **RET**

Bloque 8

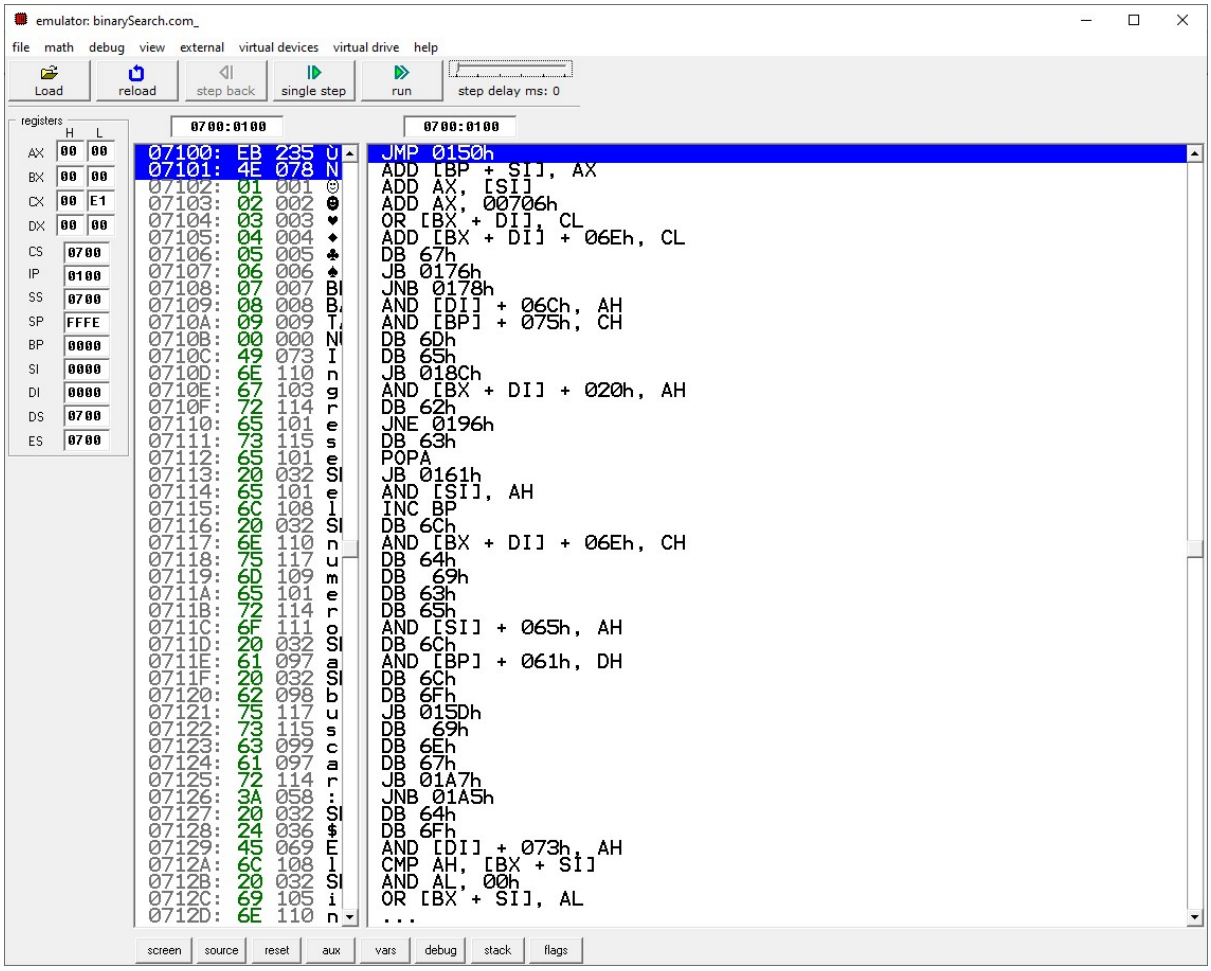
```
1        PRINT:
2            mov dl, 13
3            mov ah, 2
4            int 21h
5
6            mov dl, 10
7            mov ah, 2
8            int 21h
9
10           mov dx, offset resultado
11           mov ah, 9h
12           int 21h
13
14           mov dl, middle
```

```
15              add dl, '0'
16              mov ah, 02h
17              int 21h
18              mov ax, 4C00h
19              int 21h
20
```

Al retornar, el procedimiento que sigue en el orden de llamadas es el procedimiento **PRINT** que sirve para imprimir el índice del número que ingresamos por teclado, que es la posición donde se encuentra ese número en el array. En la linea **2** del bloque **8** de código se copia el valor **13** en el registro **dl**, en la linea **3** se copia el valor **2** en **ah**, en la linea **4** se usa la instrucción **int 21h**, que en conjunto estas tres instrucciónes lo que hacen es aplicar un retorno de carro, para que en la linea **6, 7 y 8** se aplique un salto de linea en la consola, para que el resultado no se imprima pegado a la impresión anterior.

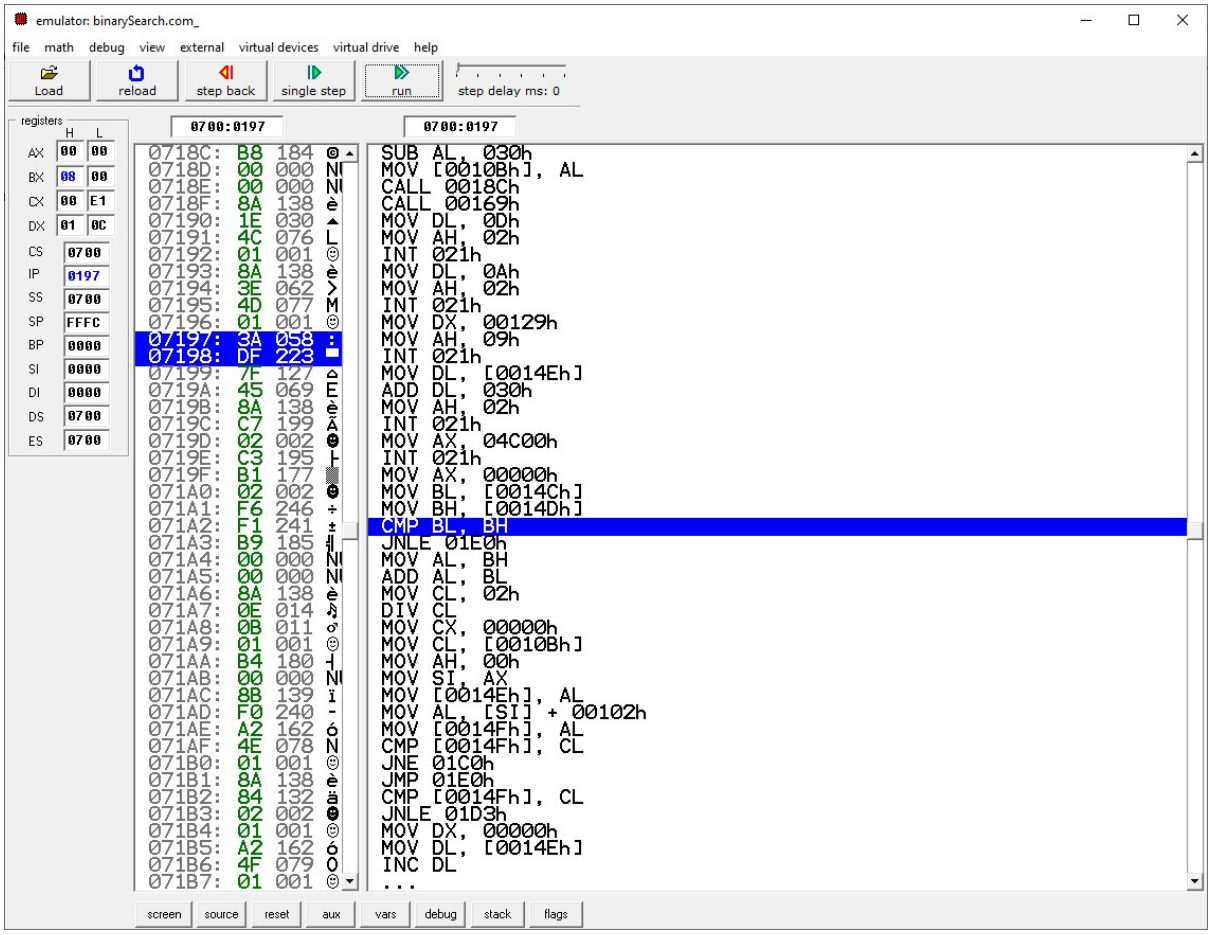
En la linea **10** se copia la dirección de memoria de la variable **resultado** usando la instrucción **offset** en el registro **dx**, en la linea **11** se copia el valor de la función **9h** en **ah**, luego se usa la interrupción **int 21h** para imprimir la cadena que se guarda en la variable resultado que es **El indice del valor ingresado es:** , en la linea **14** se copia el indice del número buscado que esta en **middle** al registro **dl**, en la linea **15** se convierte ese valor **ascci** a decimal, para que en la linea **16 y 17** se imprima el indice del número buscado por consola, en la linea **18 y 19** utilizo una instrucción para terminar la ejecución del programa.

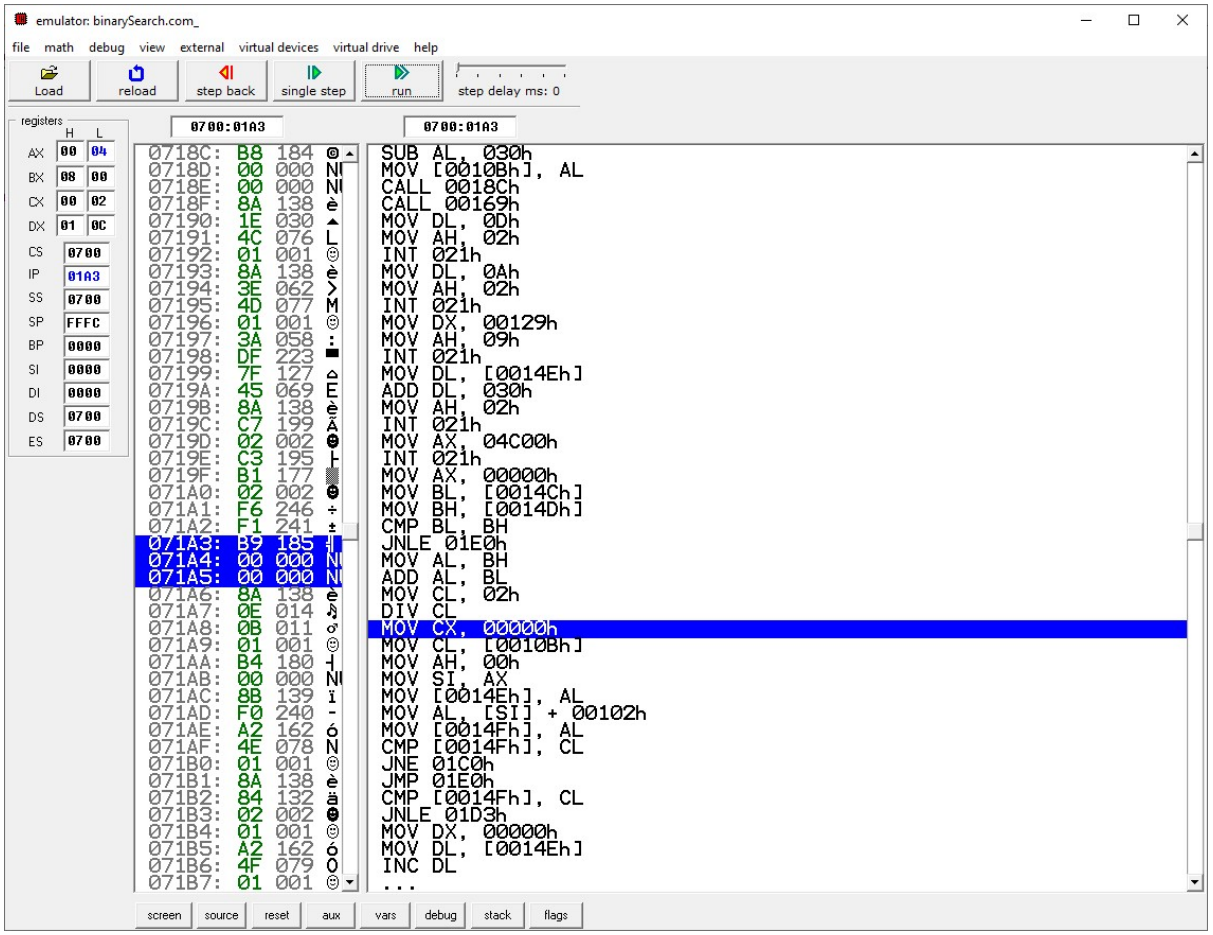
A continuación mostrare unas capturas de pantalla de la ejecución del programa en el emulador emu8086.

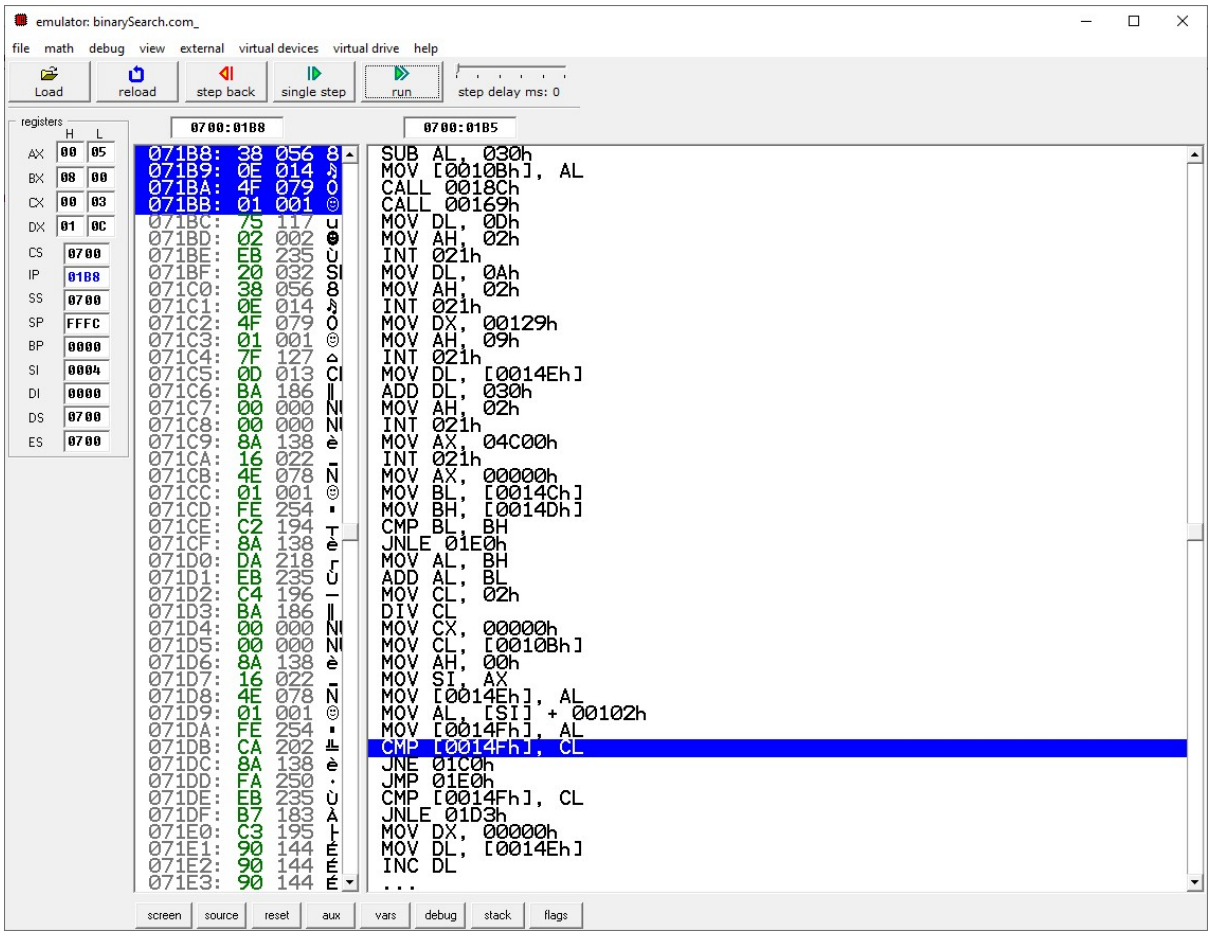

emulator: binarySearch.com_
file math debug view external virtual devices virtual drive help
Load
reload
step back
single step
run
step delay ms: 0
registers
H L
AX 00 00
BX 00 00
CX 00 E1
DX 00 00
CS 0700
IP 0100
SS 0700
SP FFFE
BP 0000
SI 0000
DI 0000
DS 0700
ES 0700
0700:0100
0700:0100
07100: EB 235 ë JMP 0150h
07101: 4E 078 N ADD [BP + SI], AX
07102: 01 001 ☺ ADD AX, [SI]
07103: 02 002 ☻ ADD AX, 00706h
07104: 03 003 ♥ OR [BX + DI], CL
07105: 04 004 ♦ ADD [BX + DI] + 06Eh, CL
07106: 05 005 ♣ DB 67h
07107: 06 006 ♠ JB 0176h
07108: 07 007 BI JNB 0178h
07109: 08 008 B AND [DI] + 06Ch, AH
0710A: 09 009 T AND [BP] + 075h, CH
0710B: 00 000 NI DB 6Dh
0710C: 49 073 I DB 65h
0710D: 6E 110 n JB 018Ch
0710E: 67 103 g AND [BX + DI] + 020h, AH
0710F: 72 114 r DB 62h
07110: 65 101 e JNE 0196h
07111: 73 115 s DB 63h
07112: 65 101 e POPA
07113: 20 032 SI JB 0161h
07114: 65 101 e AND [SI], AH
07115: 6C 108 l INC BP
07116: 20 032 SI DB 6Ch
07117: 6E 110 n AND [BX + DI] + 06Eh, CH
07118: 75 117 u DB 64h
07119: 6D 109 m DB 69h
0711A: 65 101 e DB 63h
0711B: 72 114 r DB 65h
0711C: 6F 111 o AND [SI] + 065h, AH
0711D: 20 032 SI DB 6Ch
0711E: 61 097 a AND [BP] + 061h, DH
0711F: 20 032 SI DB 6Ch
07120: 62 098 b DB 6Fh
07121: 75 117 u JB 015Dh
07122: 73 115 s DB 69h
07123: 63 099 c DB 6Eh
07124: 61 097 a DB 67h
07125: 72 114 r JB 01A7h
07126: 3A 058 : JNB 01A5h
07127: 20 032 SI DB 64h
07128: 24 036 $ DB 6Fh
07129: 45 069 E AND [DI] + 073h, AH
0712A: 6C 108 l CMP AH, [BX + SI]
0712B: 20 032 SI AND AL, 00h
0712C: 69 105 i OR [BX + SI], AL
0712D: 6E 110 n ...
screen source reset aux vars debug stack flags

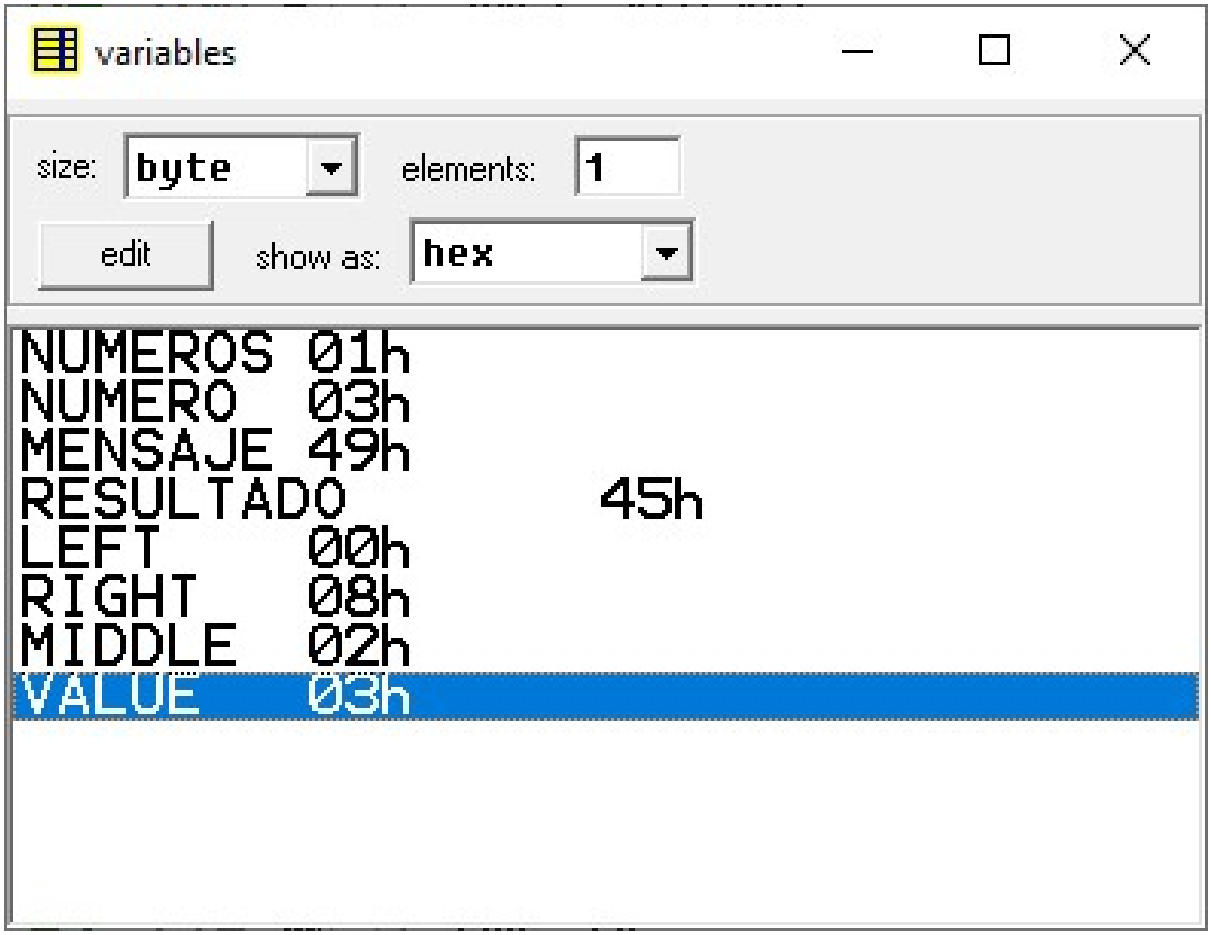

emulator screen (80x25 chars)
Ingrese el numero a buscar: _
clear screen
change font
0/16


emulator screen (80x25 chars)
Ingrese el numero a buscar: 3
clear screen
change font


emulator: binarySearch.com_
file math debug view external virtual devices virtual drive help
Load
reload
step back
single step
run
step delay ms: 0
registers
0700:0197
0700:0197
0718C: B8 184 SUB AL, 030h
0718D: 00 000 MOV [0010Bh], AL
0718E: 00 000 CALL 0018Ch
0718F: 8A 138 CALL 00169h
07190: 1E 030 MOV DL, 0Dh
07191: 4C 076 MOV AH, 02h
07192: 01 001 INT 021h
07193: 8A 138 MOV DL, 0Ah
07194: 3E 062 MOV AH, 02h
07195: 4D 077 INT 021h
07196: 01 001 MOV DX, 00129h
07197: 3A 058 MOV AH, 09h
07198: DF 223 INT 021h
07199: 7F 127 MOV DL, [0014Eh]
0719A: 45 069 ADD DL, 030h
0719B: 8A 138 MOV AH, 02h
0719C: C7 199 INT 021h
0719D: 02 002 MOV AX, 04C00h
0719E: C3 195 INT 021h
0719F: B1 177 MOV AX, 00000h
071A0: 02 002 MOV BL, [0014Ch]
071A1: F6 246 MOV BH, [0014Dh]
071A2: F1 241 CMP BL, BH
071A3: B9 185 JNLE 01E0h
071A4: 00 000 MOV AL, BH
071A5: 00 000 ADD AL, BL
071A6: 8A 138 MOV CL, 02h
071A7: 0E 014 DIV CL
071A8: 0B 011 MOV CX, 00000h
071A9: 01 001 MOV CL, [0010Bh]
071AA: B4 180 MOV AH, 00h
071AB: 00 000 MOV SI, AX
071AC: 8B 139 MOV [0014Eh], AL
071AD: F0 240 MOV AL, [SI] + 00102h
071AE: A2 162 MOV [0014Fh], AL
071AF: 4E 078 CMP [0014Fh], CL
071B0: 01 001 JNE 01C0h
071B1: 8A 138 JMP 01E0h
071B2: 84 132 CMP [0014Fh], CL
071B3: 02 002 JNLE 01D3h
071B4: 01 001 MOV DX, 00000h
071B5: A2 162 MOV DL, [0014Eh]
071B6: 4F 079 INC DL
071B7: 01 001 ...
screen source reset aux vars debug stack flags


emulator: binarySearch.com_
file math debug view external virtual devices virtual drive help
Load reload step back single step run step delay ms: 0
registers
AX 00 04
BX 08 00
CX 00 02
DX 01 0C
CS 0700
IP 01A3
SS 0700
SP FFFC
BP 0000
SI 0000
DI 0000
DS 0700
ES 0700
0700:01A3
0700:01A3
0718C: B8 184
0718D: 00 000
0718E: 00 000
0718F: 8A 138
07190: 1E 030
07191: 4C 076
07192: 01 001
07193: 8A 138
07194: 3E 062
07195: 4D 077
07196: 01 001
07197: 3A 058
07198: DF 223
07199: 7F 127
0719A: 45 069
0719B: 8A 138
0719C: C7 199
0719D: 02 002
0719E: C3 195
0719F: B1 177
071A0: 02 002
071A1: F6 246
071A2: F1 241
071A3: B9 185
071A4: 00 000
071A5: 00 000
071A6: 8A 138
071A7: 0E 014
071A8: 0B 011
071A9: 01 001
071AA: B4 180
071AB: 00 000
071AC: 8B 139
071AD: F0 240
071AE: A2 162
071AF: 4E 078
071B0: 01 001
071B1: 8A 138
071B2: 84 132
071B3: 02 002
071B4: 01 001
071B5: A2 162
071B6: 4F 079
071B7: 01 001
SUB AL, 030h
MOV [0010Bh], AL
CALL 0018Ch
CALL 00169h
MOV DL, 0Dh
MOV AH, 02h
INT 021h
MOV DL, 0Ah
MOV AH, 02h
INT 021h
MOV DX, 00129h
MOV AH, 09h
INT 021h
MOV DL, [0014Eh]
ADD DL, 030h
MOV AH, 02h
INT 021h
MOV AX, 04C00h
INT 021h
MOV AX, 00000h
MOV BL, [0014Ch]
MOV BH, [0014Dh]
CMP BL, BH
JNLE 01E0h
MOV AL, BH
ADD AL, BL
MOV CL, 02h
DIV CL
MOV CX, 00000h
MOV CL, [0010Bh]
MOV AH, 00h
MOV SI, AX
MOV [0014Eh], AL
MOV AL, [SI] + 00102h
MOV [0014Fh], AL
CMP [0014Fh], CL
JNE 01C0h
JMP 01E0h
CMP [0014Fh], CL
JNLE 01D3h
MOV DX, 00000h
MOV DL, [0014Eh]
INC DL
...
screen source reset aux vars debug stack flags


emulator: binarySearch.com_
file math debug view external virtual devices virtual drive help
Load reload step back single step run step delay ms: 0
registers
AX 00 05
BX 08 00
CX 00 03
DX 01 0C
CS 0700
IP 01B8
SS 0700
SP FFFC
BP 0000
SI 0004
DI 0000
DS 0700
ES 0700
0700:01B8
0700:01B5
071B8: 38 056
071B9: 0E 014
071BA: 4F 079
071BB: 01 001
071BC: 75 117
071BD: 02 002
071BE: EB 235
071BF: 20 032
071C0: 38 056
071C1: 0E 014
071C2: 4F 079
071C3: 01 001
071C4: 7F 127
071C5: 0D 013
071C6: BA 186
071C7: 00 000
071C8: 00 000
071C9: 8A 138
071CA: 16 022
071CB: 4E 078
071CC: 01 001
071CD: FE 254
071CE: C2 194
071CF: 8A 138
071D0: DA 218
071D1: EB 235
071D2: C4 196
071D3: BA 186
071D4: 00 000
071D5: 00 000
071D6: 8A 138
071D7: 16 022
071D8: 4E 078
071D9: 01 001
071DA: FE 254
071DB: CA 202
071DC: 8A 138
071DD: FA 250
071DE: EB 235
071DF: B7 183
071E0: C3 195
071E1: 90 144
071E2: 90 144
071E3: 90 144
SUB AL, 030h
MOV [0010Bh], AL
CALL 0018Ch
CALL 00169h
MOV DL, 0Dh
MOV AH, 02h
INT 021h
MOV DL, 0Ah
MOV AH, 02h
INT 021h
MOV DX, 00129h
MOV AH, 09h
INT 021h
MOV DL, [0014Eh]
ADD DL, 030h
MOV AH, 02h
INT 021h
MOV AX, 04C00h
INT 021h
MOV AX, 00000h
MOV BL, [0014Ch]
MOV BH, [0014Dh]
CMP BL, BH
JNLE 01E0h
MOV AL, BH
ADD AL, BL
MOV CL, 02h
DIV CL
MOV CX, 00000h
MOV CL, [0010Bh]
MOV AH, 00h
MOV SI, AX
MOV [0014Eh], AL
MOV AL, [SI] + 00102h
MOV [0014Fh], AL
CMP [0014Fh], CL
JNE 01C0h
JMP 01E0h
CMP [0014Fh], CL
JNLE 01D3h
MOV DX, 00000h
MOV DL, [0014Eh]
INC DL
...
screen source reset aux vars debug stack flags


variables
size:
byte
elements:
1
edit
show as:
hex
NUMEROS 01h
NUMERO 03h
MENSAJE 49h
RESULTADO 45h
LEFT 00h
RIGHT 08h
MIDDLE 02h
VALUE 03h


emulator screen (80x25 chars)
Ingrese el numero a buscar: 3
El indice del valor ingresado es: 2
clear screen
change font