PRÁCTICA 7

Carlos Raso Alonso

Introducción a los ordenadores

Índice

Objetivos

El objetivo de esta práctica es conseguir hacer un programa más complejo con el simulador 8085 aplicando todo lo que hemos hecho hasta ahora. En concreto el programa debe hacer distintas operaciones aritmetico-lógicas dependiendo de qué inputs reciba y posteriormente debe mostrar los resultados por pantalla.

Ejercicios

Tarea 1

Diseña una subrutina que a partir de dos números en base 10 introducidos por teclado haga la suma y presente el resultado por pantalla de texto. Usa direccionamientos directos e indirectos e indica dónde usas cada uno. ¿ Cómo gestionas los problemas del signo y del overflow? Escribe el código.

Está respondido en los comentarios del código de la tarea 3. Como esta tarea implementa un programa que efectúa sumas restas and y or, en particular también realiza sumas de dos números. Hay marcados en rojo en el código dos ejemplos de direccionamiento directo e indirecto:
El ejemplo de direccionamiento indirecto es mov A, M. En este caso se especifica en la instrucción el par de registros HL en el que se encuentra la dirección de memoria en la que está el byte que queremos introducir en A por lo que estamos ante un direccionamiento indirecto por parte del par HL, ya que el operando es el contenido de la posición de memoria en HL.
EL ejemplo de direccionamiento directo es mov D, C en el que simplemente se transmite el contenido de un registro, el C, a otro,el D. el contenido que se transmite está direccionado de manera directa puesto que solo se ha tenido que especificar el registro que contiene el operando en la instrucción.

Tarea 2

Diseña una subrutina similar a la anterior pero que reste ambos números en vez de sumarlos usando también los dos mismos tipos de direccionamiento e indicando dónde están en el código. ¿Cómo gestionas los problemas de signo y de carry? Escribe el código

De nuevo, esta tarea está respondida como comentarios en el código de la tarea 3, que también hace restas de dos números. Los ejemplos de direccionamiento directo e indirecto están explicados en el ejercicio anterior.

Tarea 3

A partir de los códigos de las tareas 1 y 2 haz un programa capaz de hacer sumas, restas, ands y ors. Escribe el código

Para usar el programa tienen que estar habilitadas las interrupciones por teclado y el teclado en el puerto 04h. Para ejecutar una operación de tipo and or, suma o resta de números de tres dígitos se debe introducir la siguiente secuencia:
signo1-centenas1-decenas1-unidades1-operación-signo2-centenas2-decenas2-unidades2-igual
Si se quiere poner un número de menos de tres dígitos se deben rellenar los restantes con 0. El programa falla si cualquiera de los números, los introducidos o el resultado, son menores de -127 o mayores de 127 o si se introducen dos caracteres segidos muy rápido. También falla si se inserta

una secuencia fallida. En cualquiera de estos casos se mostrará un mensaje de error. Por ejemplo si se quiere restar 54 - 18 se deberá escribir la secuencia: “+054-+018=” y el programa mostrará por pantalla el resultado con su signo.

```
.define
      numbers_count 10
.org 00h
; Inicializamos el par BC que contendrá la siguiente dirección de memoria en la
; que se debe escribir para que salga por la pantalla de texto y también el
; registro D, el cual almacena el número de dígitos introducido por el usuario.
; Esto dictará qué tipo de carácter se podrá introducir en cada momento.
      mvi B, E0h
      mvi C, 00h
      mvi D, 00h
; Saltamos al bucle principal loop
      jmp loop


.org 24h
; En cada interrupción del teclado únicamente se llama a la rutina tecla_in
      call tecla_in
      ret



.data 150h
; En data se guardan los códigos ascii de los 10 números decimales y también se
; guardan tres números: n1, n2 y sol. Cada uno de estos números se compone de
; tres byts seguidos que son los tres dígitos decimales del número,
; correspondientes por ejemplo a los tres bits n1_digits, un bit que representa
; el signo del número (1 si es negativo y 0 si es positivo), por ejemplo n1_sign
; Por último, en un byte se representa el número en complemento a 2 (de lo que
; se deduce que el rango que se puede representar es de -127 a 127). Por ejemplo
; n1. Además se guarda el tipo de operación que ha introducido el usuario en
; forma de código ascii en la dirección op.
      numbers: db 30h,31h,32h,33h,34h,35h,36h,37h,38h,39h
      n1_digits: db 00h, 00h, 00h
      n1_sign: db 00h
      n1: db 00h
      n2_digits: db 00h, 00h, 02h
      n2_sign: db 00h
      n2: db 00h
      sol_digits: db 00h, 00h, 00h
      sol_sign: db 00h
      sol: db 00h
      op : db 00h


.org 200h
loop:
; En el bucle loop el programa se queda a la espera de una nueva interrupción de
; teclado.
      jmp loop


tecla_in:
; La rutina tecla_in se activa cada vez que se pulsa una tecla del teclado.
; Su función es distinguir si la tecla continúa la secuencia de manera correcta.
; En caso de que así sea, tratar la información de manera pertinente y en caso
; de que no, mostrar un mensaje de error.
```

```
; Primero se comprueba si la tecla inrtoducida debe ser un signo, operación o
; igual. Sabemos esto según el número de dígitos introducidos hasta el momento
      mov A, D
      cpi 00h
      jz insert_sign1
      cpi 05h
      jz insert_sign2
      cpi 04h
      jz insert_op
      cpi 09h
      jz insert_equals

insert_num:
; Si la tecla a introducir no debe ser un signo, operación o la tecla igual,
; debe ser un número por lo que se prosigue comprobando si es un número con la
; rutina check number. En caso de que no sea un número se muestra un mensaje de
; error, puesto que se habrá introducido una tecla incorrecta.
      in 04h
      call check_number
      cpi 00h
      jz error
; Una vez sabemos que la tecla es un número, vemos si se está introduciendo un
; dígito del primer o del segundo número. Para ello se guarda provisionalmente
; el contenido del acumulador (el código ascii del número introducido) en E.

      mov E, A
      mov A, D
      cpi 01h
      jz primer_num
      cpi 02h
      jz primer_num
      cpi 03h
      jz primer_num
; Si el dígito es del segundo número se resta 6 al número de dígitos
; introducidos para hallar el dígito del segundo número que se está
; introduciendo (el dígito de las centenas es el dígito 0, el de las
; decenas el 1 y el de las unidades el 2)
      sbi 06h
      lxi H, n2_digits
; Cuando sabemos qué dígito es respecto al primer número hallamos la posición de
; memoria en la que lo debemos guardar sumando 1 a la posición n2_digits
; por cada número de dígito ya que en la primera posición se guardan las
; centenas, en la segunda las decenas y en la última las unidades. Esto se hace
; con el bucle loop_digito2
loop_digito2:
      cpi 00h
      jz end_loop_digito2
      inx H
      dcr A
      jmp loop_digito2
end_loop_digito2:
; Una vez tenemos la dirección de memoria en la que debemos almacenar el dígito
; introducido recuperamos el valor ascii del número en el acumulador y le
; restamos 30h para obtener la expresión binaria del dígito, luego lo
; mostramos por pantalla y lo almacenamos en la posición de memoria
; correspondiente.
      mov A, E
      stax B
      inx B
      sui 30h
```

```
      mov M, A
      jmp end
; Si el dígito introducido es del primer número se procede de manera similar.
; solo cambia el número que restamos para obtener el dígito dentro del primer
; número, que en este caso es 1.
primer_num:
      dcr A
      lxi H, n1_digits
loop_digito1:
      cpi 00h
      jz end_loop_digito1
      inx H
      dcr A
      jmp loop_digito1
end_loop_digito1:
      mov A, E
      stax B
      inx B
      sui 30h
      mov M, A
      jmp end

insert_op:
; Si la tecla introducida debe ser una operación, primero se comprueba si
; realmente es uno de los caracteres ascii asociados a una operación y, en caso
; de que lo sea, se almacena este código en la posición de memoria op y se
; muestra por pantalla el tipo de operación.
      in 04h
      cpi 2bh
      jz not_error
      cpi 2dh
      jz not_error
      cpi 7ch
      jz not_error
      cpi 26h
      jz not_error
      jmp error
not_error:
      lxi H, op
      mov M, A
      stax B
      inx B
      jmp end

insert_sign1:
; Si en la secuencia no se han introducido caracteres, se debe empezar con un
; signo. En insert_sign1 se compreba que la tecla introducida sea un signo y,
; si lo es, se guarda 1 ó 0 en la dirección n1_sign dependiendo de si se ha
; introducido - ó +, respectivamente. También se muestra el signo por pantalla.
      in 04h
      cpi 2Dh
      jz neg1
      cpi 2Bh
      jz pos1
      jmp error
pos1:
      lxi H, n1_sign
      mvi M, 00h
      stax B
      inx B
      jmp end
```

```
neg1:
      lxi H, n1_sign
      mvi M, 01h
      stax B
      inx B
      jmp end

insert_sign2:
; Si se han introducido 5 caracteres en la secuencia significa que se debe
; introducir posteriormente el signo del segudo número. Se procede de manera
; similar a insert_sign1 pero,en este caso, se guarda en la posición de memoria
; n2_sign.
      in 04h
      cpi 2dh
      jz neg2
      cpi 2Bh
      jz pos2
      jmp error
pos2:
      lxi H, n2_sign
      mvi M, 00h
      stax B
      inx B
      jmp end
neg2:
      lxi H, n2_sign
      mvi M, 01h
      stax B
      inx B
      jmp end

insert_equals:
; Una vez se han introducido los dos números y la operación, es decir, 9
; caracteres, se debe introducir un igual para que se ejecute la operación así
; que se comprueba si esta es la tecla introducida. Si lo es, se escribe el
; igual por pantalla
      in 04h
      cpi 3dh
      jnz error
      stax B
      inx B
; tras escribir el igual, se convierten los tres dígitos del primer y segundo
; número junto con el signo a un número expresado en complemento a2 y este se
; almacena en la posición correspondiente de la memoria (n1 o n2). Para
; convertirlos se usa la rutina convert_bin.
      lxi H, n1_digits
      call convert_bin
      lxi H, n2_digits
      call convert_bin
; Cuando ya se tiene la expresión en ca2 de ambos operandos primero se comprueba
; la operación a realizar a partir de la posición de memoria op. También se
; carga en el registro E el primer número (expresado en ca2) para luego operar
      lxi H, n1
      mov E, M

      lxi H, op
      mov A, M
      cpi 2bh
      jz op_sum
      cpi 2dh
      jz op_sub
```

```
      cpi 26h
      jz op_and
      cpi 7ch
      jz op_or
op_sum:
; En el caso de la operación suma, primero se carga n2 en A. Luego se suma a
; este número n2, el cual ya estaba almacenado en E. A la vez se guarda el signo
; de n1, de n2 y de la suma en ca2 entre los dos en los registros C, D y B
; respectivamente. Para conocer el signo se utiliza la rutina is_neg. El
; resultado de la suma se almacena en la pila para recuperarlo más tarde.
      lxi H, n2
      mov A, M          ; Ejemplo de direccionamiento indirecto
      push B
      call is_neg
      mov D, C          ; Ejemplo de direccionamiento directo
      add E
      push PSW
      call is_neg
      mov B, C
      mov A, E
      call is_neg
      mov A, C
; Cuando los signos de n1, n2 y n1 + n2 están almacenados en los registros
; arriba especificados, se procede a comprobar si se ha prducido overflow.
; Para ello se ve primero si n1 y n2 tienen distinto signo, en cuyo caso nunca
; habrá overflow por lo que podemos ir a la dirección is_correct1
      xra D
      cpi 01h
      jz is_correct1
; En caso de que n1 y n2 tengan el mismo signo puede haber overflow. Para saber
; si lo hay solo tenemos que comprobar si el signo de uno de ellos corresponde
; con el signo de la suma de ambos, en cuyo caso la suma será correcta, es decir
; no habrá overflow. Sin embargo, si n1 y n2 tienen el mismo signo y n1 + n2
; tiene signo distinto en ca2 sabemos que se ha producido overflow (puesto que
; esto es imposible) por lo tanto procedemos a mostrar un mensaje de error por
; pantalla llamando a pop_2_error. Es necesario llamar a esta instrucción y no a
; error porque se ha alterado el par de bits BC y se tiene que recuperar de la
; pila ( y está en la segunda posición de salida de la pila, por detrás de  el
; antiguo valor de PSW)
      mov A, C
      cmp B
      jz is_correct1
      jmp pop_2_error
is_correct1:
; Si no hay error de overflow se prosigue con el flujo de programa.
      pop PSW
      pop B
      jmp end_op

op_sub:
; en la operación resta se procede de manera similar como en la operación suma,
; también en la detección de overflow. Lo único que cambia en este caso es que
; antes de iniciar la suma, n2 se complementa en complemento a2 para invertir su
; signo y que se efectúe la resta correctamente.
      lxi H, n2
      mov A, M
      cma
      inr A

      push B
      call is_neg
```

```
      mov D, C
      add E
      push PSW
      call is_neg
      mov B, C
      mov A, E
      call is_neg
      mov A, C
      xra D
      cpi 01h
      jz is_correct2
      mov A, C
      cmp B
      jz is_correct2
      jmp pop_2_error
is_correct2:
      pop PSW
      pop B
      jmp end_op

op_and:
; En la operación and se carga n2 en A y se efectúa la operación con n2
; almacenado en E.
      lxi H, n2
      mov A, M
      ana E
      jmp end_op

op_or:
; en la operación or se procede de manera similar a la operación and
      lxi H, n2
      mov A, M
      ora E

end_op:
; Una vez se ha efectuado la operación indicada correctamente se guarda su
; resultado en la dirección sol.
      lxi H, sol
      mov M, A
; Cuando ya se tiene la solución de la operación en ca2 almacenada en sol se
; llama a la rutina convert_dec que calcula los tres dígitos decimales de sol
; y su signo para mostrarlos por pantalla.
      call convert_dec
      lxi H, sol_sign
      mov A, M
      cpi 00h
      jz sol_pos
      mvi A, 2dh
      jmp end_sol_pos
sol_pos:
      mvi A, 2bh
end_sol_pos:
      stax B
      inx B
      lxi H, sol_digits
      mov A, M
      ; a cada dígito decimal se le suma 30h para obtener su expresión en ascii
      adi 30h
      stax B
      inx B
      inx H
```

```
      mov A, M
      adi 30h
      stax B
      inx B
      inx H
      mov A, M
      adi 30h
      stax B
      ; Una vez se ha escrito la solución por pantalla se detiene el procesador
      hlt
end:
; Cuando se ha procesado la tecla introducida se incrementa el número de
; dígitos introducidos en 1 y se termina la rutina
      inr D
      ret



check_number:
; esta rutina deja A como está si es la expresión ascii de un número y lo cambia
; a 00h de lo contrario
      push D
      push H
; numbers_count contiene el número de números que hay para saber cuántas
; iteraciones debe hacer el allowed_loop como máximo para comparar el valor de
; A con la expresión de todos los números en ascii
      mvi E, numbers_count
      lxi H, numbers
allowed_loop:
      mov D, M
      cmp D
      jz end_allowed
      inx H
      dcr E
      jnz allowed_loop
not_allowed:
      mvi A, 00h
end_allowed:
      pop H
      pop D
      ret



mul:
; Esta rutina efectúa la operación A <= A + B * C, teniendo en cuenta solo
; números positivos. Además, detecta si hay overflow en el acumulador tras
; hacer la suma. Para detectarlo comprueba su bit de signo en cada suma sucesiva
; que se hace. Si es signo negativo significa que ha habido overflow.
      push PSW
      mov A, B
      ; primero si B == 0 el valor de A se mantiene
      cpi 00h
      jnz end_comp_b_0
      pop PSW
      ret
end_comp_b_0:
      ; Tras comprobar que B != 0 se suma C a A, se resta 1 a B y si B sigue
      ; siendo distinto de 0 se vuelve a ejecutar la rutina hasta que B sea 0
      ; y, por tanto, la multiplicación estará completa. Tras sumar C a A
      ; se comprueba el signo de A, si es negativo, como se ha explicado,
```

```
      ; significa que hay overflow y, por tanto, se muestra un mensaje de error
      pop PSW
      add C
      push B
      call is_neg
      mov B, A
      mov A, C
      cpi 01h
      jz pop_3_error
      mov A, B
      pop B
      dcr B
      jnz mul
      ret



div:
; Esta rutina divide de manera entera A entre C. Guarda el resto en A y el
; resultado en C
      push H
      mov H, A
      ; Primero se inicializan A y L a 0
      mvi A, 00h
      mvi L, 00h
div_loop:
; En cada iteración del bucle div_loop se suma el valor de C a A y se incrementa
; L en 1. Luego se compara el valor de A con el valor en H. Si H es mayor que A
; se vuelve a div_loop, si H == A entonces la división es exacta y si H es menor
; que A, la solución de la división es el valor de L en la anterior iteración
      add C
      inr L
      cmp H
      jz division_exacta
      jc div_loop
      dcr L
      sub C
division_exacta:
; Cuando L tiene el resultado de la división, el valor de A se le resta al valor
; de H (que es el dividendo original de la división) para obtener el resto, ya
; que en este punto A es el producto entre el cociente y el divisor de la
; división entera. De este modo el resto se almacena en A y solo queda cambiar
; a C el valor de L, que es donde está actualmente el resultado de la operación.
      mov C, A
      mov A, H
      sub C
      mov C, L
      pop H
      ret



convert_dec:
; Convierte el contenido del acumulador en tres dígitos decimales que se guardan
; en las posiciones sol_digit, sol_digit + 1 y sol_digit + 2. Pone el signo en
; sol_sign
      push H
      push B
      ; Primero se observa el signo del número del acumulador y se almacena en
      ; sol_sign. Si el signo es negativo, tras almacenarlo en sol_sign el
      ; número se complementa en a2 para poder trabajar con números positivos
```

```
      ; en la división que se ejecutará luego una vez almacenado el signo.
      call is_neg
      mov D, A
      mov A, C
      cpi 01h
      jz change_sign_to_dec
      mov A, D
      jmp end_change_sign_to_dec
change_sign_to_dec:
      lxi H, sol_sign
      mov M, A
      mov A, D
      cma
      inr A
end_change_sign_to_dec:
; Una vez se tiene el número del acumulador representado (su valor entero) con
; signo positivo primero se divide el número entre 100. el resultado es el
; primer dígito decimal, correspondiente a las decenas. El resto de esta
; división se divide por 10. El resultado de esta división es el dígito de las
; decenas y el resto el de las unidades. Entre ambas operaciones se hacen los
; guardados en memoria pertinentes de cada dígito decimal como se ha descrito.
      lxi H, sol
      mov M, A
      lxi H, sol_digits
      mvi C, 100
      call div
      mov M, C
      mvi C, 10
      call div
      inx H
      mov M, C
      inx H
      mov M, A
      pop B
      pop H
      ret



convert_bin:
; Esta rutina convierte los tres dígitos contenidos en HL, HL + 1, HL + 2 en un
; byte binario si es posible (si los tres dígitos respresentan un número entre
; -127 y 127). Toma el signo para representar este número en un byte de HL + 3.
; La representación del número en un byte binario se almacena en HL + 4.
      push PSW
      push H
      push B
      ; Primero se multiplica el primer dígito por 100 y se almacena en A
      ; el resultado tras haber inicializado A a 0 (si hay un problema de
      ; overflow la rutina mul se ocupará de enviar un mensaje de error).
      mvi A, 00h
      mov B, M
      mvi C, 100
      call mul
      ; Se repite la operación con 10 y el segundo dígito, sin inicializar A
      ; a 0.
      inx H
      mov B, M
      mvi C, 10
      call mul
      ; Por último se suma el dígito de las unidades y se comprueba si este
```

```
      ; provoca un problema de overflow a través del signo de A de manera
      ; similar a como lo hace la rutina mul
      inx H
      mov B, M
      add B
      call is_neg
      push PSW
      mov A, C
      cpi 01h
      jz pop_error
      ; Se ve qué signo tiene almacenado en la memoria el número. Si es
      ; negativo, se complementa en ca2 el reultado obtenido
      inx H
      mov A, M
      cpi 01h
      jz change_sign
      pop PSW
      jmp end_change_sign
change_sign:
      pop PSW
      cma
      inr A
end_change_sign:
; Tras haber hallado el número se almacena en la memoria, en la posición
; indicada.
      inx H
      mov M, A
      pop B
      pop H
      pop PSW
      ret



is_neg:
; Esta rutina comprueba si el primer bit de A es 1. Si lo es pone C = 1 y si no
; lo es pone C = 0. Para ello se aplica una máscara: se hace una operación and
; con A y 80h que resultará en 80h si A es negativo y 00h si A es positivo (en
; ca2). Tras aplicar la máscara se compara con 80h y en función del resultado se
; cambia el valor de C a 1 (si es negativo) o se deja en 0.
      push PSW
      mvi C, 00h
      ani 80h
      cpi 80h
      jnz end_is_neg
      mvi C, 01h
end_is_neg:
      pop PSW
      ret



pop_3_error:
; las instrucciones en las direcciones direcciones pop_n_error y error muestran
; un mensaje de error por pantalla y paran el procesador. El número n es el
; número de pops que se hacen sobre B antes de mostrar el mensaje. Esto es para
; recuperar la siguiente posición de memoria en la que se debe escribir por
; pantalla para escribir en la siguiente posición de la pantalla si el par
; BC ha sido alterado antes de lanzar el mensaje de error pero el valor
; original está en la pila y por tanto se puede recuperar sacando el número de
; elementos de la pila correctos.
```

```
      pop B
pop_2_error:
      pop B
pop_error:
      pop B
error:
      mvi A, 45h
      stax B
      inx B
      mvi A, 52h
      stax B
      inx B
      stax B
      inx B
      mvi A, 4fh
      stax B
      inx B
      mvi A, 52h
      stax B
      hlt
```

Cuestión 1

¿Qué diferencia hay entre la suma y el OR?

ii) la OR es una operación lógica mientras que la suma es aritmética.

Cuestión 2

La instrucción STA 1234h

ii) Utiliza direccionamiento directo (ya que la dirección de memoria en la que se va a almacenar el contenido del acumulador está especificada en la propia instrucción)

Conclusiones y aprendizajes

Tras finalizar la práctica he aprendido cómo se pueden formar programas más complejos en 8085 fragmentándolos en rutinas más simples que ejecuten tareas concretas como sumar dos números metidos por input. Además he aprendido cómo puedo gestionar los inputs que se entran por teclado según el significado que se les dé. Por otro lado, como mejora a mi práctica ,creo que podría llegar a sumar y restar números hasta 999 si implementamos la suma y la resta con el algoritmo que se usa para sumar y restar a mano usando los dígitos de las decenas centenas y unidades, aprovechando que los números se insertan ya con esta distinción . Además de este modo no hay que hacer conversiones entre binario y decimal en esta operación.