Informe de la práctica 5

Introducción

En esta 5ª práctica de Introducción a los Ordenadores encontraremos los objetivos, los ejercicios planteados y conclusiones:

- Conocer los modos de direccionamiento del i8085 y familiarizarse con sus instrucciones.

Ejercicio I. Modos de direccionamiento del i8085

Escribid dos programas diferentes para sumar dos matrices de 5x1 y guardad el resultado en una tercera, podeis utilizar direccionamiento indirecto, por pareja de registros o por registros individuales.

Pensad en dos hipótesis: primera, guardamos el resultado en una de las dos matrices de entrada, mat3 es irrelevante; segunda, las matrices de entrada se han de conservar, es decir, el resultado se almacena sobre mat3.

El programa debería utilitzar un contador, instrucciones de incremento, comparación y salto condicional. Haced la suma de números hexadecimales y sin tener en cuenta números que puedan dar overflow.

Calculad las medidas del código de vuestro programa y el número de ciclos para su ejecución. (*VÉASE EXPLICACIÓN DEL CÓDIGO*)

Usad la ayuda del programa simulador para conocer las instrucciones del i8085 y sus modos de direccionamiento, también ayudaros de las explicaciones del apartado de teoría.

Tened en cuenta que no todas las instrucciones soportan todos los modos de direccionamiento. Haced la suma de números hexadecimales y sin tener en cuenta números que puedan dar overflow.

El inicio del programa será el siguiente:

```
.define
        num 5
.data 00h
        mat1: db 1,2,3,4,5
        mat2: db 6,7,8,9,0
        mat3: db 0,0,0,0,0
.org 100h
```

PROGRAMA 1: sumamos mat1 + mat2 y guardamos en mat2

```
.define
        num 5
.data 00h
        mat1: db 1,2,3,4,5
        mat2: db 6,7,8,9,0
        mat3: db 0,0,0,0,0
.org 100h
        LXI H, mat1     ; HL <= 00h (cargamos la posición 00h
                          inicicialmente) [ 10 ciclos ]
        LXI B, mat2     ; BC <= 05h (cargamos la posición 05h
                          inicialmente) [ 10 ciclos ]
        MVI D, num      ; D <= 5 (cargamos el inmediato 5)
                          [ 7 ciclos ]
loop:
        MOV E, M        ; cargamos el contenido de la dirección
                          00h (inicialmente, 1) en el registro E
                          [ 7 ciclos ]
        LDAX B          ; guardo en acumulador el contenido de par
                          de registros BC ( direccionamiento indirecto )
                          [ 7 ciclos ]
```

```
        ADD E       ; sumo acumulador y el contenido del registro
                    E, guardando el resultado en acumulador
                    [ 4 ciclos ]
        STAX B      ; guardo la suma en la posición donde había
                    inicialmente un 6 (posición 05h) [ 7 ciclos ]
        INX H       ; incrementamos el par HL [ 6 ciclos ]
        INX B       ; incrementamos el par BC [ 6 ciclos ]
        DCR D       ; decrementamos el registro D [ 4 ciclos ]
        JNZ loop    ; mientras no sea 0, saltamos a loop
                    [ 7 o 10 ciclos, según si saltamos o no ]

hlt                 ; fin del programa [ 4 ciclos ]
```

En resumen, el primer programa realiza la suma de 2 matrices 5x1 y guarda el resultado en una de dichas matrices sobreescribéndola.

Ahora calcularemos tanto los ciclos que tarda en ejecutarse como las medidas del código. En cuanto a los ciclos tenemos que:

I. Para cargar los datos utilizamos 2 instrucciones LXI para las matrices 1 y 2 y una instrucción MVI para el contador: (10 x 2) +7 = 27

II. En el loop tenemos 2 casos:

   a) Si no se cumple la condición de salto de JNZ

      Para realizar la suma de matrices utilizamos las siguientes instrucciones: MOV (7 ciclos), LDAX (7 ciclos), ADD (4 ciclos), STAX (7 ciclos), 2 INX (6 ciclos), DCR (4 ciclos) y JNZ (7 ciclos).

      Si sumamos todo, el número total es: (7 x 4) + (6 x 2) + (4 x 2) = 48 ciclos. Sabiendo que este caso se repite 4 veces, al final son (48 x 4) = 192 ciclos.

   b) Si sí se cumple la condición de salto de JNZ

      Utilizamos las mismas instrucciones que en el caso (a), pero JNZ tardará 10 ciclos en vez de 7. Por lo tanto, son 3 ciclos más y solo se repite 1 vez. Esto implica que tenemos 51 ciclos.

III. Por último, tenemos la instrucción de fin del programa HLT, que tarda 4 ciclos en realizarse.

Si sumamos todo obtenemos: 27 + 192 + 51 + 4 = **274 ciclos**

En cuanto a la medida del código debemos remitirnos a la memoria de instrucciones del i8085, que trabaja en hexadecimal:

Memoria (Instrucciones)

Dirección	Nemotécnico	Código
0100	LXI H,0000H	21
0101		00
0102		00
0103	LXI B,0005H	01
0104		05
0105		00
0106	MVI D,05H	16
0107		05
0108	MOV E,M	5E
0109	LDAX B	0A
010A	ADD E	83
010B	STAX B	02

"Memoria desde 100h hasta 10Bh"

Memoria (Instrucciones)

Dirección	Nemotécnico	Código
0107		05
0108	MOV E,M	5E
0109	LDAX B	0A
010A	ADD E	83
010B	STAX B	02
010C	INX H	23
010D	INX B	03
010E	DCR D	15
010F	JNZ 0108H	C2
0110		08
0111		01
0112	HLT	76

"Memoria desde 107h hasta 112h"

Estudiaremos las imágenes en decimal para mayor comprensión.

El programa empieza en la posición 100h , que en decimal es el número 256, y termina en la 112h, que es el número 274.

Por lo tanto usamos 18 posiciones de la memoria de instrucciones para el código, a los que hay que añadir las posiciones ocupadas para las matrices 1 y la matriz resultante, que se sobreescribirá encima de la matriz 2. Éstas ocupan las posiciones 00h hasta la posición 08h, que corresponden a los mismos números en decimal. En total tendremos 18 + 9 = 27 posiciones ocupadas.

Memoria (Instrucciones)

Dirección	Nemotécnico	Código
0000	LXI B,0302H	01
0001	STAX B	02
0002	INX B	03
0003	INR B	04
0004	DCR B	05
0005	RLC	07
0006	DAD B	09
0007	DCX B	0B
0008	DCR C	0D
0009	DCR B	05
000A	NOP	00
000B	NOP	00

"mat1 y el resultado en la memoria"

PROGRAMA 2: sumamos mat1 + mat2 y guardamos en mat3

```
.define
    num 5
.data 00h
    mat1: db 1,2,3,4,5
    mat2: db 6,7,8,9,0
    mat3: db 0,0,0,0,0
.org 100h
    LXI B, mat1      ; BC <= 00h (cargamos la posición 00h
                     inicicialmente) [ 10 ciclos ]
    LXI D, mat3      ; DE <= 0Ah (cargamos la posición 0Ah
                     inicicialmente) [ 10 ciclos ]
    MVI H, num       ; H <= 5 (cargamos el inmediato 5)
                     [ 7 ciclos ]

loopCopia:           ; copiamos mat1 en mat3
    LDAX B           ; guardamos en acumulador el
                     contenido de par de registros BC
                     (direccionamiento indirecto) [ 7 ciclos ]
    STAX D           ; guardo acumulador en la posición
                     en DC [ 7 ciclos ]
    INX B            ; incrementamos el par BC [ 6 ciclos ]
    INX D            ; incrementamos el par DE [ 6 ciclos ]
    DCR H            ; decrementamos el registro H [4 ciclos]
    JNZ loopCopia    ; mientras H != 0, saltamos a loopCopia
                     [ 7 o 10 ciclos, según si saltamos o no ]

    LXI H, mat2      ; HL <= 05h (cargamos la posición 05h
                     inicicialmente) [ 10 ciclos ]
```

```
        LXI B, mat3     ; BE <= 0Ah (cargamos la posición 0Ah
                        inicicialmente) [ 10 ciclos ]
        MVI D, num      ; D <= 5 (cargamos el inmediato 5)
                        [ 7 ciclos ]

loopSuma:
        MOV E, M        ; cargamos el contenido de la dirección
                        05h (inicialmente, 6) en el registro E
                        [ 7 ciclos ]
        LDAX B          ; guardamos en acumulador el
                        contenido de par de registros BC
                        (direccionamiento indirecto) [ 7 ciclos ]
        ADD E           ; sumo acumulador y el contenido del
                        registro E, guardando el resultado en
                        acumulador [ 7 ciclos ]
        STAX B          ; guardo la suma en la posición donde
                        había inicialmente un 1 (posición 0Ah)
                        [ 7 ciclos ]
        INX H           ; incrementamos el par HL [ 6 ciclos ]
        INX B           ; incrementamos el par BC [ 6 ciclos ]
        DCR D           ; decrementamos el registro D
                        [ 4 ciclos ]
        JNZ loopSuma    ; mientras no sea 0, saltamos a loop
                        [ 7 o 10 ciclos, según si saltamos o no ]

hlt                     ; fin del programa [ 4 ciclos ]
```

En resumen, el segundo programa realiza la suma de 2 matrices 5x1 y guarda el resultado en una tercera matriz. Por lo tanto, no sobreescribimos ninguna de las matrices que se suman.

Ahora calcularemos tanto los ciclos que tarda en ejecutarse como las medidas del código. En cuanto a los ciclos tenemos que:

I. Para cargar los datos, utilizamos 2 instrucciones LXI para las matrices 1 y 3 y una instrucción MVI para el contador: (10 x 2) +7 = 27

II. En el loopCopia tenemos 2 casos:

a) Si no se cumple la condición de salto de JNZ

Para sobreescribir la matriz 1 sobre la matriz 3, que está vacía, utilizamos las siguientes instrucciones: LDAX (7 ciclos), STAX (7 ciclos), 2 INX (6 ciclos), DCR (4 ciclos) y JNZ (7 ciclos).

Si sumamos todo, el número total es: (7 x 3) + (6 x 2) + 4 = 37 ciclos. Sabiendo que se repite este caso 4 veces, al final son (37 x 4) = 148 ciclos.

b) Si sí se cumple la condición de salto de JNZ

Utilizamos las mismas instrucciones que en el caso (a), pero JNZ tardará 10 ciclos en vez de 7. Por lo tanto, son 3 ciclos más y solo se repite 1 vez. Esto implica que tenemos 40 ciclos.

III. Después del loopCopia, cargamos la matriz 2 para sumarla con la matriz 3, es decir, tenemos 2 instrucciones LXI para cargar las posiciones de memoria y una MVI para reestablecer el contador: (10 x 2) +7 = 27

IV. En el loopSuma también tenemos 2 casos:

a) Si no se cumple la condición de salto de JNZ:

Para realizar la suma de matrices, utilizamos las siguientes instrucciones: MOV (7 ciclos), LDAX (7 ciclos), ADD (4 ciclos), STAX (7 ciclos), 2 INX (6 ciclos), DCR (4 ciclos) y JNZ (7 ciclos).

Si sumamos todo, el número total es: (7 x 4) + (6 x 2) + (4 x 2) = 48 ciclos. Sabiendo que se repite este caso 4 veces, al final son (48 x 4) = 192 ciclos.

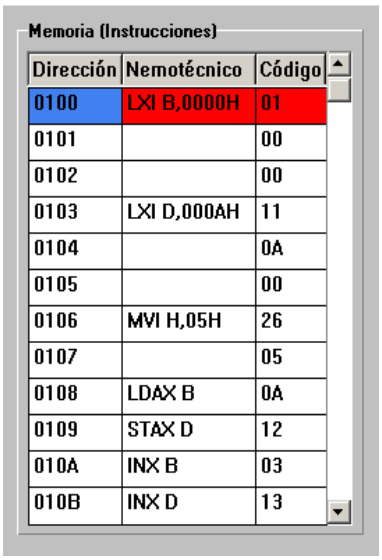
b) Si sí se cumple la condición de salto de JNZ:

Utilizamos las mismas instrucciones que en el caso (a), pero JNZ tardará 10 ciclos en vez de 7. Por lo tanto, son 3 ciclos más y solo se repite 1 vez. Esto implica que tenemos 51 ciclos.

V. Por último, tenemos la instrucción de fin del programa HLT, que tarda 4 ciclos en realizarse.

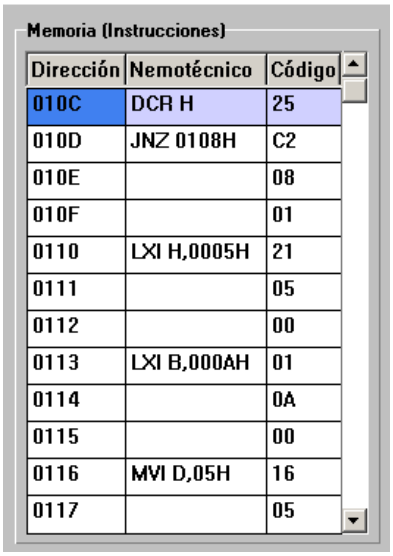
Si sumamos todo tenemos: (27 x 2) + 148 + 40 + 192 + 51 = **485 ciclos**

En cuanto a la medida del código, debemos remitirnos a la memoria de instrucciones del i8085, que trabaja en hexadecimal. Estudiaremos las imágenes en decimal para mayor comprensión:


Memoria (Instrucciones)

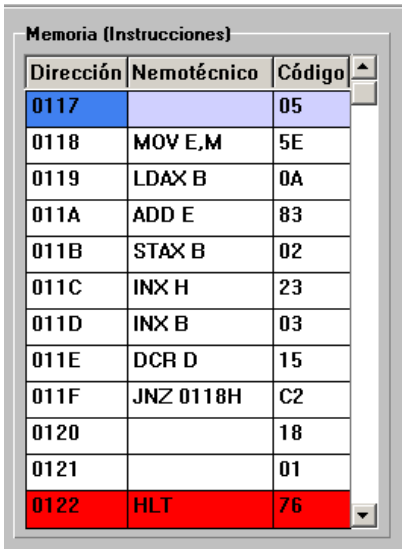
Dirección	Nemotécnico	Código
0100	LXI B,0000H	01
0101		00
0102		00
0103	LXI D,000AH	11
0104		0A
0105		00
0106	MVI H,05H	26
0107		05
0108	LDAX B	0A
0109	STAX D	12
010A	INX B	03
010B	INX D	13

"Memoria de 100h hasta 10Bh"


Memoria (Instrucciones)

Dirección	Nemotécnico	Código
010C	DCR H	25
010D	JNZ 0108H	C2
010E		08
010F		01
0110	LXI H,0005H	21
0111		05
0112		00
0113	LXI B,000AH	01
0114		0A
0115		00
0116	MVI D,05H	16
0117		05

"Memoria de 10Ch hasta 117h"


Memoria (Instrucciones)

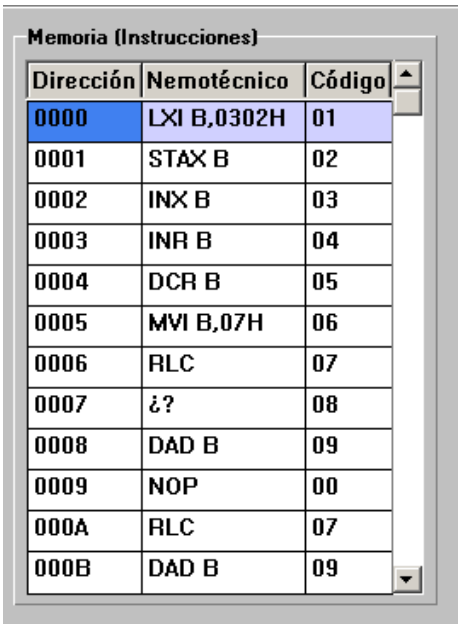
Dirección	Nemotécnico	Código
0117		05
0118	MOV E,M	5E
0119	LDAX B	0A
011A	ADD E	83
011B	STAX B	02
011C	INX H	23
011D	INX B	03
011E	DCR D	15
011F	JNZ 0118H	C2
0120		18
0121		01
0122	HLT	76

"Memoria de 117h hasta 122h"

El programa 2 también empieza en la posición 100h, que en decimal es 256, pero termina en la 122h, que la posición 290.

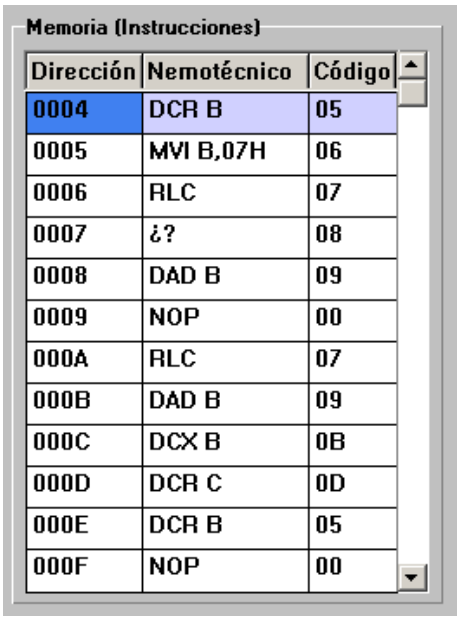
Por lo tanto, usamos 34 posiciones de la memoria de instrucciones para el código, a los que hay que añadir las posiciones ocupadas para las 2 matrices a sumar y la resultante. Éstas ocupan las posiciones 00h hasta la 0Eh, que corresponden en decimal a las posiciones 0 hasta la 14. En total tendremos 34 + 15 = 39 posiciones ocupadas.

En las siguientes imágenes vemos que mat1 ocupa las posiciones 00h hasta 04h; mat 2, desde 05h hasta 09h; el resultado en mat3, desde 0Ah hasta 0Eh:


Memoria (Instrucciones)

Dirección	Nemotécnico	Código
0000	LXI B,0302H	01
0001	STAX B	02
0002	INX B	03
0003	INR B	04
0004	DCR B	05
0005	MVI B,07H	06
0006	RLC	07
0007	¿?	08
0008	DAD B	09
0009	NOP	00
000A	RLC	07
000B	DAD B	09

"mat1 y mat2 del código 2"


Memoria (Instrucciones)

Dirección	Nemotécnico	Código
0004	DCR B	05
0005	MVI B,07H	06
0006	RLC	07
0007	¿?	08
0008	DAD B	09
0009	NOP	00
000A	RLC	07
000B	DAD B	09
000C	DCX B	0B
000D	DCR C	0D
000E	DCR B	05
000F	NOP	00

"mat2 y el resultando en mat3"

1. Pregunta 1

¿En qué simplificaría mucho el código del programa uno de los modos de direccionamiento del simulador Ripes?

El ADD en Ripes permite sumar el contenido de 2 registros y guardarlo en un registro, sea este último alguno de los 2 sumandos u otro distinto. Esto nos reduciría en varias líneas y ciclos el código, ya que no tenemos que ultilizar la instrucción MOV del i8085 para poder sumar las matrices ni estaríamos obligados usar el acumulador siempre. Además en Ripes hay muchos más registros de propósito general, mientras que en i8085 solo tenemos 6 disponibles.

2. Pregunta 2

¿Cuántos ciclos de reloj tarda en ejecutarse una instrucción aritmético – lógica cualquiera? Utilizad el fichero adjunto donde especifica el ISA del 8085. Indica cuál es la medida media de tus instrucciones (*VÉANSE LOS COMENTARIOS DEL CÓDIGO*). Calcula los ciclos por instrucción medios para estos códigos.

Depende de la instrucción aritmético-lógica. Entre estas instrucciones tenemos ADD, ADI, DCR o DCX, SUB, etc. Si nos fijamos, algunas de ellas se pueden usar tanto para un único registro o para un par de registros, como HL.

Por ejemplo, la instrucción ADD puede ir con un registro, tardando solo 4 ciclos de reloj, o con M (es decir, el par HL), tardando 7 ciclos en total. Esto se debe a que :

- ADD reg: sumamos el byte de datos contenido en el registro especificado con el contenido de acumulador y lo guardamos en acumulador. Por lo tanto, es más directa que la siguinte.
- ADD M: sumamos el contenido de la posición de memoria direccionada por los registros HL con el contenido de acumulador y lo guardamos en acumulador. Es decir, debemos acceder a la posición de memoria contenida en los registros y, después, al contenido de dicha posición. Todo ello para poder hacer la suma del valor que haya con el contenido de acumulador.

Por lo tanto, vemos que hay una clara diferencia entre la primera y la segunda, porque ADD M usa direccionamiento indirecto. Las instrucciones aritmético-lógicas varían entre 4 ciclos, como ADD reg o INR reg, y 10 ciclos, como DCR M o DAD RP. En un plano más general, como los programas de arriba, la media de ciclos por instrucción ronda unos 7 ciclos.

3. Pregunta / Tarea 3

Subid vuestro código y marcad cuál es la instrucción de vuestro programa que tarda más ciclos en ejecutarse.

En ambos programas, la instrucción que más tarda es LXI (10 ciclos), que sirve para cargar la posición de memoria de los números de las matrices a un par de registros. También cabe comentar que la instrucción de salto condicional JNZ puede llegar a tardar 10 ciclos en algunos casos, aunque normalmente solo sean 7 ciclos.

En los códigos, estas 2 instrucciones están marcadas de forma: ***INSTRUCCIÓN X, Y.***

4. Pregunta / Tarea 4

Traducid el código para hacerlo servir con el simulador Ripes. ¿Cuántos ciclos tarda en ejecutarse? Comparad los resultados (medida de código, accesos a memoria y ciclos promedio por instrucción) con los valores obtenidos para i8085.

Código en Ripes:

```
.data
        mat1: .word 1,2,3,4,5
        mat2: .word 6,7,8,9,0
        mat3: .word 0,0,0,0,0
        num: .word 5


.text
        la a1, mat1             #cargamos las direcciones de mat1,
        la a2, mat2             # mat2,
        la a3, mat3             # mat3 y
        la a7, num              # num, que es el contador
        lw a7, 0(a7)            # cargamos el número 5


loop:
        lw a4, 0(a1)            # cargamos el valor de mat1 en a4
                                # (inicialmente 1)
        lw a5, 0(a2)            # cargamos el valor de mat2 en a5
                                # (inicialmente 6)
        add a6, a4, a5          # sumamos a6 <= a4 + a5
        sw a6, 0(a3)            # guardamos el resultado en mat3
        addi a7, a7, -1         # decrementamos el contador
        addi a1, a1, 4          # pasamos a la siguiente posición de
        addi a2, a2, 4          # las 3 matrices
```

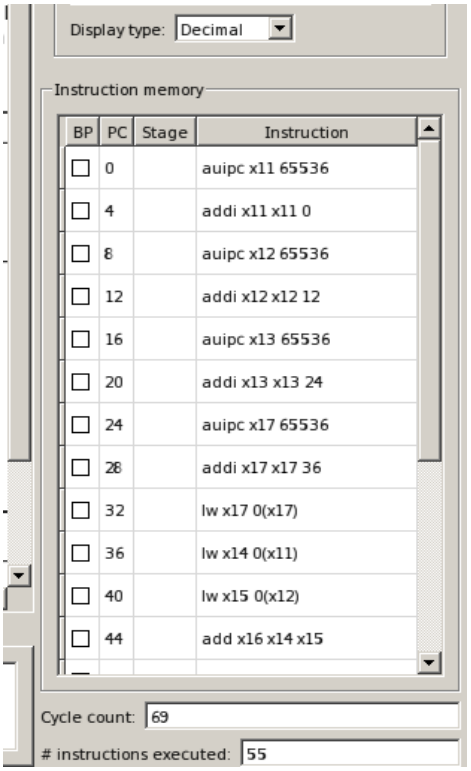
```
        addi a3, a3, 4
        bgt a7, zero, loop          # mientras el contador no sea 0, loop

        nop                         # fin del programa
```

Este código en Ripes tarda 69 ciclos que, comparado con los 274 ciclos del primer promagrao los 485 del segundo programa en i8085, es mucho menor.

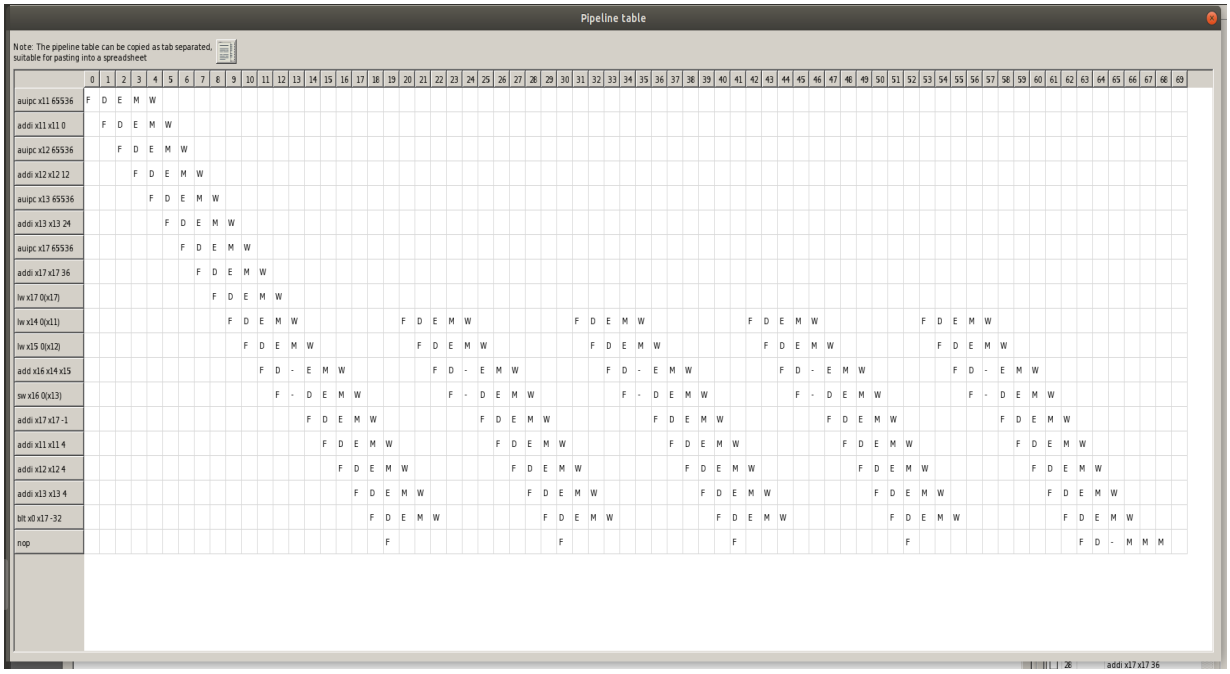
Esto se debe a que, en los programas de i8085, las intrucciones tardan entre 4 y 10 ciclos en realizarse. Sin embargo, en Ripes la mayoría tardan 5, alguno incluso 6 por un nop(stall).

Además, en cuanto a líneas de cógido, en Ripes tenemos 22 líneas, sin espacios ni comentarios. Si observamos los códigos en i8085, en presentan normalmente más líneas y, por ello, también más ciclos totales.

Por últimos, los accesos a memoria en Ripes son más directos que en el i8085. Otra causa de su reducción de ciclos.



Ciclos en Ripes



"Pipeline del código en Ripes, teniendo 69 ciclos totales"

Ejercicio 2: Subrutinas

Haciendo uso del programa anterior, realizad una subrutina que codifique una zona de memoria. La zona de memoria se indicará poniendo al registro doble HL la dirección de comienzo de la zona de datos a codificar.

Estos datos se consideran como los parámetros de entrada de la subrutina. La codificación se hará mediante una XOR entre cada byte de la zona de datos y la llave. Los valores de los registros no han de quedar afectados por la llamada a la subrutina.

Haced un programa que haga uso de esta subrutina para probarla con diferentes combinaciones de valores de la llave y los datos para codificar.

Ayuda: Llama a una subrutina en i8085:

```
call nom_subrutina
...
.org
adreça_subrutina:
PUSH...
/* Codi de la subrutina aquí  TODO...*/
POP...
RET
```

Ayuda: Llama a una subrutina en Ripes:

```
call nom_subrutina
...
.text
adreça_subrutina:
        addi sp,sp,-16 # Reservar stack frame
        sw s1,4(sp)
        sw s0,8(sp)
/* Codi de la subrutina aquí TO DO...*/
        lw, s0, 8(sp)
        lw s1, 4(sp)
        addi sp, sp, 16
        ret
```

No todos los registros utilitzados en una subrutina se tienen que preservar. I8085 solo preservará los registros guardados explícitamente en la pila, con PUSH i POP. Lo mismo pasa con Ripes. En este caso, dado el número de registros del que dispone esta arquitectura, hay unos registros más focalizados en ser utilizados como registros temporales y otros que requieren ser preservados.

Por ejemplo, los saved registers (s0-s11) se llaman así porque la subrutina los tiene que preservar, mientras que los temporary registers (t0-t6) no hace falta (el caller no puede asumir que se preservaran después de una llamada a función).

Otros tipos de llamadas a subrutinas: Tail calls

En informática, una llamada de cola o tail call es una llamda de subrutina realizada como acción final de un procedimiento. Estas subrutinas hacen un uso mucho más sencillo y simplificado de la pila. Ejemplos de Tail calls son:

```
function foo(data) {
        a(data);
        return b(data);
}
```

Aquí, tanto *a(dada)* como *b(dada)* son llamadas a subrutines, pero b es el último que se ejecuta la función antes de volver, encontrándose así en la posición de la cola. b(dada) es un Tail call. No obstante, no todas las llamadas de cola se encuentran necesariamente al extremo sintáctico de una subrutina:

```
function bar(data) {
        if ( a(data) ) {
                return b(data);
        }
        return c(data);
}
```

Aquí tanto b(data) como c(data) son Tail calls.

PROGRAMA i8085 CON SUBRUTINA: sumamos mat1 + mat2 y guardamos en mat3

```
.define
        num 5
        clau 55h
.data 00h
        mat1: db 1,2,3,4,5
        mat2: db 6,7,8,9,0
        mat3: db 0,0,0,0,0
.org 100h
        LXI H, mat1   ; HL <= 00h (cargamos la posición 00h
                        inicicialmente)
        LXI B, mat2   ; BC <= 05h (cargamos 05h inicialmente)
        MVI D, num    ; D <= 5 (cargamos el inmediato 5)
loop:
        MOV E, M      ; cargamos el contenido de la dirección
                        00h (inicialmente, 1) en el registro E
        LDAX B        ; guardo en acumulador el contenido de par
                        de registros BC (direccionamiento indirecto)
        ADD E         ; sumo acumulador y el contenido del registro
                        E, guardando el resultado en acumulador
        STAX B        ; guardo la suma en la posición donde había
                        inicialmente un 6 (posición 05h)
        INX H         ; incrementamos el par HL
        INX B         ; incrementamos el par BC
        DCR D         ; decrementamos el registro D
        JNZ loop      ; mientras no sea 0, saltamos a loop
        call encripta ; guardamos el contenido de PC en la pila
hlt
```

```
.org 25h
encripta:    ; encriptamos el resultado y lo guardamos en mat3
        PUSH psw        ; guarda el contenido de AF en la pila
        PUSH H          ; guarda el contenido de HL en la pila
        PUSH B          ; guarda el contenido de BC en la pila
        PUSH D          ; guarda el contenido de DE en la pila
        MVI E, num      ; E <= 5 (cargamos el inmediato 5)
        LXI H, mat2      ; HL <= 05h (cargamos la posición 05h
                         inicicialmente)
        LXI B, mat3      ; HL <= 0Ah (cargamos la posición 0Ah
                         inicicialmente)

        sloop:
                MOV A,M     ; guardamos el 1º valor de mat2 en
                            acumulador
                XRI clau    ; hacermos Acumulador XOR clau
                STAX B      ; guardo la suma en la posición 0Ah
                            inicialmente
                INX H       ; incrementamos el par HL
                INX B       ; incrementamos el par BC
                DCR E       ; decrementamos el registro E
                JNZ sloop   ; mientras no sea 0, saltamos a sloop

                POP D       ; retornamos el par de registros DE
                POP B       ; retornamos el par de registros BC
                POP H       ; retornamos el par de registros HL
                POP psw     ; retornamos el par de registros AF
                ret         ; retornamos el PC (estructura LIFO)
```

Pregunta 4

¿Qué instrucción usamos en los dos casos para asignar la posición inicial al registro SP?

Usamos la instrucción PUSH. Esta instrucción copia 2 bytes de datos en el stack, que pueden ser el contenido de un par de registros o la «palabra de estado del programa».

Además resta 1 del stack pointer y copia el contenido del registro de «alto orden» del par de registros en la dirección resultante. Justo después, se resta otra vez 1 al stack pointer y se copia el registro de «bajo orden» en la dirección resultante, es decir, los registros no varían.

Pregunta 5

¿Cuál es la instrucción utilizada para guardar el PC en la pila cuando trabajamos con subrutines? ¿Y para recuperar de nuevo el valor del PC?

Utilizamos la instrucción «call» para guardar el contenido del PC en el stack y para recuperar el valor de nuevo, la instrucción «ret».

Ambas instrucciones su usan en subrutinas: la primera, «call», se usa para llamar a las subrutinas. Una vez hayamos realizado todas las instrucciones de dicha subrutina, usaremos la instrucción «ret» para salir. Ésta nos devuelve a la línia de código siguiente a su respectivo call *<subrutina>.*

Conclusiones

En esta práctica hemos introducido un nuevo simulador: i8085. Además hemos ampliado conocimientos en Ripes con la comparación de ambos simuladores. Todo a través de:

- *Realizar los ejercicios 1 y 2, con sus diversas preguntas ,gracias al directo de youtube y los PDF's de ayuda para el i8085.*
- *Con la guia de «Set_Instruccions_8085», hemos encontrado todas las instrucciones necesarias para realizar los 3 programas en i8085 y saber sus ciclos, sus direccionamientos, etc...*
- *En cuanto al programa en Ripes, hemos observado las direfencias que hay entre ambos simuladores a la hora de realizar una misma operación: medidas de código, ciclos, instrucciones...*
- *Hemos introducido y empleado el nuevo concepto de subrutina.*

En resumen, doy por cumplidos los objetivos propuestos más arriba, aunque faltaría mucho más rodaje en el simulador i8085.