Introducció als Ordinadors

Pràctica 7

Mario Vilar

19 de juny de 2022

Índex

1 Introducció **2**

1.1 Objectius 2

2 Cos del treball **3**

2.1 Part guiada 3

2.2 Part no guiada 5

2.2.1 Primera part 5

2.2.2 Segona part 9

2.2.3 Tercera part 10

2.2.4 Codi sencer 11

2.3 Epíleg 18

1 Introducció

El micro-processador 8085 permet capturar caràcters introduïts per teclat mitjançant una interrupció `trap`. Aquests caràcters es poden emmagatzemar en format ASCII hexadecimal a una regió específica de memòria que implementa una 'pantalla de text'. D'aquesta manera, és possible mostrar missatges introduïts pel consola per pantalla. De forma més general, podem interactuar amb el micro- processador 8085 introduint dades per la consola, que seran tractades pel micro segons les instruccions del nostre codi i, posteriorment, mostrades com a resultat per pantalla.

En aquesta pràctica, farem precisament això: Introduirem per consola un parell de nombres i un operador, el micro-processador 8085 realitzarà una operació aritmetico-lògica amb aquest nombres i mostrarem el resultat de l'operació per pantalla. Aquesta tasca es pot realitzar de diverses maneres, segons les prestacions que exigim al nostre codi.

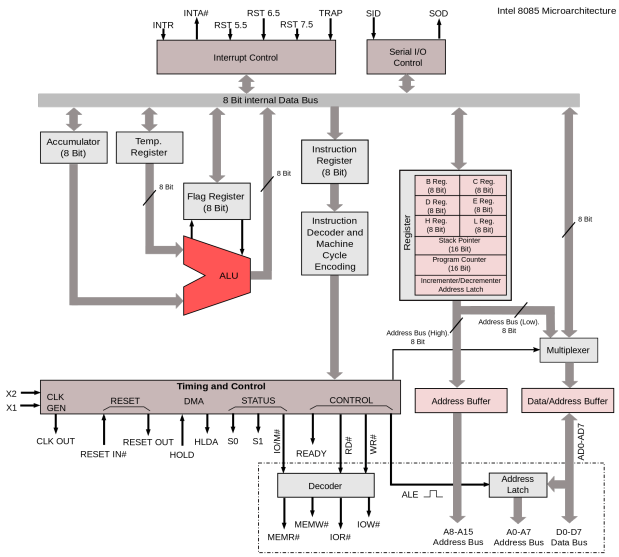

Figura 1: Esquema del processador 8085.

1.1 Objectius

L'objectiu d'aquesta pràctica és fer un programa més complex amb el simulador 8085 aplicant tot el que hem fet fins ara. En concret el programa ha de fer diferents operacions aritmeticològiques depenent de quins *inputs* rebi i posteriorment ha de mostrar els resultats per pantalla.

2

Cos del treball

2.1 Part guiada

Exercici 1. *Analitza, amb l'ajuda del professor el següent codi. Carrega'l al simulador i executa'l pas a pas. Fixa't en quina és la primera direcció de memòria de text del 8085. Quina és la funció del parell de registres BC en aquest codi?*

```
.org 100h ; Posicio Pila
pila:
.org 200h
; Programa Principal
lxi h, pila     ; Punter de pila apuntant a 100h
sphl
mvi b, E0h      ; Parell BC apuntant
mvi c, 00h      ; a la memoria de text
bucle:          ; Loop infinit
    jmp bucle
.org 0024h      ; Direccio de interrupcio TRAP
call string_in  ; Crida a subrutina d introduccio
ret             ; de dades per consola
.org 300h
; Rutina captura i mostra
string_in:
in 00h          ; Port d'entrada
cpi 00h         ; Si no hi ha caracter introduit, surt
jz no_tecla     ; Si hi ha, escriu-lo per pantalla
tecla:
stax b
inx b
no_tecla:
ret
```

BC ens serveix per posar en contacte la pantalla de text i les entrades que fem per teclat. D'aquesta manera, aquest registre ens permet establir una correspondència entre les posicions en la pantalla, que s'anirà incrementant per cada caràcter escrit, i el text que vulguem escriure. És important, en concret, la instrucció `inx b`; si no hi fos, estaríem reescrivint sobre la mateixa posició cada vegada i solament es mostraria un caràcter.

És interessant observar que inicialitzem el valor de BC a E000h: convé associar com el port de la pantalla de text E000h. En efecte, podrem escriure fins al valor E3E8h, és a dir, 1000 bytes de dades.

Exercici 2. *Modifica el codi posterior per tal que només es puguin introduir per teclat valors numèrics del 0 al 9 i els operadors +, -, &, |. Executa'l per veure com funciona.*

```
.define
allowed_count 15
.data 00h
allowed: db 30h,31h,32h,33h,34h,35h,36h,37h,38h,39h,
            2Bh,2Dh,26h,7Ch,3Dh
.org 100h
pila:
.org 200h
; Programa Principal
lxi h,pila
sphl
lxi b, E000h
bucle:
jmp bucle
.org 0024h
call string_in
ret
.org 300h
; Rutina captura i mostra
string_in:
in 00h
cpi 00h
jz no_tecla
tecla:
call check_allowed
cpi 00h
jz no_tecla
stax b
inx b
no_tecla:
ret
check_allowed:
push d
push h
mvi e, allowed_count
lxi h, allowed
allowed_loop:
mov d,m
cmp d
jz is_allowed
inx h
dcr e
jnz allowed_loop
jmp not_allowed
is_allowed:
mov a,d
jmp end_allowed
not_allowed:
```

```
mvi a,00h
end_allowed:
pop h
pop d
ret
```

Aquest programa és llarg i força més complex que l'anterior. El comentarem breument. Primerament, es defineixen els caràcters que permetrem imprimir per pantalla i la posició de la pila.

Carreguem al parell de registres HL la posició de pila, que és 100h. Es carrega a sp el valor contingut en HL i fem que el registre BC sigui el pont connector entre la pantalla i el teclat: el parell BC apuntarà a la memòria de text. Entrem al bucle infinit, esperant l'entrada.

Detectem l'entrada de text i s'activa trap, anem a la posició 24h de la memòria d'instruccions (la direcció d'interrupció trap) i cridem a la subrutina d'introducció de dades per consola.

string-in sembla una rutina de captura i mostra: passem l'entrada a l'acumulador pel port 00h i comprovem que s'introdueix algun caràcter i estigui, en efecte permès. Per fer-ho, cridem a la subrutina check-allowed i comprovem que la seva sortida estigui permesa, guardant en cas afirmatiu el valor de l'acumulador en la memòria de text (en cas contrari, carreguem un zero).

En el check-allowed hi trobem un loop que ens permetrà comprovar si el valor de l'acumulador coincideix amb algun dels valors que es troba en la memòria de dades, en particular aquells que hem definit al principi. Haurem de distingir casos per tractar les entrades.

2.2 Part no guiada

2.2.1 Primera part

Exercici 3. *Dissenyeu una subrutina que a partir de dos nombres (base 10) introduïts pel teclat del simulador i8085 faci la suma i presenti el resultat en la pantalla de text del i8085. Feu servir els adreçaments directe i indirecte i indiqueu al codi on tenim aquests adreçaments. Com gestioneu el problema del signe? Com gestioneu el problema del overflow?*

Demostració. El codi de la nostra interrupció TRAP serveix per anar guardant a memòria allò que ens introdueixen per teclat. Només saltarem a una altra part del codi si ens introdueixen la igualtat = (3Dh), ja que és el nostre indicador que se'ns ha introduït tot allò necessari per fer una operació i procedirem a tractar-la. Farem tres versions, per si estem parlant d'operacions amb un, dos o tres nombres.

S'ha implementat una subrutina per netejar la pantalla.

```
.define
allowed_count 15
.data 00h
allowed: db 30h,31h,32h,33h,34h,35h,36h,37h,38h,39h,
            2Bh,2Dh,26h,7Ch,3Dh
```

```
.org 100h
pila:
.org 200h
; Programa Principal
lxi h,pila
sphl
lxi b, E000h
bucle:
jmp bucle
.org 0024h
call string_in
call operation
ret
.org 300h
; ----------------------------------
string_in:
in 00h
cpi 00h
jz no_tecla
tecla:
call check_allowed
cpi 00h
jz no_tecla
cpi 43h
jz clear
stax b
inx b
no_tecla:
ret
check_allowed:
push d
push h
mvi e, allowed_count
lxi h, allowed
allowed_loop:
mov d,m
cmp d
jz is_allowed
inx h
dcr e
jnz allowed_loop
jmp not_allowed
is_allowed:
mov a,d
jmp end_allowed
not_allowed:
mvi a,00h
end_allowed:
```

```
pop h
pop d
ret
clear:
lxi h,0000h
mov l,c
clear_loop:
mvi a,00h
stax b
dcx b
dcx h
add l ; activem bit zero si cal
jnz clear_loop
stax b
ret
; ------------------------------------
operation:
cpi 3Dh
jz trigger
ret
trigger:
call operate
ret
operate:
push b
lxi b,e000h ; direccionament immediat
ldax b
mov e,a ; direccionament directe
inx b ; direccionament indirecte
inx b
ldax b
mov l,a ; necessari per casos posteriors
inx b
add e
sui 30h
inx b
stax b
pop b
ret
```

Amb aquest nombre hem resolt el cas més fàcil: una xifra, amb resultat d'una xifra, i sense tenir en compte ni el signe ni l'overflow. Si el resultat és de dos xifres, és a dir, si hi ha overflow, i ho veurem amb el bit de carry. Reformularem llegurament el `trigger` perquè estigui més compartimentat, tot i que emprem un altre `sui`, la qual cosa ens augmentarà la complexitat (com el programa que estem fent és de gran escala, influenciarà poc).

La clau amb el *carry* (que no és un *carry* com a tal, perquè es dona a la setzena unitat en el cas d'hexa-

decimal) és separar el nombre en dos, escriure les desenes i després les unitats. Tot això està marcat per la sintonia del canvi de base (d'hexadecimal a decimal).

Observació 4. *Notem que hem fet ús d'una rutina* `clear`*, que ens permeti netejar la pantalla quan pitgem el caràcter* `c` *per teclat.*

```
trigger:
call read
call operate
ret
read:
lxi b,e000h ; direccionament immediat
ldax b
sui 30h ; restem 30, passem ASCII a hex
mov e,a ; usem DE
inx b
inx b ; ignorem el signe
ldax b
sui 30h ; restem 30, passem ASCII a hex
mov l,a ; usem HL
inx b
ret
operate:
inx b ; ignorem el signe igualtat
mov a,e ; usem DE
add l ; acumulador emmagatzema la suma
cpi 0Ah
jp carry ; al ser superior que 0Ah es tracta a part
adi 30h ; direccionament immediat
stax b ; direccionament indirecte
inx b
ret
carry:
mov e,a ; direccionament directe
mvi a,31h
stax b
inx b
mov a,e ; direccionament registres
sui 0Ah
ret
```

Finalment, cal tenir en compte quan els operands tenen més d'una xifra. Notem que amb el primer codi teníem una cosa semblant, ja que estàvem sumant els codis `ASCII` (`30h`, `31h`...) i no pas els nombres en sí. El que canviarà serà com llegim els nombres. *Ho farem en el programa final, on ho ajuntarem tot.* ◁

2.2.2 Segona part

Exercici 5. *Dissenyeu una subrutina que a partir de dos nombres (base 10) introduïts pel teclat del simulador i8085 faci la resta i presenti el resultat en la pantalla de text del i8085. Feu servir els adreçaments directe i indirecte i indiqueu al codi on tenim aquests adreçaments. Com gestioneu el problema del signe? I el problema del carry?*

Demostració. Algunes notes prèvies. Aprofitarem el codi de l'exercici anterior per validar únicament certs caràcters, i ja s'han recollit justament els que necessitem. Mirem si el resultat de fer la resta ens fa saltar el bit de signe. Si és així, imprimim un signe negatiu al davant de la operació i després invertim el resultat.

```
trigger:
call read
call operate
ret
read:
lxi b,e000h
ldax b
sui 30h ; restem 30, passem ASCII a hex
mov e,a ; direccionament directe, usem DE
inx b ; direccionament indirecte
inx b ; ignorem el signe
ldax b
sui 30h ; restem 30, passem ASCII a hex
mov l,a ; usem HL
inx b
ret
operate:
inx b ; ignorem el signe igualtat
mov a,e ; direccionament directe, usem DE
sub l ; acumulador emmagatzema la resta
jm carry_resta ; al ser negatiu es tracta a part
adi 30h ; direccionament immediat
stax b
inx b
ret
carry_resta:
mov e,a
mvi a,2Dh
stax b ; direccionament indirecte
inx b
mov a,e ; direccionament registres
cma ; complementem acumulador
adi 01h ; afegim 1 i ja hem fet el complement a 2
ret
```

Per invertir el nombre es pot optar per restar o i el nombre en qüestió. Aquí s'ha decidit treballar amb el

complement a 2; és el que hem fet durant tot aquesta assignatura. ◁

2.2.3 Tercera part

Exercici 6. *Dissenyeu una subrutina que a partir de dos nombres (base 10) introduïts pel teclat del simulador i8085 faci l'*`AND` *i presenti el resultat en la pantalla de text del i8085.*

Demostració. Les operacions lògiques, segurament, seran les més fàcils de programar. Ja sabem que `1&1=1`, `1&0=0`, `0&1=0` i `0&0=0` (en altres paraules, quan fem operacions amb un sol dígit mai no ens pot donar un *carry*). El codi, a continuació:

```
operate:
inx b ; ignorem el signe igualtat
mov a,e ; usem DE
ana l ; acumulador emmagatzema la and
adi 30h ; direccionament immediat
stax b ; direccionament indirecte
inx b
ret
```

◁

Exercici 7. *Dissenyeu una subrutina que a partir de dos nombres (base 10) introduïts pel teclat del simulador i8085 faci l'*`OR` *i presenti el resultat en la pantalla de text del i8085.*

Demostració. Per la `OR` sí hem de tenir en compte un carry però l'algorisme és gairebé idèntic al de la suma, només cal que canviem les operacions aritmètiques per aquesta lògica. Insistim, tot i que els resultats són diferents, el funcionament és prou similar.

```
operate:
inx b ; ignorem el signe igualtat
mov a,e ; usem DE
ora l ; acumulador emmagatzema la resta
cpi 0Ah
jp carry_or ; al ser positiu es tracta a part
adi 30h
stax b
inx b
ret
carry_or:
mov e,a
mvi a,31h
stax b
inx b
mov a,e
sui 0Ah
ret
```

◁

2.2.4 Codi sencer

Exercici 8. *A partir dels codis generats en els apartats 1 i 2, feu un programa capaç de fer sumes, restes, AND's i OR's.*

Inicialment, l'estructura plantejada per a resoldre aquest problema és la següent:

1. Una subrutina `getter`, que ens permeti obtenir el primer i el segon nombre, independentment del nombre de dígits que tingui.
2. Un `trigger`, és a dir, un caràcter entrat per teclat que provoqui l'execució de l'operació en qüestió.
3. Un `operation_select` ens identificarà el tipus d'operació desitjada en funció del caràcter que hem introduït (en particular +, −, & o |).
4. `operation` ens executarà l'operació en qüestió, amb les particularitats que aquesta pugui tenir.

Com podem veure en programes anteriors, no hem inclòs .

En totes les seccions anteriors hem treballat amb nombres d'una xifra. En cas que es trobi un programa per tractar nombres de més, l'adjuntarem en una secció especial. El codi final per nombres d'una sola xifra, per tant, ha sigut el següent:

```
.define
allowed_count 15
.data 00h
allowed: db 30h,31h,32h,33h,34h,35h,36h,37h,38h,39h,
            2Bh,2Dh,26h,7Ch,3Dh
.org 100h
pila:
.org 200h
; Programa Principal
lxi h,pila
sphl
lxi b, E000h
bucle:
jmp bucle
.org 0024h
call string_in
call operation
ret
.org 300h
; ---------------------------------
string_in:
in 00h
cpi 00h
jz no_tecla
tecla:
```

```
call check_allowed
cpi 00h
jz no_tecla
cpi 43h
jz clear
stax b
inx b
no_tecla:
ret
check_allowed:
push d
push h
mvi e, allowed_count
lxi h, allowed
allowed_loop:
mov d,m
cmp d
jz is_allowed
inx h
dcr e
jnz allowed_loop
jmp not_allowed
is_allowed:
mov a,d
jmp end_allowed
not_allowed:
mvi a,00h
end_allowed:
pop h
pop d
ret
clear:
lxi h,0000h
mov l,c
clear_loop:
mvi a,00h
stax b
dcx b
dcx h
add l ; activem bit zero si cal
jnz clear_loop
stax b
ret
; -------------------------------------
operation:
cpi 3Dh
jz trigger
ret
```

```
trigger:
call read
call operate
ret
read:
dcx b
dcx b
ldax b ; accedim al segon nombre
sui 30h ; restem 30, passem ASCII a hex
mov e,a ; usem DE
dcx b
dcx b ; mirem el signe posteriorment
ldax b ; carreguem el primer nombre
sui 30h ; restem 30, passem ASCII a hex
mov l,a ; usem HL
inx b ; accedim al signe
ldax b
cpi 2Bh ; si es fa una suma se salta a la suma
jz suma
cpi 2Dh ; si es fa una resta se salta a la resta
jz resta
cpi 41h
jz and ; si es fa un and se salta al and
cpi 4Fh ; si es fa un or se salta a or
jz or
; posem b al final de la op per si es demana un clear
lxi b,e0003h
ret
operate:
suma:
inx b ; ignorem el signe igualtat
mov a,e ; usem DE
add l ; acumulador emmagatzema la suma
cpi 0Ah
jp carry ; al ser superior que 0Ah es tracta a part
adi 30h ; direccionament immediat
stax b ; direccionament indirecte
inx b
ret
carry:
mov e,a ; direccionament directe
mvi a,31h
stax b
inx b
mov a,e ; direccionament registres
sui 0Ah
ret
resta:
```

```
inx b ; ignorem el signe igualtat
 ; direccionament directe, usem DE
sub l ; acumulador emmagatzema la resta
jm carry_resta ; al ser negatiu es tracta a part
adi 30h ; direccionament immediat
stax b
inx b
ret
carry_resta:
mov e,a
mvi a,2Dh
stax b ; direccionament indirecte
inx b
mov a,e ; direccionament registres
cma ; complementem acumulador
adi 01h ; afegim 1 i ja hem fet el complement a 2
ret
or:
inx b ; ignorem el signe igualtat
mov a,e ; usem DE
ana l ; acumulador emmagatzema la and
adi 30h ; direccionament immediat
stax b ; direccionament indirecte
inx b
ret
```

Hem trobat una dificultat bastant gran amb els `ret`; els diversos mètodes que anàvem implementant per al `carry` en la suma, la resta i la comparació `AND` no es corresponien correctament amb els valors que hi havia a la pila. Detallant-ho més, a l'acabar la subrutina ens enviava a una posició de la memòria d'instruccions que no tenia cap mena de sentit. S'ha corregit aquest error posant l'adreça dedicada a la interrupció `TRAP` corresponent amb el codi al final.

A més, aquestes subrutines no emmagatzemaven el que havien d'emmagatzemar a memòria, així que s'han reestructurat per adaptar-se a les directrius establertes:

```
.define
allowed_count 16
.data 00h
allowed: db 2Bh,2Dh,26h,30h,31h,32h,33h,34h,35h,36h,37h,38h,39h
   ,3Dh,43h,7Ch
.org 100h
pila:
.org 200h
; Programa Principal
lxi h,pila
sphl
lxi b, E000h
bucle:
```

```
jmp bucle
.org 300h
; ---------------------------------
string_in:
in 00h
cpi 00h
jz no_tecla
tecla:
call check_allowed
cpi 00h
jz no_tecla
cpi 43h
jz clear
stax b
inx b
no_tecla:
ret
check_allowed:
push d
push h
mvi e, allowed_count
lxi h, allowed
allowed_loop:
mov d,m
cmp d
jz is_allowed
inx h
dcr e
jnz allowed_loop
jmp not_allowed
is_allowed:
mov a,d
jmp end_allowed
not_allowed:
mvi a,00h
end_allowed:
pop h
pop d
ret
clear:
push psw
lxi h,0000h
mov l,c
clear_loop:
mvi a,00h
stax b
dcx b
dcx h
```

```
add l ; activem bit zero si cal
jnz clear_loop
stax b
pop psw
ret
; ------------------------------------
operation:
cpi 3Dh
jz trigger
ret
trigger:
call read_and_op
ret
read_and_op:
push b
dcx b
dcx b
ldax b ; accedim al segon nombre
sui 30h ; restem 30, passem ASCII a hex
mov l,a ; usem HL per emmagatzemar
dcx b
dcx b ; ignorem el signe de moment
ldax b ; carreguem el primer nombre
sui 30h ; restem 30, passem ASCII a hex
mov e,a ; usem DE per emmagatzemar
inx b ; accedim al signe
ldax b
pop b ; retornem a la posicio final
cpi 2Bh ; si es fa una suma se salta a la suma
jz suma
cpi 2Dh ; si es fa una resta se salta a la resta
jz resta
cpi 26h
jz and ; si es fa un and se salta al and
cpi 7Ch ; si es fa un or se salta a or
jz or
ret
.org 600h
suma:
mov a,e ; usem DE
add l ; acumulador emmagatzema la suma
cpi 0Ah
jp carry ; al ser superior que 0Ah es tracta a part
adi 30h ; direccionament immediat
stax b ; direccionament indirecte
ret
carry:
mov e,a ; direccionament directe
```

```
mvi a,31h
stax b
inx b
mov a,e ; direccionament registre
sui 0Ah
adi 30h ; direccionament immediat
stax b ; direccionament indirecte
ret
resta:
mov a,e ; direccionament directe, usem DE
sub l ; acumulador emmagatzema la resta
jm carry_resta ; al ser negatiu es tracta a part
adi 30h ; direccionament immediat
stax b
ret
carry_resta:
mov e,a
mvi a,2Dh
stax b ; direccionament indirecte
inx b
mov a,e ; direccionament registre
cma ; complementem acumulador
adi 01h ; afegim 1 i ja hem fet el complement a 2
adi 30h ; direccionament immediat
stax b
ret
and:
mov a,e ; usem DE
ana l ; acumulador emmagatzema la and
adi 30h ; direccionament immediat
stax b ; direccionament indirecte
ret
or:
mov a,e ; usem DE
ora l ; acumulador emmagatzema la resta
cpi 0Ah
jp carry_or ; al ser positiu es tracta a part
adi 30h
stax b
ret
carry_or:
mov e,a
mvi a,31h
stax b
inx b
mov a,e
sui 0Ah
adi 30h
```

```
stax b
ret
.org 0024h
call string_in
call operation
ret
```

2.3 Epíleg

Exercici 9.

1. *Quina diferència hi ha entre la suma i la* `OR`*?*
   - *són iguals*
   - *la* `OR` *és una operació lògica i la suma és una operació aritmètica*
   - *la* `OR` *és una operació aritmètica i la suma és una operació lògica*
   - *cap de les anteriors és correcta.*
2. *La instrucció* `STA 1234h`
   - *és una operació que carrega el contingut de la posició de memòria* `1234h` *en l'acumulador*
   - *fa servir adreçament directe*
   - *fa servir adreçament immediat*
   - *totes són certes*

Demostració.

1. Tot i que tenen un comportament molt semblant, no són iguals. Per exemple, al fer `8 OR 8 = 8` però, en canvi, 8 + 8 = 16. Podem classificar `or` com una operació lògica (com`AND`, `NOR` o `XOR`) i l'operació de la suma és per conveni una operació aritmètica (suma, resta, multiplicació, divisió...).
2. La instrucció `STA 1234h` guarda una còpia del contingut actual de l'acumulador a la posició de memòria 1234h. Fa servir adreçament directe, ja que l'adreça es proporciona directament. No hi ha cap tipus d'adreçament immediat, ja que no apareix cap immediat a la operació. I, evidentment, la última no pot ser cerca si ja fallen anteriors.

◁