Introducció als Ordinadors

Pràctica 6

Mario Vilar

24 de maig de 2022

Índex

1 Introducció

El i8085 és un processador de 8 bits que consta d'un format d'adreçament de 16 bits. Per tal de fer això, el que fa és agrupar els registres en parelles.

El 8085 és un disseny de von Neumann convencional basat en l'Intel 8080. A diferència del 8080, no multiplexa els senyals d'estat al bus de dades, però el bus de dades de 8 bits es multiplexa amb els vuit bits inferiors del bus d'adreces de 16 bits per limitar el nombre de pins a 40. Els senyals d'estat es proporcionen mitjançant pins de senyal de control de bus dedicats i dos pins d'identificació d'estat de bus dedicats anomenats So i S1. El pin 39 s'utilitza com a pin de retenció. El processador es va dissenyar utilitzant circuits nMOS, i les versions posteriors "H"es van implementar en el procés nMOS millorat d'Intel anomenat HMOS II ("MOS d'alt rendiment"), desenvolupat originalment per a productes de RAM estàtica ràpida. Només es necessita una única font d'alimentació de 5 volts, com els processadors de la competència i a diferència del 8080. El 8085 utilitza aproximadament 6500 transistors.

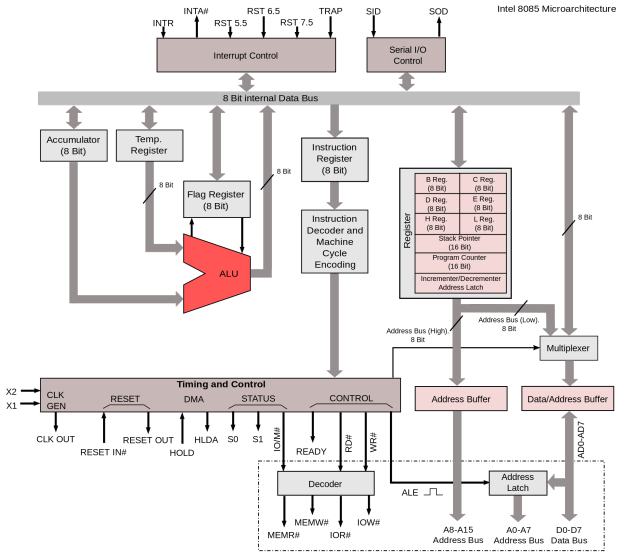

Figura 1: Esquema del processador 8085.

1.1 Objectius

Conèixer els modes d'adreçament del 8085 i familiaritzar-se amb les seves instruccions. Per altra banda, un altre objectiu és solidificar els coneixements adquirits per part de l'alumne en les pràctiques anteriors.

1. Estudi de l'espai de memòries del microprocessador.
2. Comprendre el funcionament de la pila.
3. Veure un exemple d'utilització de subrutines.
4. Comprendre l'adreçament a ports E/S.

2 Cos del treball

2.1 Pràctica

2.1.1 Part 1

Executeu pas a pas el següent programa:

```
.define
    num 02h
.data 00h
    mat1: db 1,2
    mat2: db 3,4
    mat3: db 0,0
.data 20h
    pila:
.org 600h
    lxi h, pila
    sphl
    mvi b, num
    lxi d, mat1
    lxi h, mat2
loop:
    call suma
    dcr b
    jnz loop
    nop
    hlt
suma:
    push psw
    ldax d
    add m
    stax d
    inx h
    inx d
    pop psw
    ret
```

```
.data
    mat1: .byte 1, 2
    mat2: .byte 3, 4
    mat3: .byte 0, 0
.text
    addi s0,2
    la a1, mat1
    la a2, mat2
loop:
    jal suma
    addi s0,s0,-1
    bgtz s0,loop
    j end
suma:
    addi sp,sp,-12
    sw s1,12(sp)
    sw s2,8(sp)
    sw s3,4(sp)
    lb s1,0(a1)
    lb s2,0(a2)
    add s3,s1,s2
    sb s3,0(a1)
    addi a1,a1,1
    addi a2,a2,1
    lw s2,4(sp)
    lw s2,8(sp)
    lw s1,12(sp)
    addi sp,sp,-12
    ret
end:
    nop
```

Demostració. El programa a executar el trobem a `600h` (el PC apuntarà inicialment a 600h). Deixem una distància prudencial entre les dades que definim i les instruccions del programa. Sigui com sigui, primerament carreguem la posició de l'element inicial de la `pila` (40h) a HL i guardem aquest mateix registre a l'stack pointer. A continuació, carreguem `02h` al registre B i tornem a carregar la posició de `mat1`, ara a DE, i `mat2` a HL.

Acte seguit, s'entra al `loop` i es crida a la subrutina `suma`. D'aquesta manera, el `PC` apuntarà a la primera instrucció de la subrutina i `SP`, a `001E` (`001F` conté `06h` i `001E`, `0Fh`, i `060F` és la instrucció posterior a la crida de `suma`). Es fa un push de *Primary Search Word* (disminuïm `SP` en dos unitats i emmagatzemem `PSW`). Carreguem `DE` a l'acumulador i li afegim, a la següent instrucció, el contingut de memòria apuntat pels registres `HL` i tornem a guardar a acumulador. El guardem a `DE` i incrementem `HL` i `DE` en una unitat. Fem un `pop` de `psw` i tornem a la instrucció `dcr b`, situada després de la crida de la subrutina. Decrementem el registre `b` i tornem al `loop`, que s'executarà solament un parell de vegades més: al principi hi hem emmagatzemat un `02h`, de manera que hi haurà tres iteracions.

Pel que fa al programa en Ripes, en principi la funcionalitat és la mateixa. Notar un avantatge del paradigma `RISC-V`: mitjançant l'adreçament indexat, en aquest cas aplicat a `sp`, podem accedir a successives posicions de memòria simplement modificant l'*offset*, mentre que a `8085` aquesta funcionalitat no hi és. ◁

Exercici 1. *L'adreçament de la instrucció* `lxi` *és: directe, indirecte, immediat, implícit? Quina instrucció guarda el* `PC` *a la pila?* `push pc`*,* `pop pc`*,* `call`*,* `mov m,pc`*. Quin espai ocupa en memòria la subrutina* `suma`*? Quants cicles triga en executar-se la subrutina* `suma`*?*

1. L'adreçament de la instrucció `lxi` és immediat.
2. La instrucció que guarda el `PC` a la pila és el `call PC`. `call` desa el contingut del comptador de programa (l'adreça de la propera instrucció seqüencial) dins del *stack* i tot seguit salta a l'adreça especificada per `label`.
3. Per calcular l'espai que ocupa en memòria la subrutina `suma` hem de veure el simulador. Simplement, comprovem a quina posició comença i a quina posició acaba i efectuem una resta. En efecte: ocupa 8 bytes de memòria, un per cada instrucció.
4. Per calcular els cicles que triga a executar-se la subrutina `suma` ens cal analitzar una per una les instruccions que es van succeint i sumem el nombre de cicles de rellotge que necessita cadascuna en la seva execució.

Instrucció	Cicles
`push psw`	12
`ldax b`	7
`add m`	7
`stax d`	7
`inx h`	6
`inx d`	6
`pop psw`	10
`ret`	10
Total	65

Figura 2: Instruccions i els cicles associats.

En efecte, `suma` triga 65 cicles a executar-se.

Exercici 2 (Estudi de l'espai de memòries del microprocessador)**.** *Dibuixeu el mapa de memòria de dades: direccions i contingut. Indiqueu les instruccions que modifiquen les dades de la memòria. En cadascuna d'elles, indiqueu quines modificacions es produeixen. Situeu la memòria de programa. Dins d'ella, localitza el sub-bloc que pertany a la subrutina 'suma'.*

Demostració. Per respondre acuradament a la pregunta hem de diferenciar entre la memòria de dades i la memòria de programa o d'instruccions. També aplicaria la memòria de l'*stack*, la qual no comentarem perquè resulta irrellevant (no dins del programa, perquè s'acaba utilitzant, però sí en l'explicació):

Direcció de memòria	Contingut	Sub-bloc
0600h	`lxi h, pila`	∅
0603h	`sphl`	∅
0604h	`mvi b, num`	∅
0606h	`lxi d, mat1`	∅
0609h	`lxi h, mat2`	∅
060Dh	`call suma`	∅
060Fh	`dcr b`	∅
0610h	`jnz loop`	∅
0613h	`nop`	∅
0614h	`hlt`	∅
0615h	`push psw`	`suma`
0616h	`ldax d`	`suma`
0617h	`add m`	`suma`
0618h	`stax d`	`suma`
0619h	`inx h`	`suma`
061Ah	`inx d`	`suma`
061Bh	`pop psw`	`suma`
061Ch	`ret`	`suma`

Figura 3: Dibuix del mapa de memòria d'instruccions, direccions i contingut.

Direcció	Contingut
00h	1
01h	2
02h	3
03h	4
20h	`pila` (0)

Figura 4: Mapa de la memòria de dades

Els espais buits que hi ha entre les instruccions, en la memòria d'instruccions, corresponen al nombre de bits que ocupa o retorna aquesta operació. ◁

Exercici 3 (Funcionament de la pila)**.** *Indiqueu el començament de la pila en l'espai de memòria de dades del microprocessador. Indiqueu quines instruccions modifiquen la pila.*

Demostració. L'*stack pointer* és un registre de 16 bits que emmagatzema l'adreça superior de l'*stack memory*. Farem una taula on indicarem la instrucció, una breu descripció i el canvi que provoca en SP:

Instrucció	*Descripció*	*Canvi en la pila*
`PUSH RP`	Afegeix RP a la pila. PUSH copia dos bytes de dades a l'*stack*. La dada pot ser el contingut d'un parell de registres o la "paraula d'estat del programa"(PSW)	El registre SP disminueix en 2 bytes i un nou element s'insereix a la pila.
`POP RP`	Treu RP de la pila. `POP RP` copia el contingut de la posició de memòria adreçada per l'*stack pointer* al registre de sota ordre del parell de registres especificats	El registre SP augmenta en 2 bytes i l'ítem superior serà eliminat de l'*stack*.
`CALL label`	`CALL` desa el contingut del comptador de programa (l'adreça de la propera instrucció seqüencial) dins de l'*stack* i tot seguit salta a l'adreça especificada per `LABEL`.	SP passa a apuntar l'ítem superior de la memòria de l'*stack*.
`SPHL`	`SPHL` és una instrucció amb l'ajuda de la qual el punter de pila s'inicialitzarà amb el contingut del parell de registres `HL`.	És una manera indirecta d'inicialitzar SP i, per tant, un potencial canvi en la pila.

Cal dir que `POP` i `PUSH` tenen diferents `op-codes` en funció de quin operand es doni. I amb aquesta taula, en principi, en tenim prou. ◁

2.1.2 Part 2

Executeu pas a pas el següent programa:

```
.data 100h
pila:
.org 24h
jmp ports
.org 500h
lxi h,pila
sphl
call ports
nop
hlt
ports:
push psw
in 04h
ani 00000001
out 05h
pop psw
ret
```

Exercici 4. *Què fa la subrutina* `ports`*? Per això, introduïu dades amb els interruptors o amb el teclat; observeu en un port de sortida el resultat de la subrutina.*

Demostració. A primera vista, `ports` s'utilitza quan s'activa una de les interrupcions, mirant el port `04h` i projectant el resultat en `05h`: guarda l'entrada per teclat a l'acumulador i, d'aquesta manera, fa l'operació lògica `and` amb `00000001` (`01h`). Després, per organització de la memòria d'instruccions del programa, s'executaria tot el codi que hem guardat a `500h`. De fet, fixem-nos que la subrutina `ports` es torna a executar i, a continuació, finalitza amb `hlt`.

Així doncs, aquest programa s'encarrega de comprovar la paritat del nombre que introduïm per teclat: l'`ani`, l'`and` amb immediat, s'encarrega de comprovar l'últim bit del nombre per veure si aquest és 1 o és 0. En el cas que sigui 1, és a dir, que sigui imparell, acumularem un 1. En cas que sigui 0, és a dir, que sigui parell, acumularem un 0. En qualsevol cas, el contingut d'aquest acumulador serà canalitzat cap al port 05h de sortida. Si som suficientment hàbils, podríem projectar aquesta sortida per pantalla, sempre i quan tinguéssim en compte la codificació del nombre per la *display*. ◁

2.1.3 Part 3

Exercici 5. *Dissenyeu un programa en assemblador que representi en un* display *de 7 segments els números del 0 al 5. Els números s'introduiran a partir del teclat. A més, ha de permetre l'opció d'esborrar el* display*; per això, en prémer la lletra* `c`*, es produirà un* `clear` *del* display.

Demostració. En aquest programa hem de tenir en consideració diversos punts. En primer lloc, la codificació dels nombres en assemblador: el 0 seria el 30, l'1 el 31 i així successivament. `c` seria 43. De totes maneres, aquest funcionament natural s'altera en el *display* de 7 segments; l'hem d'entendre d'una altra manera.

Per poder mostrar un número al *display* 7 segments, el que hem de fer és il·luminar aquells segments per formar aquest número. *Per exemple, si volem mostrar el 4, cal activar els segments* f, g, b, c *i desactivar la resta de segments.* Això és perquè un *display* 7 segments és un component que té 7 segements LEDs (és a dir, 7 bits), més un LED que farà de punt (un bit, el més significatiu, que serà zero).

Nombre	Codificació
0	`01110111b`
1	`01000100b`
2	`00111110b`
3	`01101101b`
4	`01001101b`
5	`01101011b`

Format: ocdegbaf

Figura 5: Relació entre els nombres del 0 al 5 i la seva codificació.

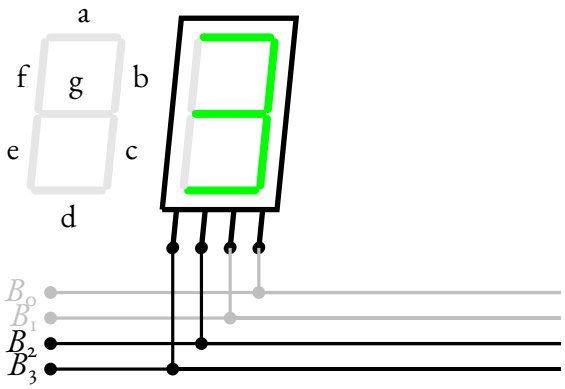

Figura 6: Representació d'un *display* de set segments.

Com veiem, no es pot trobar una funció que ens expressi directament una correspondència entre els valors que necessitem per representar-los al *display* de 7 segments i els valors que prenen al ser introduïts per teclat.

Amb tot això, optarem per declarar una sèrie de dades que emmagatzemin els nombres que ens permeten fer la impressió per pantalla. Intentarem jugar amb l'acumulador perquè s'accedeixi a la posició de memòria on tenim els valors guardats.

Fem el programa:

```
.data 30h
zero: db 01110111b
un: db 01000100b
dos: db 00111110b
tres: db 01101110b
quatre: db 01001101b
cinc: db 01101011b
.data 43h
clear: db 00000000b
.org 600h
loop: ; bucle infinit
    jmp loop
hlt
interrupt:
    in 00h
    mov c,a
    ldax b
    out 07h
    ret
.org 24h
call interrupt
ret
```

Com que en 8085 el programa s'executa seqüencialment, comcençarem per `600h` i, per tant, un bucle infinit. Quan es produeix un esdeveniment sortim del bucle i, en particular, quan es dona una excepció `trap`. Aquesta haurà estat produïda per una pulsació del teclat. Saltarem, concretament, a la posició `24h` de la memòria d'instruccions i cridarem a la subrutina `interrupt`. `interrupt` s'encarrega de:

1. captarem l'entrada pel port `00h` i la carregarem a l'acumulador, un valor entre `30h` i `46h`;
2. notem que ara emmagatzemem un valor que es troba justament en l'interval en què estan compreses les dades que conservem, la correspondència per poder imprimir els valors per pantalla;
3. movem els dos bytes d'`A` al registre `C` per poder carregar amb `ldax` el contingut de la posició de memòria apuntat per `BC`, el nombre a mostrar, a l'acumulador;
4. projectar l'acumulador pel port de sortida, en aquest cas, `07h`;
5. retornar a la instrucció on s'havia fet la crida (`PC<=`), `call interrupt`, la qual, al seu torn, tor-

narà al `loop` infinit (PC<=).

Després de la crida a `interrupt`, es retorna al bucle, a l'espera d'una nova instrucció.

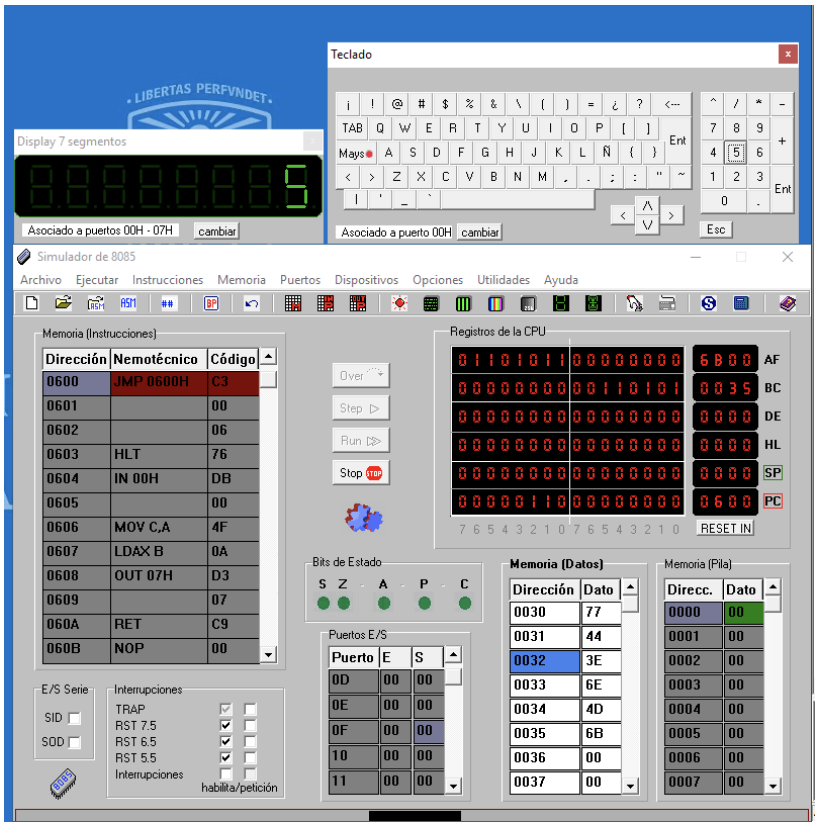

Figura 7: Captura de pantalla de l'execució del programa.

◁

Observació 6. *Inicialment havíem utilitzat el següent programa:*

```
.data 100h
zero: db 01110111b
un: db 01000100b
dos: db 00111110b
tres: db 01101110b
quatre: db 01001101b
cinc: db 01101011b
.data 113h
clear: db 00000000b
.org 600h
loop: ; bucle infinit
    jmp loop
hlt
interrupt:
    in 00h
    adi 70h
```

```
    mov c,a
    ldax b
    out 07h
    ret
.org 24h
call interrupt
ret
```

La única diferència són les posicions on guardem les dades. Aquí, intentem respectar les primeres posicions de la memòria de dades per si estan reservades a valors especials, per no obtenir errors. És un pas una mica inútil, ja que al simulador ens trobem el següent, abans d'escriure qualsevol programa:

Direcció	Dades
0030	00
0031	00
0032	00
0033	00
0034	00

Figura 8: Memòria de dades, primeres posicions.

Direcció	Instrucció	Codi
0000	NOP	00
0001	NOP	00
0002	NOP	00
0003	NOP	00
0004	NOP	00

Figura 9: Memòria d'instruccions.

Posant les dades a `100h`*, ens veuríem forçats a fer un* `adi` *per accedir-hi, augmentant així el nombre de cicles de rellotge i, per tant, el temps d'execució del programa. No ens compensa.*

Exercici 7. *El simulador 8085 té una part de la memòria de dades que implementa una 'Pantalla de text'. Modifiqueu el programa per escriure la lletra que vulgueu a la pantalla.*

Demostració. El programa que utilitzarem, que simplement carrega els caràcters a pantalla (no fa el *clear*), és el següent:

```
.org 100h
lxi h, e000h
loop:
    jmp loop
hlt
rai:
    in 00h
    mov M,A
    inx H
    ret
.org 24h
    call rai
    ret
```

Anem a fer-ne un anàlisi:

1. carreguem `HL` amb el port de la pantalla, `E000h`;

2. entrem al bucle infinit, a l'espera d'una entrada per teclat;
3. saltem a 24h pel `trap`, guardant 0106h a l'SP, i cridem a la subrutina;
4. es guarda automàticament 0027h a l'SP per recordar on hem de retornar;
5. es llegeix l'entrada de teclat i es guarda en acumulador;
6. es guarda A en la posició de memòria direccionada pel contingut dels registres HL, justament on tenim els valors de la pantalla;
7. s'incrementa H per introduir el següent valor;
8. es retorna a 0027h i, finalment, a 0106h, el bucle infinit.

Podem aplicar aquest procediment per al nombre de passos que convingui. Ara un programa més refinat, que té en compte el punter de pila i utilitza el parell PC per apuntar a la memòria de text, per altra banda també té una subrutina d'introducció de dades per consola (bastant semblant a l'anterior, tot i que aquesta té en compte quan s'introdueix un caràcter i quan no).

```
.org 100h
pila:
.org 200h
; Programa Principal
lxi H, pila
sphl
mvi B,E0h
mvi C,00h
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:
stax B
inx B
no_tecla:
ret
```

Finalment, a tall d'observació, adjuntem un codi adaptat que projecta el resultat per pantalla i en fa un *clear* quan entrem la lletra c:

```
.org 100h
lxi d, e000h
loop:
```

```
    jmp loop
hlt
rai:
    in 00h
    stax d
    inx d
    inx b
    cpi 43h
    jz clear
    mov l,a
    mov a,m
    ret
.org 24h
    call rai
    ret
.org 600h
clear:
    mvi a,00h
    stax d
    dcx d
    dcx b
    add c
    jnz clear
stax d
jmp loop
```

La idea utilitzada és la mateixa, però necessitem més registres perquè volem tenir controlat el nombre de posicions de memòria que hem usat per escriure per pantalla; en algun moment voldrem fer el *clear*. El *clear*, com veiem, consisteix en fer un bucle descendent esborrant tot registre fins a deixar la pantalla buida un altre cop. ◁

2.2 Teorico-pràctica

Aquesta setmana no en tenim.

3 Conclusions

En aquesta pràctica hem realitzat tres petits programes, que, de fet, ens els ha lliurat el professor i nosaltres els hem entès per tal de:

1. Familiaritzar-se amb les instruccions del i8085.
2. Entendre el procediment de programació.
3. Començar a veure l'entorn de programació.
4. Entendre els funcionaments dels punters de memòria.

5. Estudi de l'espai de memòries del microprocessador.
6. Comprendre el funcionament de la pila.
7. Veure un exemple d'utilització de subrutines.
8. Comprendre l'adreçament a ports E/S.