UNIVERSITAT DE BARCELONA

Pràctica 5: Introducció al simulador i8085

Alejandro Guzman

19 Maig 2022

ÍNDEX

1 OBJECTIUS

Aquesta és la primera pràctica que realitzarem amb el simulador i8085.

L'objectiu principal de la pràctica és, de la mateixa manera que la primera pràctica del simulador Ripes, serà aprendre a utilitzar i analitzar tot l'entorn relacionat amb aquest nou simulador.

Per això, realitzarem dos exercicis amb els quals aprendrem alguns modes d'adreçament del simulador i a com utilitzar subrutines, a part de les diferències bàsiques entre el simulador i8085 i el RIPES utilitzat en les anteriors pràctiques o, en general, amb l'arquitectura RISC-V (les instruccions no tenen totes la mateixa mida, les adreces són de 16 bits, no tenim 32 registres de propòsit general, només 6 i, a més, són només de 8 bits, i no de 32 com estàvem acostumats...).

2 QÜESTIONS PLANTEJADES AL GUIÓ DE PRÀCTIQUES

1 Exercici 1

El primer exercici consisteix en realitzar un petit programa amb el simulador i8085, el qual consisteix en sumar dues matrius.

Se'ns demana que implementem el codi de dues maneres diferents, la primera és amb l'ajuda d'una tercera matriu per tal de desar el resultat de les sumes en aquesta i la segona és directament desar el resultat en la segona matriu, sense cap matriu auxiliar.

A contrinuació es mostren les dues implementacions del programa demanat:
Primera implementació

```
.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
    LXI B, mat3
    MVI D, num
carga:
    MOV A,M
    STAX B
    INX H
    INX B
    DCR D
    JNZ carga
    LXI B,mat3
    MVI D,num
loop:
    LDAX B
    ADD M
    STAX B
    INX H
    INX B
    DCR D
    JNZ loop
HLT
```

Segona implementació

```
.define
num 5
.data 00h
    mat1: db 1,2,3,4,5
    mat2: db 6,7,8,9,0
.org 100h
    LXI H, mat1
    LXI B, mat2
    MVI D, num
loop:
    LDAX B
    ADD M
    STAX B
    INX H
    INX B
    DCR D
```

```
17      JNZ  loop
18  HLT
```

En el primer codi, realitzem un bucle per copiar mat1 a mat3 i un segon per sumar mat3 a mat2, sobreescrivint el resultat a mat3.

En canvi en la segona implementació directament realitzem un sol bucle per sumar mat1 a mat2 i sobreescriure el resultat a mat2, d'aquesta manera ens estalviem temps i nombre d'instruccions, tot i que perdem els valors inicials de mat2.

Ara respondrem les qüestions que se'ns plantejen en el guió de pràctiques relacionades amb les implementacions del programa realitzades.

1.1 Pregunta 1

En què simplificaria molt el codi del programa un dels modes d'adreçament del simulador Ripes?
L'adreçament relatiu del simulador Ripes seria molt útil per aquest programa, simplificant-lo substancialment ja que podríem guardar a un registre (per exemple a0) la direcció de memòria mat1 i, aleshores, com cada posició de memòria té espai per un byte, sabem que mat2 és mat1 + 5, i mat3 és mat1 + 10. Usant offsets adequats podríem recórrer totes les matrius alhora, executant el programa amb només 5 iteracions i sense necessitat de carregar les adreces de totes les matrius a registres diferents.

En la segona implementació ens hem apropat a aquesta simplificació utilitzant només dos registres, però gràcies a l'adreçament relatiu del simulador Ripes podríem inclús fer servir un únic registre per sumar dues matrius.

Calculeu les mides del codi del vostre programa i el nombre de cicles per a la seva execució. Per a calcular el nombre de cicles de rellotge no tindrem en compte el que es troba escrit després de les directives *define* ni *data*, ja que no formen part propiament del programa. A continuació veiem dues taules amb les instruccions emprades, quina mida tenen, en quants cicles s'executen, quants cops apareixen al programa i quants als bucles. Cada taula és per cada implementació realitzada (en total 2).

Instrucció	**Mida (*bytes***	**N de cicles**	**N d'aparicions**	**N a bucles**
LXI RP,DA16	3	10	3	0
MVI reg,DAT	2	7	2	0
MOV reg,M	1	7	1	1
STAX RP	1	7	2	2
INX RP	1	6	4	4
DCR reg	1	4	2	2
JNZ LABEL	3	7/10	2	2
LDAX RP	1	7	1	1
ADD M	1	7	1	1
HLT	1	4	1	0

Table 2.1: Taula de cicles de la primera implementació

La mida del programa és la mida de cada instrucció pel nombre d'aparicions al codi sense tenir en compte les repeticions per estar dintre d'un bucle. Per tant calculem la mida d'ambdues

Instrucció	Mida (*bytes*	N de cicles	N d'aparicions	N a bucles
LXI RP,DA16	3	10	2	0
MVI reg,DAT	2	7	1	0
MOV reg,M	1	7	0	0
STAX RP	1	7	1	1
INX RP	1	6	2	2
DCR reg	1	4	1	1
JNZ LABEL	3	7/10	1	1
LDAX RP	1	7	1	1
ADD M	1	7	1	1
HLT	1	4	1	0

Table 2.2: Taula de cicles de la segona implementació

implementacions:

- 1^a implementació: $3 \cdot 3 + 2 \cdot 2 + 1 \cdot 1 + 1 \cdot 2 + 1 \cdot 4 + 1 \cdot 2 + 3 \cdot 2 + 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 = 31$ *bytes*. En el simulador ho indica de la forma 11Eh => 100h + 30.
- 2^a implementació: $3 \cdot 2 + 2 \cdot 1 + 1 \cdot 0 + 1 \cdot 1 + 1 \cdot 2 + 1 \cdot 1 + 3 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 = 18$ *bytes*. En el simulador ho indica de la forma 111h => 100h + 17.

El nombre de cicles necessaris per l'execució és la suma dels cicles totals de les instruccions realitzades fora dels bucles, dels cicles totals de les instruccions que es troben als bucles que no siguin instruccions de salt (nº cicles per execució · nº aparicions en bucles · nº iteracions de cada bucle) i els cicles totals de les instruccions de salt (nº cicles per execució en cas de salt · nº de vegades que realitza el salt · n^o de bucles + nº cicles per execució en cas de no salt · nº de vegades que no realitza el salt · nº de bucles).

- 1^a implementació: Cicles totals de les instruccions realitzades fora dels bucles + Cicles totals de les instruccions que es troben als bucles que no siguin instruccions de salt + Cicles totals de les instruccions de salt = $(10 \cdot 3 + 7 \cdot 2 + 4 \cdot 1) + 5 \cdot (7 \cdot 1 + 7 \cdot 2 + 6 \cdot 4 + 4 \cdot 2 + 7 \cdot 1 + 7 \cdot 1) + 2 \cdot (10 \cdot 4 + 7 \cdot 1) = 48 + 335 + 94 = 477$ cicles.
- 2^a implementació: Cicles totals de les instruccions realitzades fora dels bucles + Cicles totals de les instruccions que es troben als bucles que no siguin instruccions de salt + Cicles totals de les instruccions de salt = $(10 \cdot 2 + 7 \cdot 1 + 4 \cdot 1) + 5 \cdot (7 \cdot 1 + 6 \cdot 2 + 4 \cdot 1 + 7 \cdot 1 + 7 \cdot 1) + 1 \cdot (10 \cdot 4 + 7 \cdot 1) = 31 + 185 + 47 = 263$ cicles.

1.2 Pregunta 2

Quants cicles de rellotge triga en executar-se una instrucció aritmètic – lògica qualsevol? Feu servir el fitxer adjunt on especifica el ISA del 8085. Indica quina és la mida mitjana de les teves instruccions. Calcula els cicles per instrucció mitjà per aquests codis.
Havent consultat el fitxer adjunt, totes les instruccions aritmètic-lògiques s'executen en 4 cicles si operen amb un registre i 7 si ho fan directament, anant a buscar l'operador a memòria i 7 també si s'opera amb un immediat (1 *byte*).
Per calcular la mida mitjana de les instruccions consultem les taules 2.1 i 2.2 per a la primera i a la segona implementació, respectivament.

- 1^a implementació: La mida total del programa és de 31 *bytes* i consta de 19 instruccions → La mida mitjana de les instruccions és de $31/19 \approx 1.63$.

- 2^a implementació: La mida total del programa és de 18 *bytes* i consta de 11 instruccions → La mida mitjana de les instruccions és de $18/11 \approx 1.64$.

De la mateixa manera per calcular els cicles per instrucció mitjà consultem les taules 2.1 i 2.2 per a la primera i a la segona implementació, respectivament.

- 1^a implementació: El nº de cicles total és de 477 cicles i executem $6 + 5 \cdot (1 + 2 + 4 + 2 + 1 + 1) + 2 \cdot (4 + 1) = 71$ instruccions (repetint les dels bucles, és a dir, tot el programa). Llavors el cicle per instrucció mitjà és $477/71 \approx 6.72$.
- 2^a implementació: El nº de cicles total és de 263 cicles i executem $4 + 5 \cdot (1 + 2 + 1 + 1 + 1) + 1 \cdot (4 + 1) = 39$ instruccions (repetint les dels bucles, és a dir, tot el programa). Llavors el cicle per instrucció mitjà és $263/39 \approx 6.74$.

1.3 Pregunta/Tasca 3

Pugeu el vostre codi i marqueu quina és la instrucció del vostre programa que triga més cicles en executar-se
Les dues instruccions que més triguen a executar-se segons la documentació del simulador són **LXI RP,DA16** i **JNZ LABEL**, on la primera triga 10 cicles en executar-se i la segona triga 7 cicles si no es realitza el salt condicional i 10 cicles si es realitza.

Els codis de les dues implementacions han estat mostrats abans de respondre a les preguntes d'aquest exercici.

2 Pregunta/Tasca 4

Traduiu el codi per fer-lo servir amb el simulador Ripes. Quants cicles triga en executar-se? Compareu els resultats (mida de codi, accessos a memòria i cicles promig per instrucció) amb els valors obtinguts per l'i8085

Traduïm cada implementació per separat en el simulador Ripes.

```
1  .data
2  mat1: .byte 1, 2, 3, 4, 5,
3  mat2: .byte 6, 7, 8, 9, 0,
4  mat3: .byte 0, 0, 0, 0, 0,
5  .text
6      la a0, mat1             # Carreguem a a0 el valor de mat1
7      addi a1, zero, 5        # Assignem el valor del registre a1 a 5
8  loop:
9      lb a2, 0(a0)            # Amb offset 0 accedim a mat1 i carreguem el seu valor
10     lb a3, 8(a0)            # Amb offset 8 accedim a mat2 i carreguem el seu valor
11     add a4, a2, a3          # Sumem els valors de les dues matrius i ho guardem a a4
12     sb a4, 16(a0)           # Guardem la suma a mat3 accedint amb un offset de 16 i a0
13     addi a0, a0, 1          # Augmentem en 1 per accedir al següent element
14     addi a1, a1, -1         # Decrementem en 1 fins que sigui 0
15     bnez a1, loop           # Iterem en el bucle fins haber sumat tots els elements de les dues matrius
```

Figura 2.1: Primera implementació en Ripes

Les traduccions presenten diverses diferències respecte a les implementacions en el simulador i8085.
En primer lloc en Ripes no podem operar des de memòria, així que hem de carregar tots els elements de les matrius als diversos registres de la cpu. També podem usar un sol registre per adreçar les matrius ja que amb diversos *offsets* les recorrem totes. En canvi, en i8085 s'ha de

```
.data
mat1: .byte 1, 2, 3, 4, 5,
mat2: .byte 6, 7, 8, 9, 0,
.text
    la a0, mat1             # Carreguem a a0 el valor de mat1
    addi a1, zero, 5        # Assignem el valor del registre a1 a 5
loop:
    lb a2, 0(a0)            # Amb offset 0 accedim a mat1 i carreguem el seu valor
    lb a3, 8(a0)            # Amb offset 8 accedim a mat2 i carreguem el seu valor
    add a3, a2, a3          # Sumem els valors de les dues matrius i ho sobreescribim a a3
    sb a3, 8(a0)            # Guardem la suma a mat2 (sobreescribim) accedint amb un offset de 8 i a0
    addi a0, a0, 1          # Augmentem en 1 per accedir al següent element
    addi a1, a1, -1         # Decrementem en 1 fins que sigui 0
    bnez a1, loop           # Iterem en el bucle fins haber sumat tots els elements de les dues matrius
```

Figura 2.2: Segona implementació en Ripes

carregar l'adreça de cada matriu a un parell de registres diferent.
A diferència de i8085, les operacions a Ripes no són necessàriament contra acumulador i podem realitzar la suma sense haber copiat préviament els valors de mat1 i mat3 (en cas d'utilitzar la tercera matriu). Fem l'operació i un cop tenim el resultat a un registre, ho guardem a memòria. Per tant podem executar el mateix programa que al simulador i8085 en només un bucle de les mateixes iteracions (cinc iteracions).

A continuació observem una comparativa per a cada implementació en cada simulador (les mides es troben en *bytes*:

	Mida de codi	**Mida de codi mitjana**	**Accessos a memòria**	**Cicles d'execució**	**Cicles mitjans**
i8085	31	1.63	25	477	6.71
Ripes	40	4	15	55	1.45

Table 2.3: Taula de cicles de la primera implementació

	Mida de codi	**Mida de codi mitjana**	**Accessos a memòria**	**Cicles d'execució**	**Cicles mitjans**
i8085	18	1.64	15	263	6.74
Ripes	40	4	15	55	1.45

Table 2.4: Taula de cicles de la primera implementació

A i8085 en la primera implementació realitzem 10 accessos a memòria al primer bloc (**carga**): un pel moviment de la posició adreçada pel parell de registres HL a l'acumulador (**MOV A,M**) i un altre per guardar el contingut de l'acumulador a la direcció de memòria adreçada pel parell de registres BC (**STAX B**). Al segon bloc de codi en realitzem 15: un per carregar l'element a sumar de mat3 (**LDAX B**), un altre per sumar amb el valor corresponent de mat2 (**ADD M**) i un altre per guardar el resultat a la posició de mat3 corresponent (**STAX B**). Tots els accessos es realitzen a bucles que iteren 5 cops i, per tant, en total tenim 25 accessos.
En canvi en la segona implementació suprimim el bloc de **carga** i el segon bloc és igual en nombre d'accessos a memòria, per tant en total tenim 15 accessos.

Les dades de Ripes són mes senzilles d'obtenir.
Totes les instruccions són de 4 *bytes* (32 bits) i s'executen en 5 cicles de rellotge usant R5SP, amb *pipeline*. Com realitzem 10 instruccions en les dues implementacions (la pseudoinstrucció **la**

a0, mat1 consta de dues), la mida del codi és de 10·4 = 40 bytes, i la mida mitjana per instrucció de 4 bytes ja que totes tenen la mateixa.
Només accedim 15 cops a memòria per a les dues implementacions (per carregar els valors de les matrius mat1 i mat2, i guardar-los en mat3 o en mat2 (3 accessos · 5 elements a les matrius). Finalment, consultant la taula que resumeix l'execució del programa podem saber el nombre de cicles totals i el nombre de cicles mitjans per instrucció (55 cicles i executem 38 instruccions: 1.45 cicles per instrucció).

3 Exercici 2

En aquest segon exercici se'ns demana una subrutina que codifiqui una zona de memòria mitjançant una XOR entre cada *byte* de la zona de dades i la clau. S'ha de guardar el valor dels registres a la pila per no sobreescriure'ls i, per tant, modificar-los a l'execució de la subrutina.
Els dos codis amb les dues implementacions pertinents (amb tercera matriu i sense) són semblants als de l'anterior exercici però amb certes modificacions.
Primera implementació

```
.define
    num 5
    key1 45h ; clau per codificar mat1
    key2 C9h ; clau per codificar mat2
    key3 A4h ; clau per codificar mat3
    pila 80h ; posició per inicialitzar la pila
.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, pila ; HL <= pila
    SPHL ; SP <= HL = pila
    LXI H, mat1
    LXI B, mat3
    MVI D, num
carga:
    MOV A,M
    STAX B
    INX H
    INX B
    DCR D
    JNZ carga
    LXI B,mat3
    MVI D,num
loop:
    LDAX B
    ADD M
    STAX B
    INX H
    INX B
    DCR D
    JNZ loop
    LXI H,mat1
    MVI B,key1
    CALL sub_codificadora
    LXI H,mat2
    MVI B,key2
    CALL sub_codificadora
    LXI H,mat3
    MVI B,key3
    CALL sub_codificadora
    HLT
```

```
44
45 .org 150h
46 sub_codificadora:
47     PUSH H
48     PUSH B
49     PUSH D
50     PUSH PSW
51     MVI D, num
52 codifica:
53     MOV A, M
54     XRA B
55     MOV M, A
56     INX H
57     DCR D
58     JNZ codifica
59     POP PSW
60     POP D
61     POP B
62     POP H
63 RET
```

Segona implementació

```
1  .define
2      num 5
3      key1 45h ; clau per codificar mat1
4      key2 A4h ; clau per codificar mat2
5      pila 80h
6  .data 00h
7      mat1: db 1,2,3,4,5
8      mat2: db 6,7,8,9,0
9  .org 100h
10     LXI H, pila ; HL <= pila
11     SPHL ; SP <= HL = pila
12     LXI H, mat1
13     LXI B, mat2
14     MVI D, num
15 loop:
16     LDAX B
17     ADD M
18     STAX B
19     INX H
20     INX B
21     DCR D
22     JNZ loop
23     LXI H,mat1
24     MVI B,key1
25     CALL sub_codificadora
26     LXI H,mat2
27     MVI B,key2
28     CALL sub_codificadora
29 HLT
30 .org 150h
31 sub_codificadora:
32     PUSH H
33     PUSH B
34     PUSH D
35     PUSH PSW
36     MVI D, num
37 codifica:
38     MOV A, M
39     XRA B
40     MOV M, A
```

```
41      INX H
42      DCR D
43      JNZ codifica
44      POP PSW
45      POP D
46      POP B
47      POP H
```

Abans del propi algorisme del programa, hem definit 3 constants "claus" per a codificar les diferents zones de memòria del programa (tres matrius en el cas de la primera implementació i dues en el cas de la segona implementació). També hem definit una constant quer serveix com a posició per inicialitzar la pila, acció necessària per a evitar treballar amb direccions de memòria que poguin portar a conflicte.

A l'inici de programa afegim dues instruccions, la primera carrega al parell de registres HL la posició desitjada de la pila (80h) (serà suficient ja que només guardem el contingut de 5 parells de registres a la pila, necessitant només 20 *bytes* de memòria, 2 per registre, i, per tant, 10 posicions de memòria: de la 70h a la 79h, que estan disponibles).

Al final del programa es realitzen crides a les diverses subrutines per a codificar les diferents matrius amb les claus definides prèviament. L'inici de la zona de memòria es guarda en el parell de registres HL i el registre B guarda la clau amb la que es codificarà la zona de memòria d'aquell moment.

La zona de memòria de la subrutina es una zona lliure i disponible per escriure les instruccions.

3.1 Subrutina

Dins de la subrutina el primer pas és guardar els reigstres que modificarem per tal de no alterar els seus valors i guardar al registre D el nombre d'elements de la matriu per tal de codificar-los tots.
Dins del bucle de la subrutina i en cada iteració codifiquem cada element de la matriu; primer carreguem el valor dels registres H i L a l'acumulador, codifiquem i després ho guardem en la mateixa posició de memòria d'on hem agafat l'element sense codificar. Finalment recuperem el valor dels registres i finalitzem la subrutina.

Per recuperar els valors dels registres és important enrecordar-se de que la pila segueix una estructura LIFO i, per tant, hem d'extreure els valors en ordre invers al que els hem carregat.

La primera implementació suma les matrius mat1 i mat2 i guarda el resultat en la matriu mat3 codificant posteriorment els elements de les matrius mitjançant l'operació lògica XOR.

La segona implementació és molt similar, però tal i com hem explicat a l'exercici 1, sobreescribim els elements de mat2 per a guardar el resultat de la suma de les dues matrius i així evitar utilitzar una tercera.

Els valors codificats d'ambdues implementacions són els que es mostren a les taules següents (amb les claus mostrades al codi).

Matriu	**Valors després de sumar**	Valors després de codificar
mat1	1h, 2h, 3h, 4h, 5h	44h, 47h, 46h, 41h, 40h
mat2	6h, 7h, 8h, 9h, 0h	CFh, CEh, C1h, C0h, C9h
mat3	7h, 9h, Bh, Dh, 5h	A3h, ADh, AFh, A9h, A1h

Table 2.5: Matrius de la primera implementació

Matriu	Valors després de sumar	Valors després de codificar
mat1	1h, 2h, 3h, 4h, 5h	44h, 47h, 46h, 41h, 40h
mat2	7h, 9h, Bh, Dh, 5h	A3h, ADh, AFh, A9h, A1h

Table 2.6: Matrius de la segona implementació

3.2 Pregunta 4

Quina instrucció fem servir en tots dos casos per assignar la posició inicial al registre SP?
És necessari inicialitzar la posició inicial de la pila (el contingut inicial del registre SP, *Stack Pointer*). La instrucció usada és **SPHL**, que carrega al SP el contingut del parell de registres HL (per això guardem a HL l'immediat *pila* a l'inici del programa). A Ripes seria tan senzill com assignar la posició que nosaltres vulguem directament: el registre SP és el x2. Per tant, amb l'operació **ADDI x2, x0, imm** podríem carregar al SP la posició inicial desitjada (és només un exemple ja que x0 és el registre 0 i en realitat aquesta instrucció seria un moviment de l'immediat especificat al SP).

3.3 Pregunta 5

Quina es la instrucció utilitzada per guardar el PC en la pila quan treballem amb subrutines? I per recuperar de nou el valor del PC?
La instrucció utilitzada és el **CALL etiqueta**, en aquest cas *etiqueta* és sub_codificadora.
Per recuperar el valor del PC usem la instrucció **RET**, però hem hagut d'extreure prèviament de la pila tot allò que hem carregat per no carregar al PC quelcom erroni.

3 EXERCICIS TEORICOPRÀCTICA 5

1 Exercici primera instrucció

Indiqueu quines són les entrades que ha de tenir la U.C. i quines són les sortides per executar aquesta instrucció
La entrada de la U.C és el opcode 0010111 i les sortides en el bus de control són PC<= PC+4, Enable Write, ALU <= ADD (bits de selecció d'aquesta operació), M0 <=0, M1 <=1, M2 <=0, M3<=01, DWR<=A, DLR1 <=x, DLR2 <=x on x vol dir que no importa si el bit és 1 o 0.

2 Exercici segona instrucció

Indiqueu quines són les entrades que ha de tenir la U.C. i quines són les sortides per executar aquesta instrucció
La entrada de la U.C és el opcode 0010011 i les sortides en el bus de control són PC<= PC+4 Enable Write, ALU <= ADD (bits de selecció d'aquesta operació), M0 <=0, M1 <=0, M2 <=0, M3<=01, DWR<=A, DLR1 <=A, DLR2 <=x on x vol dir que no importa si el bit és 1 o 0.

3 Exercici tercera instrucció

Indiqueu quines són les entrades que ha de tenir la U.C. i quines són les sortides per executar aquesta instrucció
La entrada de la U.C és el opcode 0000011 i les sortides en el bus de control són PC<= PC+4 Enable Write, ALU <= ADD (bits de selecció d'aquesta operació), M0 <=0, M1 <=0, M2 <=0, M3<=01, DWR<=B, DLR1 <=A, DLR2 <=x on x vol dir que no importa si el bit és 1 o 0.

4 Exercici cinquena instrucció

Indiqueu quines són les entrades que ha de tenir la U.C. i quines són les sortides per executar aquesta instrucció
La entrada de la U.C és el opcode 0110011 i les sortides en el bus de control són PC<= PC+4 Enable Write, ALU <= ADD (bits de selecció d'aquesta operació), M0 <=0, M1 <=0, M2 <=1, M3<=01, DWR<=D, DLR1 <=B, DLR2 <=C.

5 Exercici sisena instrucció

Exactament igual que la primera però... Que fa la propera instrucció?
La propera instrucció suma l'inmediat -20 al registre x14 i ho guarda al mateix registre.

6 Exercici vuitena instrucció

Indiqueu quines són les entrades que ha de tenir la U.C. i quines són les sortides per executar aquesta instrucció
La entrada de la U.C és el opcode 0100011 i les sortides en el bus de control són PC<= PC+4 Enable Write, @([X10] + Imm)<= X13 Mem Write enabled, ALU <= ADD (bits de selecció d'aquesta operació), M0 <=0, M1 <=0, M2 <=1, M3<=01, DWR<=0, DLR1 <=A, DLR2 <=D.

7 Exercici final

Com seria la UC necessària per executar aquest programa?
En el programa no hem especificat cap entrada del registre State, per tant no l'utilitzem, d'altra banda l'entrada de l'opcode de les instruccions hauria de ser de 7 bits. El bus de control hauria de tenir tants bits com bits de selecció necessitem per executar les instruccions del programa, per tant 8 bits de selecció + x bits de selecció per seleccionar la operació de la ALU (segons el nombre d'operacions que tingui la unitat aritmetico-lògica) + 32 bits per al PC (vuit instruccions).

4 Conclusions

La primera pràctica del simulador i8085 ha servit per començar a experimentar i testejar el nou simulador i conèixer les principals diferències amb l'anterior simulador que utilitzàvem, el Ripes. D'aquesta manera he pogut conèixer el format de les principals instruccions i diversos modes d'adreçament, així com la estructura del propi simulador.

També ha sigut útil per començar a programar amb subrutines, en aquest cas per codificar zones específiques de memòria.

En general ha sigut una pràctica interessant però amb un grau de dificultat elevat, suposo que pel canvi de simulador, el qual implica canvi en el format de l'estructura del programa, del format de les instruccions, de l'adreçament a memòria, etc. La primera pràctica