Introducció als Ordinadors

Pràctica 5

Mario Vilar

17 de maig de 2022

Índex

1 Introducció

El 8085 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.

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.

2 Cos del treball

2.1 Teorico-pràctica

Exercici 1. *Indiqueu quines són les entrades que ha de tenir la U.C. i quines són les sortides per executar aquesta instrucció. (Primera instrucció.)*

Demostració. La primera instrucció s'encarrega de sumar l'immediat de 20 bits, `0x10000000` (els 20 bits més significatius de la posició de memòria de `xx` i el contingut del descodificador de l'immediat), al `pc`, en aquell moment `0x00000000`, i escriu el resultat en el registre `a0`, associat al nom `x10`. Es consulta el contingut de la posició `0x00000000` a la memòria d'instruccions, que resulta ser `0x10000517` i això arriba al pas de descodificació.

Les sortides de la unitat de control serveixen per modular el funcionament del nostre processador a través de senyals específiques en els seus components i ja han estat especificades en l'enunciat de pràctica (en la seva majoria). Són les següents:

1. `WrEn PC <= 1`,
2. `WrEn REG <= 1`,
3. `WrEn MEM <= 0`,
4. `Branch <= 0`,
5. `ALU <= ADD`,
6. `MUX0 <= 0`,
7. `MUX1 <= 0`,
8. `MUX2 <= 1`,
9. `MUX3 <= 01`,
10. `DEC_REG <= AUIPC`.

Per definició, l'*op-code* és una de les entrades de la nostra unitat de control (en aquest cas, serà `0010111`) i les altres dues seran `funct3=x` i `funct7=x`, ja que la instrucció `auipc` és `U-type`. ◁

Exercici 2. *Indiqueu quines són les entrades que ha de tenir la U.C. i quines són les sortides per executar aquesta instrucció. (Segona instrucció.)*

Demostració. La segona instrucció s'encarrega de sumar l'immediat de 12 bits, `0x00000000` (un zero), al registre `x10`, que ja contenia `0x10000000`, de manera que obtenim el `0x10000000` de 32 bits que emmagatzema la posició de memòria d'`xx`. Es consulta el contingut de la posició `0x00000004` a la memòria d'instruccions, que resulta ser `0x00050013` i això passa al pas de descodificació.

Les sortides de la unitat de control serveixen per modular el funcionament del nostre processador a través de senyals específiques en els seus components i ja han estat especificades en l'enunciat de pràctica (en la seva majoria). Són les següents:

1. `WrEn PC <= 1`,
2. `WrEn REG <= 1`,
3. `WrEn MEM <= 0`,
4. `Branch <= 0`,
5. `ALU <= ADD`,
6. `MUX0 <= 0`,
7. `MUX1 <= 1`,
8. `MUX2 <= 1`,
9. `MUX3 <= 01`,
10. `DEC_REG <= ADDI`.

Per definició, l'*op-code* és una de les entrades de la nostra unitat de control (en aquest cas, serà `0010011`) i les altres dues seran `funct3=000` i `funct7=x`, ja que la instrucció `addi` és `I-type`. ◁

Exercici 3. *Indiqueu quines són les entrades que ha de tenir la U.C. i quines són les sortides per executar aquesta instrucció. (Tercera instrucció.)*

Demostració. La tercera instrucció carrega el contingut de la direcció apuntada per `a0` en `a1`. Fixem-nos: es consulta el contingut de la posició `0x00000008` a la memòria d'instruccions, que resulta ser `0x00052583` i això passa al pas de descodificació.

La descodificació ens dona l'índex del registre a llegir, i es llegeix el banc de registres, obtenint el contingut de `a0`: la direcció de memòria esmentada en els passos anteriors, `0x10000000`. Passa asíncronament per la ALU i es llegeix, també asíncronament, la memòria, obtenint `0x00000004`. S'escriu, ara de manera síncrona, al banc de registres, a l'índex 11 (`x11`).

Les sortides de la unitat de control serveixen per modular el funcionament del nostre processador a través de senyals específiques en els seus components i ja han estat especificades en l'enunciat de pràctica (en la seva majoria). Són les següents:

1. `WrEn PC <= 1`,
2. `WrEn REG <= 1`,
3. `WrEn MEM <= 0`,
4. `Branch <= 0`,
5. `ALU <= ADD`,
6. `MUX0 <= 0`,
7. `MUX1 <= 1`,
8. `MUX2 <= 1`,
9. `MUX3 <= 10`,
10. `DEC_REG <= LW`.

Per definició, l'*op-code* és una de les entrades de la nostra unitat de control (en aquest cas, serà `0000011`) i les altres dues seran `funct3=010` i `funct7=x`, ja que la instrucció `lw` és `I-type`. ◁

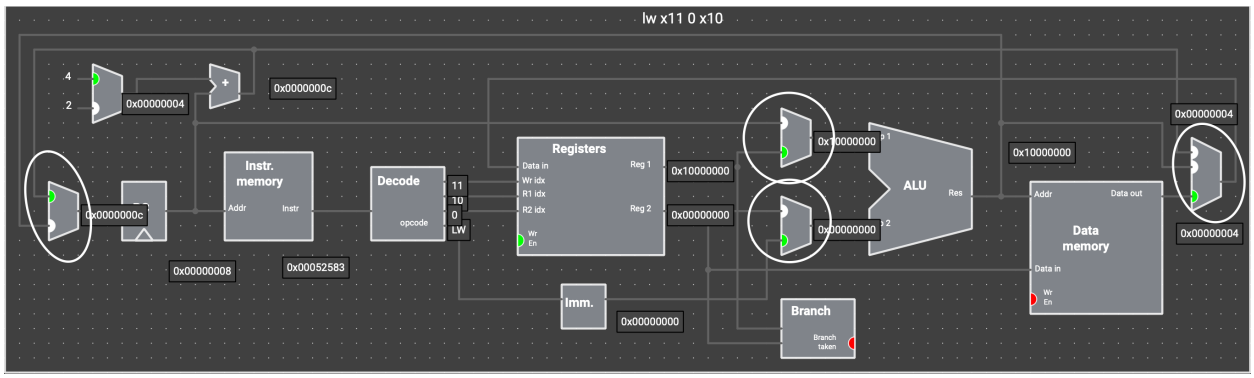


Figura 1: Esquema de la tercera instrucció.

Exercici 4. *Indiqueu quines són les entrades que ha de tenir la U.C. i quines són les sortides per executar aquesta instrucció. (Quarta instrucció.)*

Demostració. La quarta instrucció carrega el contingut de la direcció apuntada per `[a0]+4` en `a2`. Fixem-nos: es consulta el contingut de la posició `0x0000000c` a la memòria d'instruccions, que resulta ser `0x000452603` i això passa al pas de descodificació.

La descodificació ens dona l'índex del registre a llegir (`x10`), i es llegeix el banc de registres, obtenint el contingut de `a0`: la direcció de memòria esmentada en els passos anteriors, `0x10000000`. Passa asíncronament per la ALU juntament amb l'immediat, l'`offset`, que és `0x00000004`. El resultat `0x10000004` es llegeix, també asíncronament, la memòria, obtenint `0x00000005`. S'escriu, ara de manera síncrona, al banc de registres, a l'índex `x12`.

Les sortides de la unitat de control serveixen per modular el funcionament del nostre processador a través de senyals específiques en els seus components i ja han estat especificades en l'enunciat de pràctica

(en la seva majoria). Són les següents:

1. `WrEn PC <= 1`,
2. `WrEn REG <= 1`,
3. `WrEn MEM <= 0`,
4. `Branch <= 0`,
5. `ALU <= ADD`,
6. `MUX0 <= 0`,
7. `MUX1 <= 1`,
8. `MUX2 <= 1`,
9. `MUX3 <= 10`,
10. `DEC_REG <= LW`.

Per definició, l'*op-code* és una de les entrades de la nostra unitat de control (en aquest cas, serà `0000011`) i les altres dues seran `funct3=010` i `funct7=x`, ja que la instrucció `lw` és `I-type`. ◁

Exercici 5. *Indiqueu quines són les entrades que ha de tenir la U.C. i quines són les sortides per executar aquesta instrucció. (Cinquena instrucció.)*

Demostració. La cinquena instrucció s'encarrega de sumar els registres `a2` i `a1` per emmagatzemar-ho en `a1`. Fixem-nos: es consulta el contingut de la posició `0x00000010` a la memòria d'instruccions, que resulta ser `0x00b606b3` i això passa al pas de descodificació.

La descodificació ens dona l'índex del registre a llegir (`x12` i `x11`), i es llegeix el banc de registres, obtenint el contingut d'`a2` (`0x00000005`) i `a1` (`0x00000004`). El contingut dels dos registres passen asíncronament a la ALU i se sumen, obtenint `0x00000009`. Amb la sel·lecció del multiplexor, aquest resultat passa com a dada d'entrada al registre i s'escriu síncronament en l'índex d'escriptura, `x13` (`a3`).

Les sortides de la unitat de control serveixen per modular el funcionament del nostre processador a través de senyals específiques en els seus components i ja han estat especificades en l'enunciat de pràctica (en la seva majoria). Són les següents:

1. `WrEn PC <= 1`,
2. `WrEn REG <= 1`,
3. `WrEn MEM <= 0`,
4. `Branch <= 0`,
5. `ALU <= ADD`,
6. `MUX0 <= 0`,
7. `MUX1 <= 1`,
8. `MUX2 <= 0`,
9. `MUX3 <= 01`,

10. `DEC_REG <= ADD`.

Per definició, l'*op-code* és una de les entrades de la nostra unitat de control (en aquest cas, serà `0110011`) i les altres dues seran `funct3=000` i `funct7=0000000`, ja que la instrucció `ADD` és `R-type`. ◁

Exercici 6. *Indiqueu quines són les entrades que ha de tenir la U.C. i quines són les sortides per executar aquesta instrucció. (Setena instrucció.)*

Demostració. La setena instrucció s'encarrega de sumar l'immediat de 12 bits, `0xffffffec` (un −20), al registre `x14`, que ja contenia `0x10000014`, de manera que obtenim el `0x10000000` de 32 bits que emmagatzema la posició de memòria d'`xx`. Es consulta el contingut de la posició `0x00000018` a la memòria d'instruccions, que resulta ser `0xfec70713` i això passa al pas de descodificació.

Les sortides de la unitat de control serveixen per modular el funcionament del nostre processador a través de senyals específiques en els seus components i ja han estat especificades en l'enunciat de pràctica (en la seva majoria). Són les següents:

1. `WrEn PC <= 1`,
2. `WrEn REG <= 1`,
3. `WrEn MEM <= 0`,
4. `Branch <= 0`,
5. `ALU <= ADD`,
6. `MUX0 <= 0`,
7. `MUX1 <= 1`,
8. `MUX2 <= 1`,
9. `MUX3 <= 01`,
10. `DEC_REG <= ADDI`

Per definició, l'*op-code* és una de les entrades de la nostra unitat de control (en aquest cas, serà `0010011`) i les altres dues seran `funct3=000` i `funct7=x`, ja que la instrucció `addi` és `I-type`. ◁

Exercici 7. *Indiqueu quines són les entrades que ha de tenir la U.C. i quines són les sortides per executar aquesta instrucció. (Vuitena instrucció.)*

Demostració. La vuitena instrucció guarda el contingut de `a3` en `[a2]+8`. Fixem-nos: es consulta el contingut de la posició `0x0000001c` a la memòria d'instruccions, que resulta ser `0x00d52423` i això passa al pas de descodificació.

La descodificació ens dona l'índex del registre a llegir (`x10` i `x13`), i es llegeix el banc de registres, obtenint el contingut d'`a0` i `a3`: la direcció de memòria esmentada en els passos anteriors, `0x10000000` i el contingut d'`a3`, `0x00000009`. El contingut del primer registre passa asíncronament per la ALU jun-

tament amb l'immediat, l'`offset`, que és `0x00000008`. Finalment, a la memòria de dades hi entra el resultat de l'ALU com a camp d'adreça i el contingut d'`a3` com a dada a escriure.

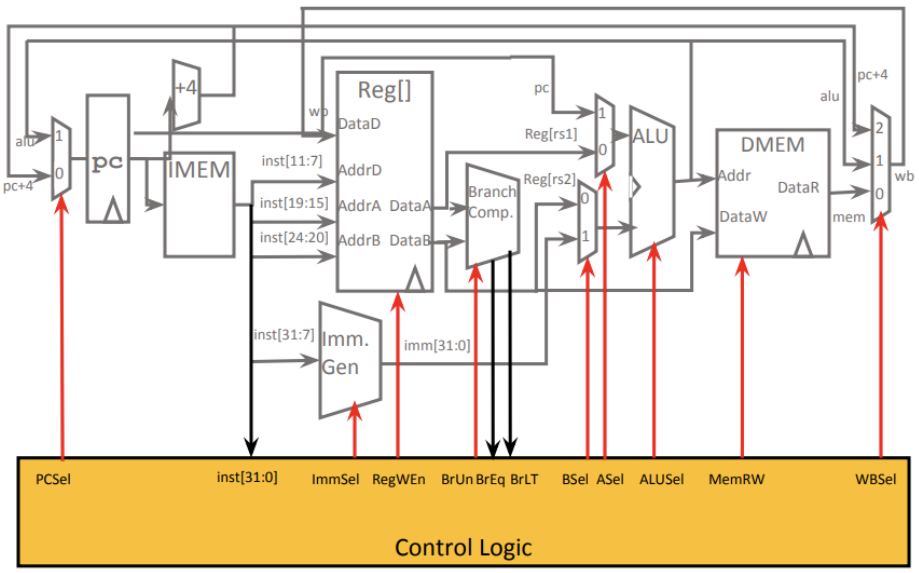
Les sortides de la unitat de control serveixen per modular el funcionament del nostre processador a través de senyals específiques en els seus components i ja han estat especificades en l'enunciat de pràctica (en la seva majoria). Són les següents:

1. `WrEn PC <= 1`,
2. `WrEn REG <= 0`,
3. `WrEn MEM <= 1`,
4. `Branch <= 0`,
5. `ALU <= ADD`,
6. `MUX0 <= 0`,
7. `MUX1 <= 1`,
8. `MUX2 <= 1`,
9. `MUX3 <= 01`,
10. `DEC_REG <= SW`

Per definició, l'*op-code* és una de les entrades de la nostra unitat de control (en aquest cas, serà `0100011`) i les altres dues seran `funct3=010` i `funct7=x`, ja que la instrucció `lw` és `S-type`. ◁

Exercici 8. *Com seria la UC necessària per executar aquest programa?*

Demostració. He triat la següent representació del processador, ja que il·lustra perfectament el que necessitem en la Unitat de Control. De fet, és molt similar als senyals que ja havíem establert:



◁

2.2 Pràctica

2.2.1 Part 1

Escriviu dos programes diferents per sumar dues matrius de 5x1 i desar el resultat en una tercera, podeu fer servir adreçament indirecte, per parella de registres o per registres. Penseu en dues hipòtesi, primera, les matrius d'entrada s'han de conservar, és a dir, el resultat s'emmagatzema sobre mat3 i segona, desem el resultat sobre una de les dues matrius d'entrada, mat3 és irrellevant. El programa hauria d'utilitzar un comptador, instruccions d'increment, comparació i salt condicional. Feu la suma de números hexadecimals i sense tenir en compte números que pugui donar overflow.

Exercici 9. *Calculeu les mides del codi del vostre programa i el nombre de cicles per a la seva execució.*

Demostració.

Primerament, el programa proposat és el següent:

```
.data 50h
mat1: db
   1,2,3,4,5
mat2: db
   6,7,8,9,0
mat3: db
   0,0,0,0,0
.org 100h
lxi d,mat1
lxi h,mat2
lxi b,mat3
loop:
ldax d
add m
stax b
inr c
inr e
inr l
mvi a,mat2
cmp e
jnz loop
hlt
```

1. Carreguem la posició de memòria de `mat1` (`0050h`) en els registres `DE` mitjançant un direccionament immediat; `HL` per `mat2` (`0055h`) i `BC` per `mat3` (`005Ah`) de manera totalment anàloga.
2. Entrem en el `loop`. Carreguem el contingut que hi hagi a la posició de memòria de `DE` a l'acumulador (és a dir, l'adreça `0050h`, que conté el primer nombre de `mat1`, un 1) amb `lxi`.
3. Sumem el contingut de l'acumulador amb allò que hi hagi al contingut de la posició de memòria especificada per `HL` amb `add m`. Com que `HL` tenia com a posició de memòria el primer nombre de `mat2`, sumem el primer nombre de `mat1` i el primer nombre de `mat2`. Aquest resultat queda en l'acumulador.
4. Copiem el resultat de l'operació anterior a la direcció de memòria que emmagatzema `BC` (`005Ah`), és a dir, sobre el primer nombre de `mat3`.
5. Incrementem els valors dels registres `C`, `E` i `L` per a poder operar amb els següents dos nombres de la matriu i guardar en la posició corresponent. En altres paraules, a la propera iteració estarem treballant en les posicions: `0051h`, `0056h`, `005Bh`, respectivament.
6. Carreguem a l'acumulador l'immediat `mat2` (la posició de memòria on comença `mat2`, `0055h`).
7. Comparem si aquest valor és igual al que trobem al registre `E`. Recordem que al registre `E` teníem `51h`. Per tant, si aquests dos valors coincidissin, tindríem que ja hem arribat al final de la matriu `mat1` (o, en altres paraules, a la primera posició de `mat3` en termes de posicionament de memòria) i hem d'aturar la iteració. En qualsevol cas, el resultat es veurà reflectit en el bit d'estat de *zero* `Z`.
8. Si el bit de *zero* resulta activat podrem acabar el bucle i, per tant, acabar el programa. Si no, saltarem a `loop` i tornarem a fer tot el procés, amb el següent nombre que correspongui dels vectors.

Les operacions que més temps triguen en executar-se són les de `lxi` i també les de `inr`. En particular,

aquesta última serveix per incrementar en 1 el valor del contingut del registre que conté la instrucció. Com ho fem tres vegades, estem parlant de tres registres, i no només d'un registre.

Pel que fa a la pregunta de quant ocupa el nostre codi, s'entén quantes posicions de memòria ocupa la part executable del nostre codi. D'aquesta manera, cal distingir entre les dades i la memòria: la part de codi de programa que ens interessa va des de la posició `0100h` fins a la posició `0115h`, ocupant en total 21 posicions de memòria.

I tenint en compte les posicions que usem per guardar dades, que van des de la `50h` fins a la `64h`, en total 15 posicions de memòria.

Calculem els cicles de rellotge que triga el programa en executar-se. Tenim 3 operacions `lxi` fora del `loop`, això són 30 cicles (10 per operació). El `loop` ocupa 72 cicles de rellotge quan es produeix el salt i 69 quan no. Tenim 5 iteracions, 4 amb salt i una sense salt, per tant, $4 \cdot 72 + 69 = 357$ cicles de rellotge. Sumem els 30 que havíem deixat fora del `loop` i fan un total de 387 cicles de rellotge. Si tenim en compte el `hlt` (4 cicles) tindrem, finalment, 391 cicles de rellotge.

Instrucció	Cicles de rellotge
`lxi RP,DA16`	10
`lxi RP,DA16`	10
`lxi RP,DA16`	10
`loop`	∅
`ldax RP`	7
`add M`	7
`stax RP`	7
`inr reg`	10
`inr reg`	10
`inr reg`	10
`mvi reg,DAT`	7
`cmp reg`	4
`jnz label`	7/10
acaba `loop`	∅
`hlt`	4

Figura 2: Representació dels cicles de rellotge que necessita el programa.

Finalment, la mitjana de cicles per instrucció de tota l'execució del programa és:

$$\frac{391 \text{ cicles}}{3 + 5 \cdot 9 + 1 \text{ instruccions}} = 7.97959 \approx 8 \text{ cicles/instrucció} \tag{2.1}$$

I si no tenim en compte l'execució i simplement ens mirem les instruccions que es llisten en codi, el total és el següent:

$$\begin{aligned} &\frac{10 + 10 + 10 + 7 + 7 + 7 + 10 + 10 + 10 + 7 + 7 + 4 \text{ cicles}}{13 \text{ instruccions}} = 7.6153846 \text{ cicles/instrucció} \\ \leq &\frac{10 + 10 + 10 + 7 + 7 + 7 + 10 + 10 + 10 + 7 + 10 + 4 \text{ cicles}}{13 \text{ instruccions}} = 7.846153846\overline{15} \text{ cicles/instrucció} \end{aligned} \tag{2.2}$$

Com veiem, tant d'una manera com l'altra estem parlant, aproximadament, de 8 cicles per instrucció. Sigui com sigui, la diferència numèrica entre els resultats no és negligible.

A la pàgina següent, el codi sense la matriu auxiliar.

Pel que fa al programa sense `mat3`, és el següent:

```
.data 50h
mat1: db
   1,2,3,4,5
mat2: db
   6,7,8,9,0
.org 100h
lxi d,mat1
lxi h,mat2
loop:
ldax d
add m
mov m,a
inr e
inr l
mvi a,mat2
cmp e
jnz loop
hlt
```

1. Carreguem les posicions de memòria en els registres `DE` per `mat1` (`0050h`); `HL` per `mat2` (`0055h`).
2. Entrem al bucle. Carreguem el que hi hagi a la posició de memòria emmagatzemada en el registre `DE` a l'acumulador (és a dir, el primer nombre de `mat1`, un 1).
3. Sumem el contingut de l'acumulador amb el valor obtingut a partir del direccionament d'`HL`. Com que `HL` referenciava la posició de memòria on hi ha el primer nombre de `mat2`, sumem el primer nombre de `mat1` i el primer nombre de `mat2`. El resultat es guarda, un altre cop, en acumulador.
4. Ens interessa guardar el resultat de l'operació anterior en la posició de memòria corresponent de `mat2` (si hem sumat el primer valor d'ambdues matrius, per exemple, en `0055h`). Copiem l'acumulador, el registre especificat per `A`, en la posició de memòria direccionada pels registres `HL`, és a dir, sobre el primer nombre de `mat2`, mitjançant la instrucció `MOV`.
5. Incrementem en 1 els valors dels registres `E` i `L` per a poder operar amb els següents dos nombres i posicions de la matriu.
6. Ara volem assegurar-nos quan aturar la iteració, això és, quan s'ha arribat al final de la matriu.
   - Carreguem a l'acumulador la posició en 8 bits en què comença `mat2` (`55h`) i comparem si aquest valor és igual al que trobem al registre `E`.
   - Recordem que al registre `E` teníem el valor de la posició de memòria de l'últim nombre que havíem sumat més una unitat (`51h`).
   - Per tant, si aquests dos valors coincidissin, tindríem que ja hem arribat al final de tots els nombres que hem de sumar i s'activaria el bit de *zero*.
7. Si s'activa el bit de *zero* podrem acabar el programa. Si no, saltarem a `loop` i tornarem a fer tot el procés, amb el següent nombre que correspongui dels vectors.

Pel que fa a la pregunta de quant ocupa el nostre codi, s'entén quantes posicions de memòria ocupa la part executable del nostre codi. D'aquesta manera, cal distingir entre les dades i la memòria: la part de codi de programa que ens interessa va des de la posició `0100h` fins a la posició `0111h`, ocupant en total 17 posicions de memòria.

I tenint en compte les posicions que usem per guardar dades, que van des de la `50h` fins a la `59h`, en total 10 posicions de memòria.

Altre cop, les operacions que més temps triguen són les de `lxi` i també les de `inr`. En particular, aquesta última serveix per incrementar en 1 el valor del contingut del registre que conté la instrucció. Com ho fem dues vegades, estem parlant de dos registres, i no només d'un registre.

Hem de calcular els cicles que triga el nostre programa en executar-se. Seguirem un comptat anàleg

al que hem fet per 2. Fora del `loop` tenim 20 cicles (10 cicles per cada crida a `lxi`). El `loop` ocupa 62 cicles si saltem a `loop` i 59 cicles si no hi saltem, ja que la instrucció `jnz` depèn de l'etiqueta en el cas que la iteració no continuï. Tenim 5 iteracions, 4 amb salt i una sense, per tant $62 \cdot 4 + 59 = 307$. Més els 20 cicles de fora del loop fan un total de 327 cicles de rellotge. Si comptem també el hlt (4 cicles), fan 331 cicles. ◁

Observació 10. *Podríem haver estalviat 4 cicles per instrucció d'increment si en lloc d'haver utilitzat* `inr` *ens haguéssim servit de* `inx`*, ja que el primer són 10 i el segon, 6. Sigui com sigui, no ho tindrem en consideració en aquest exercici, però és un detall a recordar en pròximes ocasions; augmenta l'eficiència.*

Exercici 11. *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.*

Demostració. Les instruccions aritmètic-lògiques s'han de dividir en aquelles que involucren un registre (`op REG`) o bé la memòria (`op M`). En el primer tipus, tota operació aritmètica necessita de 4 cicles i en el segon, 7. Posarem el cas particular en què `op` és una suma `ADD`:

```
[ACC] <= [ACC]+[REG]    (4 cicles de rellotge)
[ACC] <= [ACC]+M[HL]    (7 cicles de rellotge)
```

En altres paraules, el direccionament indirecte mitjançant memòria de la segona opció implica que aquesta tingui un nombre de cicles més alt. En altres paraules, primer ha d'accedir al contingut dels registres `HL` per després poder accedir al contingut de memòria indicat i realitzar la operació. En qualsevol cas, totes dues sumen un parell de valors, un d'ells el mateix acumulador i emmagatzemen el resultat de la mateixa suma a l'acumulador. ◁

Exercici 12. *Escriu el mateix programa en* `RISC-V`*. Quants cicles triga a executar-se? En què simplificaria molt el codi un dels modes d'adreçament del simulador* `Ripes`*?*

Demostració. El codi proposat és el següent, que cobreix el cas en què s'utilitzen tres matrius: `mat1`, `mat2` i `mat3`. Veiem que l'adreçament indexat cobreix part de les nostres necessitats i es veu un programa molt més net. Això sí, en termes de llargada, força similar (cal esmentar que en `RISC-V` existeixen instruccions com `la` que es divideixen en dues subinstruccions).

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

.text
la a0,mat1
la a3,mat2 # auxiliar per acabar el bucle
```

```
loop:
lw a1,0(a0)
lw a2,20(a0) # offset=20 per accedir a mat2
add a2,a1,a2
sw a2,40(a0) # offset=40 per accedir a mat3
addi a0,a0,4 # passem a la posicio seguent
beq a0,a3,end
j loop

end:
nop
```

Amb el simulador de *Single-Cycle*, executem fins al final i tenim un total de 39 cicles de rellotge. A més, com ja hem comentat, la mida del codi és semblant:

$$\frac{2c\ (\text{adreça última instrucció})}{4\ (\text{step } 4)} = 11, \tag{2.3}$$

on dividim entre quatre ja que posicions de memòria successives van separades de quatre en quatre. ◁

2.2.2 Part 2

Fent ús del programa anterior, realitzeu una subrutina que codifiqui una zona de memòria. La zona de memòria s'indicarà posant al registre doble HL l'adreça de començament de la zona de dades a codificar. Aquestes dades es consideren com els paràmetres d'entrada de la subrutina. La codificació es farà mitjançant una XOR entre cada byte de la zona de dades i la clau. Els valors dels registres no han de quedar afectats per la crida a la subrutina. Feu un programa que faci us d'aquesta subrutina per provar-la amb diferents combinacions de valors de la clau i les dades per codificar.

Demostració. El codi que ens hem decantat per utilitzar és el següent:

```
.data 50h
mat1: db 1,2,3,4,5
mat2: db 6,7,8,9,0
.org 100h
lxi d,mat1
lxi h,mat2
mvi b,FFh ;mask
loop:
    ldax d
    add m
    call codifica
    inx h
    inx d
    mvi a,mat2
    cmp e
```

```
jnz loop
hlt
codifica:
push h
xra b
mov m,a
pop h
ret
```

Si la màscara és −1 (FFh), la matriu resultant és F8h, F6h, F4h, F2h, FAh. Podríem aplicar el procediment invers amb la mateixa clau per obtenir el resultat inicial:

```
.data 50h
mat1: db F8h,F6h,F4h,F2h,FAh
mat2: db 0,0,0,0,0
.org 100h
lxi d,mat1
lxi h,mat2
mvi b,FFh ;mask
loop:
    ldax d
    add m
    call codifica
    inx h
    inx d
    mvi a,mat2
    cmp e
jnz loop
hlt
codifica:
push h
xra b
mov m,a
pop h
ret
```

I la matriu resultant és: 07h,09h,0Bh,0Dh,05h, justament la mat2 final abans del pas de codificació. D'aquesta manera, hem confeccionat el nostre primer xifratge! És molt simple, ja que consisteix d'una única i molt habitual màscara. ◁

Exercici 13. *Quina instrucció fem servir en tots dos casos per assignar la posició inicial al registre SP?*

Demostració. i8085 només preservarà els registres guardats explícitament en la pila, amb PUSH i POP. El mateix passa amb Ripes. Sigui com sigui, amb una simple execució del programa SP es veu modificat una vegada hem passat la instrucció call. De fet, SP apuntarà a FFFE, que contindrà els últims dos bytes de la següent instrucció després de la subrutina, 0D, corresponents a inx h. Sintetitzant, la resposta és call. ◁

Exercici 14. *Quina és la instrucció utilitzada per guardar el PC en la pila quan treballem amb les subrutines? I per recuperar de nou el valor de PC?*

Demostració. La instrucció `call` és la que guarda la posició del program counter per després tornar-hi. Per recuperar aquesta posició utilitzem el `ret` (retorn). En lloc d'optar per fer la màscara al final, la qual cosa ens hagués obligat a fer un altre bucle i, per tant, perdre optimització, hem decidit anar-la fent sobre la marxa. ◁

Una altra aproximació al problema de la màscara, una mica més barroera:

```
.data 50h
mat1: db 1,2,3,4,5
mat2: db 6,7,8,9,0
mat3: db 0,0,0,0,0
.org 100h
lxi d,mat1
lxi h,mat2
lxi b,mat3
loop:
    ldax d
    add m
    stax b
    inx b
    inx d
    inx h
    mvi a,mat2
    cmp e
jnz loop
call codifica
hlt
codifica:
    lxi h,mat1
    mvi b,01h ;Carreguem la mascara, en aquest cas 01h
    mvi c,00h ;Inicialitzem a 0 el registre c.
    mvi d,15d ;Carreguem les posicions totals de memoria a
       codificar a d.
second_loop:
    mov a,m
    xra b
    mov m,a
    inr c ;Incrementem en 1 el valor de C
    inx h ;Incrementem en 1 el valor de HL
    mov a,c ;Carreguem el valor de C a acumulador
    cmp d ;Comparem el valor de acumulador amb D.
    jnz second_loop
    ret
```

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.