Digitális technika 2. jegyzet

Sebők Bence

2017. tavasz

Tartalomjegyzék

1 Bevezetés **2**

2 Alapismeretek **3**
2.1 Intel 8085 processzor . . . 3

3 Assembly **3**
3.1 Tanszéki szimulátor használata . . . 3
3.1.1 1. lépés: kód beillesztése . . . 4
3.1.2 2. lépés: szimuláció beállítása . . . 5
3.1.3 3. lépés: kód követése . . . 7
3.2 Utasísátok . . . 9
3.2.1 CALL . . . 9
3.3 A stack (verem) működése . . . 9
3.3.1 Stack Pointer inicializálása . . . 10
3.3.2 Írás a stackre . . . 10
3.3.3 Olvasás a stackről . . . 11
3.3.4 Példa a stack műveletekre . . . 11
3.4 Példa feladat az ellenőrző kérdések közül . . . 12

1 Bevezetés

Ez a jegyzet a BME Digitális technika 2. tantárgyhoz szeretne segítséget nyújtani. Ezen kezdeményezés célja, hogy segítsen a hallgatóknak megérteni a tananyagot. A tantárgy nehéz, erősen ajánlott előadásra, gyakorlatra járni. Ez a jegyzet csak az előadáson, gyakorlaton készült jegyzet mellé jelent segítséget, nem tanít meg a nulláról a tantárgy minden apróságára.
Ez egy hallgatói jegyzet, nincs lektorálva, se egyetemi oktató által felügyelve. Amit itt olvasol, azt csak saját felelősségre használd, hivatalos helyeken nem hivatkozási alap ez a jegyzet.

2 Alapismeretek

2.1 Intel 8085 processzor

- PC (Program Counter, program számláló): regisztárpár, ami az aktuálisan használt memóriacímet tartalmazza.
- SP (Stack Pointer, stack mutató): a stack tetejére mutató pointer (regiszterpárban tárolja ezt a 2 bájtos címet)

3 Assembly

Digitális technika 2. tantárgy során az Intel 8085 processzor programozásához használt Assembly programozási nyelvet tanítják. Ez a fejezet erről az Assemblyről szól.

3.1 Tanszéki szimulátor használata

Van egy az IIT-n fejlesztett Intel 8085 szimulátor. Ennek segítségével tetszőleges kódot tudunk futtatni egy virtuális Intel 8085 processzoron. Tetszőleges kód alatt értem, hogy bármit, ami a processzor utasításkészlete és fordítói direktívái lehetővé tesznek.
A tanszéki szimulátor a következő linken érhető el:
`http://topcat.iit.bme.hu/tools/i8085sim/i8085sim.cgi`
A szimulátor a következő módon néz ki:

i8085 Simulator

Import file into i8085 Simulator:

Choose File | No file chosen | Import

Acceptable file types: intel hex, i8085 assembly source (beta version). There is also a small help in hungarian.

Direct assembly

Type in assembly source.

Send

Online examples

Study Examples	Module Tests	Miscellaneous	Course Prezi
Gyak1a source	Hetszegmens source	instr source	c01 source
Gyak1b source	Johnson source		c02 source
Gyak1c source	KapcsLed source		c03 source
Gyak6_I source	Kepernyo source		c04 source
Gyak6_II source	Menu source		c05 source
Gyak7_I source			c06 source
Gyak7_II source			

© Doxence, 2007-2014

A következő kódot szeretnénk a szimulátorban futtatni:

```
ORG 2000h ; 2000h-tol helyezze el a kodot a fordito
LXI H, 2100h ; HL <-- 2100h
MVI M, 11h ; [2100h] = 11h
HLT ; processzor --> HALT allapot
END ; eddig forditson
```

3.1.1 1. lépés: kód beillesztése

A szimulálandó kódot a Direct assembly ablakba kell írni:

Direct assembly

Type in assembly source.

```
ORG 2000h ; 2000h-tól helyezze el a kódot a fordító
LXI H, 2100h ; HL <-- 2100h
MVI M, 11h ; [2100h] = 11h
HLT ; processzor --> HALT állapot
END ; eddig fordítson
```

Send

Ezután a Send gombra kattintva menjünk tovább, ahol ez fogad minket:

i8085 Simulator

Reset Run Stop Step Config Elapsed time: 0 phases, 0 usec

```
. Processing PASS1...
. Processing PASS2...
= No errors.
= No warnings.
```

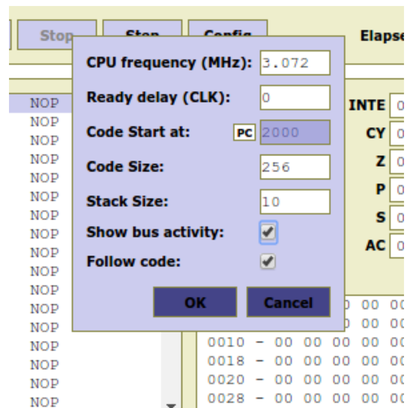
© Vajda Ferenc, 2007-2014

3.1.2 2. lépés: szimuláció beállítása

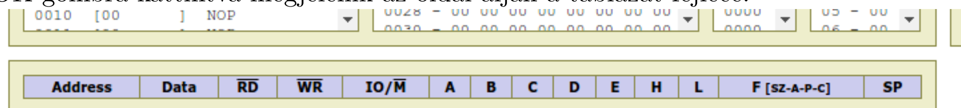
Néhány beállítást el kell végeznünk, hogy a megfelelő módon tudjuk nyomon követni a futó kódot.

1. Coda Start at: cím, ahova elhelyezzük a kódot (és kattintsunk bele a fehér téglalapba, hogy megjelenjen ott a PC felirat)
2. Show bus activity: legyen bepipálva
3. Follow code: legyen bepipálva

Az utolsó 2 opció ahhoz kell, hogy egy szép táblázatos formában jelenítse meg a kód futását és utasításonként haladjon a kód végrehajtása során. Ezek elvégzése után ezt kell látni:



Az OK gombra kattintva megjelenik az oldal alján a táblázat fejléce:



A táblázat fejléce:

1. Address: milyen címre mutat éppen a PC
2. Data: milyen adat van ezen a memóriacímen
3. $\overline{RD}$: olvasás-e a jelenlegi művelet
4. $\overline{WR}$: írás-e a jelenlegi művelet
5. $IO/\overline{M}$: a mostani utasítás memória vagy periféria műveletet hajt végre
6. regiszterek aktuális értékei
7. F [SZ-A-P-C]: flag-ek aktuális értékei
   (a) Sign flag: előjel flag
   (b) Zero flag: zérus flag
   (c) Auxillary carry flag: fél-átvitel flag
   (d) Parity flag: paritás flag
   (e) Carry flag: átvitel flag
8. SP: Stack Pointer aktuális értéke

A PC mezőbe írjuk bele a címet, ahova helyeztük a kódot, jelen példában a 2000h-t:

Ha be kell állítani a regiszterek kezdőértéket, akkor a megfelelő regiszterek mezőjébe írjuk bele a szükséges értéket.

3.1.3 3. lépés: kód követése

A bal oldali ablakban látható, hogy a PC a 2000h-ra mutat, ezt a kék háttérrel jelöli a rendszer (soron következő utasítás).
Ha rámegyünk a Step gombra az oldal tetején, akkor lefuttatja a PC által jelölt utasítást és az oldal alján elhelyezkedő táblázatban megjelenik néhány adat:

Address	Data	$\overline{RD}$	$\overline{WR}$	IO/$\overline{M}$	A	B	C	D	E	H	L	F [SZ-A-P-C]	SP
2000	21	0	1	0	00	00	00	00	00	00	00	00000000	0000
2001	00	0	1	0	00	00	00	00	00	00	00	00000000	0000
2002	21	0	1	0	00	00	00	00	00	00	00	00000000	0000

Vegyük sorra, hogy mi minden történt:

1. A bal oldali fehér ablakban a kék háttér most már a soron következő utasításra mutat (MVI M, 11), hiszen az előző utasítást már lefuttattuk és tovább lépett.

2. A táblázatban megjelentek az LXI H, 2100h utasítás során történt dolgok.

   (a) Mivel a 2000h-ra helyeztük el a kódot, a PC onnan olvassa fel az első adatot. Ott egy LXI utasítás van, ezért az első adat, amit felolvas az az utasítás opkódja.

   (b) Ökölszabály: minden utasítás első bájtja az opkódja.

   (c) Opkód: utasítás egyedi azonosító kódja

(d) Az LXI utasításhoz 2 dolog tartozik még: a regiszterpár, ahova adatot mozgatunk és maga az adat, amit mozgatunk. A regiszterpár az opkódba van kódolva, ezt a segédletből ki lehet olvasni. A másik paraméter, hogy milyen adatot akarunk elhelyezni a regiszterpárba. 2 bájtos adatot tudunk egy regiszterpárba tenni, ezért ez a 2 bájt az opkódot követő következő 2 címen helyezkedik el.

(e) Little-endian: kisebb címen kisebb helyiérték

(f) a 2100h számot szeretnénk a HL regiszterekbe tenni, szóval először felolvassuk a 2100h alsó bájtját (00h) a 2001h címről, majd a felső bájtját (21h) a 2002h címről.

(g) Egy utasítás opkódja és a paraméterei címfolytonosan helyekednek el a memóriában

(h) Címfolytonos: egymást követő memóriacímeken.

(i) Az opkód és az utasítás paramétereinek beolvasása mind olvasás művelet.

3. A Step-re kattintva lefuttatja a jelenlegi utasítást, ami az MVI M, 11h, majd megjelennek a táblázatban az eközben történtek:

   (a) Címfolytosan helyezkedik el a kód, szóval a következő címen van az MVI M utasítás opkódja, vagyis a 2003h-n.

   (b) Az MVI M utasításhoz tartozik egy paraméter: milyen adatot akarunk elhelyezni az M által mutatott memóriacímre. Ezt az adatot az opkódot követő címről olvassa fel, tehát a 2004h-ról.

   (c) Felolvastunk az MVI M, 11h-hoz tartozó minden adatot, szóval el tudjuk ténylegesen végezni az utasítást fizikailag is: elhelyezzük az M által mutatott memóriacímre a 11h-t.

   (d) Az M pointer a HL regiszterpár tartalmával jelölt memóriacímre mutat, a HL-ben jelenleg a 2100h van, hiszen az imént tettük bele. Így a 2100h címre írjuk a 11h-t. Ez látszik a táblázatban is.

   (e) Figyeljük meg, hogy a $\overline{WR} = 0$, mert ez egy adat írása egy adott memóriacímre. Az Address oszlopban a 2100h szerepel, hiszen erre a címre írunk adatot. A Data oszlopban a 11h érték szerepel, mert ezt az adatot írjuk oda.

   (f) Látható, hogy a H és L oszlopokban a regiszterpár tartalma az előbb odahelyezett 21h és 00h.

4. Ismételten a Step-re nyomva lefut a HLT utasítás is. A HLT címfolytosan helyezkedik el (hiszen nem volt például ORG direktíva vagy hasonló), tehát a 2005h-n. Az adat csupán az utasítás opkódja, mert ehhez az utasításhoz nem tartozik semmilyen paraméter sem.

5. Végeztünk, elvileg most megtanultuk használni a szimulátort. Ezek után sok gyakorlással ezekkel az alapokkal már ügyesen fogunk tudni bánni ezzel a hasznos kis segédeszközzel.

3.2 Utasísátok

3.2.1 CALL

A CALL utasítás egy feltétel nélküli szubrutinhívás.
A legfontosabb ismeretek a CALL utasítás használatához:

1. Kimenti a stackre a CALL utáni utasítás címét, vagyis eltárolja a szubrutin visszatérési címét, ahonnan folytatni kell a szubrutinból való visszatérés után.
2. A stack müvelet kapcsán csökkenti az SP tartalmát kettővel (hiszen 2 bájtot írtunk a stackre)
3. A PC-t átállítja z utasítás 2. és 3. bájtjában kapott címre, vagyis a szubrutin címétől folytatja a program végrehajtását
4. 5 gépi ciklus: opkód olvasás (1 ciklus), szubrutin címének felolvasása (2 ciklus), írás a stackre (2 ciklus)

Nézzük egy példát erre. Tételezzük fel, hogy az SP értéke 9100h.

```
; SP erteke: 9100h
ORG 1200h ; innentol helyezzuk el a kodot
CALL SZUBRUTIN ; szubrutinhivas
...
ORG 5000h ; szubrutin kezdocime
SZUBRUTIN:
IN 12h ; elso utasitas a szubrutinban
...
RET ; visszateres a szubrutinbol
...
```

A program futása során a következő adatok lesznek a memória- és a címsínen:

Cím (hexa)	Adat (hexa)	Írás / olvasás	Megjegyzés
1200	CD	olvasás	CALL opkódjának olvasása
1201	00	olvasás	szubrutin címének olvasása (cím alsó bájtja, hiszen little-endiar
1202	50	olvasás	szubrutin címének olvasása (cím felső bájtja, hiszen little-endia
90FF	12	írás	visszatérési cím felső bájtját írja a stackre
90FE	03	írás	visszatérési cím alsó bájtját írja a stackre
5000	DB	olvasás	IN opkódjának olvasása
...	...	...	...

3.3 A stack (verem) működése

A stack (verem) egy ideiglenes tároló, ahova regiszterpárok értékeit, szubrutinok visszatérési címeit tudjuk többet között eltárolni.
A stack tetejére a Stack Pointer (SP) mutat, ami egy speciális regiszter pár.

A stack-re kizárólag 2 bájtonként lehet adatot írni.
A stack-re írás lépései:

1. A magasabb helyiértékű bájt kerül az SP-1-re
2. Az alacsonyabbik bájt kerül az SP-2-re
3. 2-vel csökken az SP értéke

A stack-et használó néhány fontosabb utasítás:

1. PUSH rp: regiszterpár mentése a stackre
2. POP rp; regiszterpárba visszatöltése a stackről
3. CALL addr: szubrutinhívás, visszatérési cím mentése a stackre
4. RET: szubrutinból visszatérünk az őt meghívó programrészbe, visszatérési cím felolvasása a stackről

3.3.1 Stack Pointer inicializálása

A Stack Pointernek kezdeti értéket például az LXI utasítással adhatunk:

```
LXI SP, 9100h ; SP erteke legyen 9100h
```

3.3.2 Írás a stackre

A stack-re el tudjuk menteni adott regiszterpár értékét, majd azt vissza is tudjuk tölteni onnan.
Gyakori eset, hogy egy szubrutin során használni szeretnénk valamelyik regisztert. Ha a szubrutinnak nem feladata a regiszterek módosítása, tehát nem ott akar például visszatérési értéket visszaadni, csak egy ideiglenes tárolóra van szükségünk, akkor a következő a teendő:

1. Használni kívánt regiszterpár kimentése a stackre (PUSH utasítás)
2. Regisztertár használata, mint ideiglenes tároló
3. Használat után a regiszterpár eredeti értékének visszatöltése

Nézzük erre egy példát:

```
FOPROGRAM:
...
LXI SP, 9100h ; SP erteke legyen 9100h
...
CALL SZUBRUTIN ; szubrutin hivasa a foprogrambol
...
SZUBRUTIN:
PUSH D ; D regiszterpar elmentese a stack-re
```

```
LXI D, 1234h ; D regiszterpar hasznalata
...
POP D ; hasznalat utan az eredeti ertek visszatoltese
RET ; visszateres a foprogramba
```

Fontos, hogy POP-nál a megfelelő helyre mutasson az SP, különben nem a D eredeti értékét tölti vissza!

3.3.3 Olvasás a stackről

A stackről 2 bájtonként lehet az adatokat visszatölteni.
A stackről olvasás folyamata:

1. Felolvassuk az alacsonyabbik bájtot az SP által mutatott címről
2. Felolvassuk a magasabbik helyiértékű bájtot az SP+1 címről
3. Megnöveljük 2-vel az SP értékét

3.3.4 Példa a stack műveletekre

Vizsgáljuk meg az alábbi kódot.

```
FOPROGRAM:
...
LXI SP, 9100h ; SP erteke legyen 9100h
LXI D, 1234h ; DE regiszterpar erteke legyen 1234h
...
ORG 1000h ; innentol helyezzuk el a kodot
CALL SZUBRUTIN ; szubrutin hivasa a foprogrambol
...
ORG 2000h ; innentol helyezzuk el a kodot
SZUBRUTIN:
PUSH D ; D regiszterpar elmentese a stack-re
...
ORG 2010h ; innentol helyezzuk el a kodot
POP D ; hasznalat utan az eredeti ertek visszatoltese a stack-rol
RET ; visszateres a foprogramba
```

A program futása során a következő adatok lesznek a memória- és a címsínen:

Cím (hexa)	Adat (hexa)	Írás / olvasás	Megjegyzés
1000	CD	olvasás	CALL opkódjának olvasása
1001	00	olvasás	szubrutin címének olvasása (cím alsó bájtja, hiszen little-endian
1002	20	olvasás	szubrutin címének olvasása (cím felső bájtja, hiszen little-endia
90FF	10	írás	visszatérési cím felső bájtját írja a stackre
90FE	03	írás	visszatérési cím alsó bájtját írja a stackre
2000	D5	olvasás	PUSH D opkódjának olvasása
90FD	12	írás	D regiszter tartalmat menti az SP-1-re
90FC	34	írás	E regiszter tartalmat menti az SP-2-re
...	...	...	...
2010	D1	olvasás	POP D opkódjának olvasása, visszatoltes a DE-be
90FC	34	írás	E regiszterbe visszatolti az SP-2-n levo adatot
90FD	12	írás	D regiszterbe visszatolti az SP-1-n levo adatot
2011	C9	olvasás	RET opkódjának olvasása
90FE	03	olvasás	visszatérési cím alsó bájtját olvassa a stackről
90FF	10	olvasás	visszatérési cím felső bájtját olvassa a stackről
...	...	...	...

3.4 Példa feladat az ellenőrző kérdések közül

Tipikus vizsga példa a fordítói direktivák számonkérésére. Ennek egy rövidebb változata beugróban is gyakran szerepel. Az értelmezend kód:

```
ADAT0 EQU 14
ORG 800h
ADAT1: DB ADAT0, 11100111b
ORG 5678h
ADAT2: DS 2
ADAT3: DW 2017h
ADAT4: DW ADAT2
ADAT5: DB 314h
```

A megoldás:

Cím (hexa)	Adat (hexa)
0800	0E
0801	E7
5678	X
5679	X
567A	17
567B	20
567C	78
567D	56
567E	14

Ez mind szép és jó, na de miért ez a megoldás? Vegyük sorra, hogy mi és miért így szerepel a megoldásban:

- Az EQU fordítói direktíva egy konstans számértéket deklarál, amit az ADAT0 névvel érhetünk el. A konstans szám mögött nem szerepel számrendszert meghatározó betű, ezért alapértelmezettként decimális számrendszerben értelmezzük. Mivel a táblázat hexában kéri az adatokat, ezért a 14 decimális számot át kell váltani hexadecimális számrendszerbe, ez ugyebár az Eh lesz. Mivel bájtos szervezésű a memória, ezért ki kell egészíteni a félbájtos Eh számot bájtosra, ami egy nullás hexa számjegy eléírását jelenti.
- Egy bináris szám elejére tetszőleges számú nullát odaírva nem változik a szám értéke.
- Az ORG fordítói direktíva azt jelenti, hogy a fordító milyen memóriacímre kezdje el elhelyezni az ORG-ot követő utasításokat a memóriában. Az ORG 800h azt jelenti tehát, hogy 0800h-tól kezdi el a kódokat elhelyezni a memóriában.
- A 8085-nek 16 adatvezetéke van, ami azt jelenti, hogy egy memóriacímet 4 hexa számjeggyel írunk le.
- Az ORG után különböző adatokat definálunk többféle módon a fordító segítségével.
  - DS (Define Space, inicializálatlan helyfoglalás): helyet foglal, de nem tölti fel tartalommal
  - DB (Define Byte, inicializált 1 bájtos helyfoglalás): 1 bájtnyi helyet foglal és feltölti tartalommal
  - DW (Define Word, inicializált 2 bájtos helyfoglalás): 2 bájtnyi helyet foglal és feltölti tartalommal
- Az ORG után először DB szerepel, vagyis a 0800h címtől el fogunk helyezni 2 darab 1 bájtos adatot, mert a DB után 2 érték szerepel.
- Először a DB ADAT0 jelentését vizsgáljuk meg. Az ADAT0 konstant érték az E0h, ez pont 1 bájt, vagyis a DB trükkök nélkül el tudja helyezni a 0800h címre.
- a DB utáni második paraméter a következő címre (mert nem írtunk semmit, ami ezen változtatna) helyezi el az 1110111b-t. Hexába ezt át kell váltani: E7h.
- Még egy ORG jön, vagyis most máshova fogjuk már folytatni a kód elhelyezését, még pedig az 5678h címtől.
- A DS 2 jelentése, hogy 2 bájtnyi adatot fogunk inicializálás nélkül elhelyezni.
- A DS után szereplő számnak megfelelő bájtot foglalunk le iniciaizálás nélkül.
- a DW mivel 2 bájtot helyez el, ezért 2 memóriacímre fog adatot tenni. Címfolytosan csináljuk ezt, mert nem volt újabb ORG vagy ilyesmi, ami ezen változtatna.

- A 2017h-t kell elhelyezni a memóriában. Ez könnyű, hiszen ez pont 2 bájt, vagyis semmi trükk. Mivel little-endian a processzor, ezért a kisebb címre az alsó bájtot tesszük, a nagyobb címre a felső bájtot. Vagyis a 17h-t az 567Ah-ra, a 20h-t az 567Bh-ra.
- Vigyázzatok, szinte minden vizsgában kell a 9h számot eggyel növelni, ami ugyebár hexában az Ah!
- A következő DW-nek az ADAT2h-nek megfelelő értéket kell a memóriába tennie. Ez már picit bonyolult. Az ORG 5678h után szerepel az ADAT2 címke, tehát az ADAT2 értéke az a szám, amelyik memóriacímet jelenti ez a címke: 5678h. Ez egy 2 bájtos adat, ezt már el tudjuk helyezni little-endian módon a következő 2 memóriacímre.
- A végére maradt a nyalánkság, mert a DB 314h ismét trükkös. A probléma az, hogy a DB ugyebár 1 bájtnyi adatot tud a memóriába tenni, de a 314h összesen másfél bájton fér el, tehát fél bájtot nem tud betenni a memóriába. Mivel little-endian szervezésű a processzor, ezért az alacsonyabbik bájtot (a másfél bájtos adatból) elhelyezi a megfelelő címre, a maradék részet pedig levágja, ami nem fér el 1 bájton, tehát a felső fél bájtot).