OPOZORILO. Nobena stvar v tej datoteki ni preverjena in je lahko narobe.
(Prav tako je bila datoteka napisana v prostem času.)

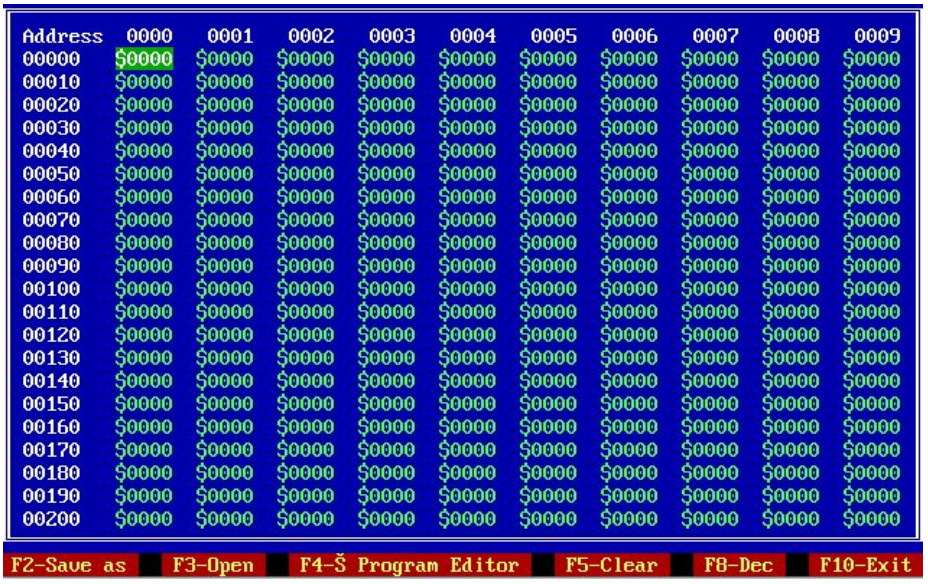
MIC / MAC

1. MIC-1

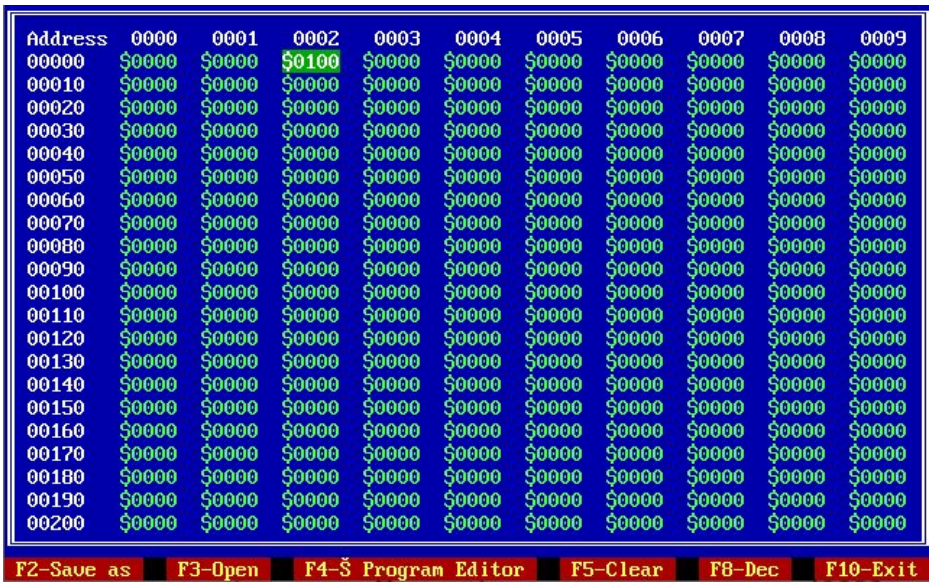
1.1 Horizontalna organizacija

Nalogo se dela z pomočjo aplikacije “dosbox” ter programa “micsim.exe”, ki je na voljo na estudiju.
Za tiste ki še ne znate, je program potem zažene z komando “dosbox micsim.exe” v terminalu.
Nato se pred vami pojavi sledeče okno v katerem boste pisali vso kodo:

Trenutno se nahajamo na sekciji, v kateri lahko pišemo kodo.
Če pa zdaj pritisnemo F4 pridemo do sekcije “RAM editor”, v katerem urejamo vrednosti v pomnilniku.

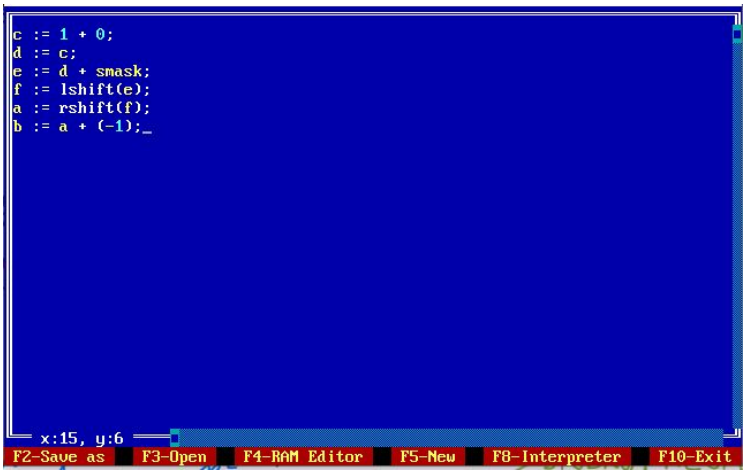


Torej, ko želimo nastaviti pomnilnik na mestu 2 na vrednost 256, se premaknemo na tisti kvadrat, ter na tisto mesto napišemo vrednosti 0100, ker je vrednost treba zapisat v heksadecimalnem zapisu.

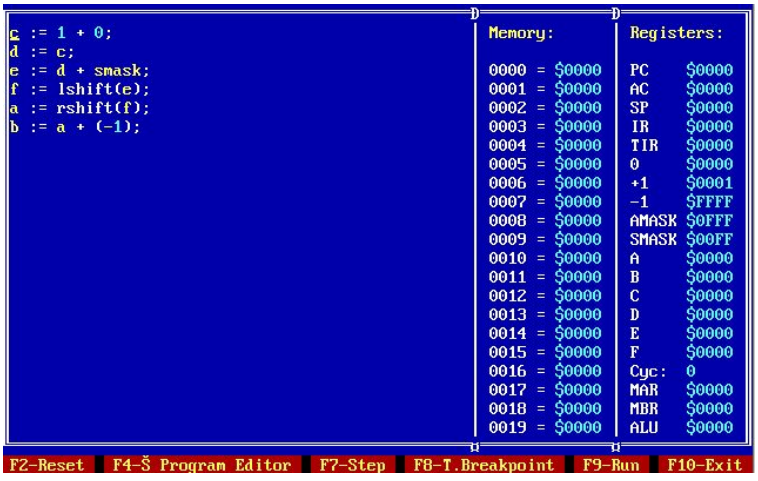


In rezultat je sledeč.

Z F4 se ponovno vrnemo na okno v katerem programiramo.
To je primer kode na katerem bomo pogledali kako stvari delujejo.

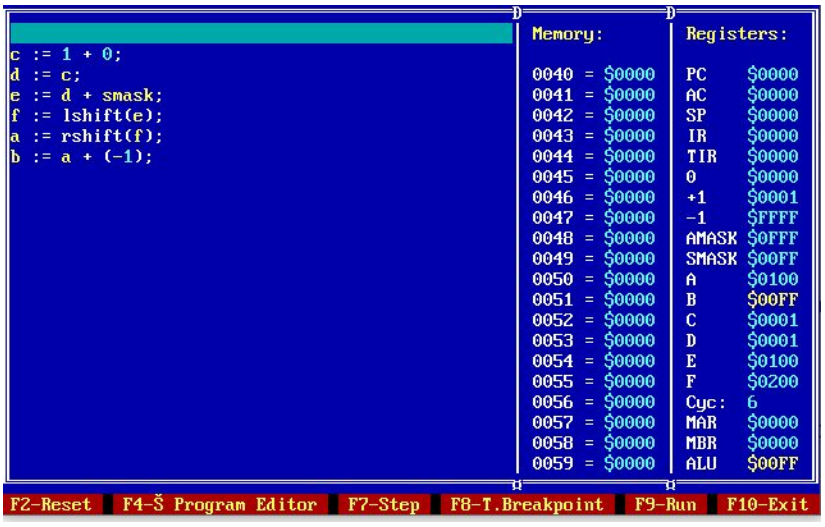


Z ukazom F8 se premaknemo na interpreter.



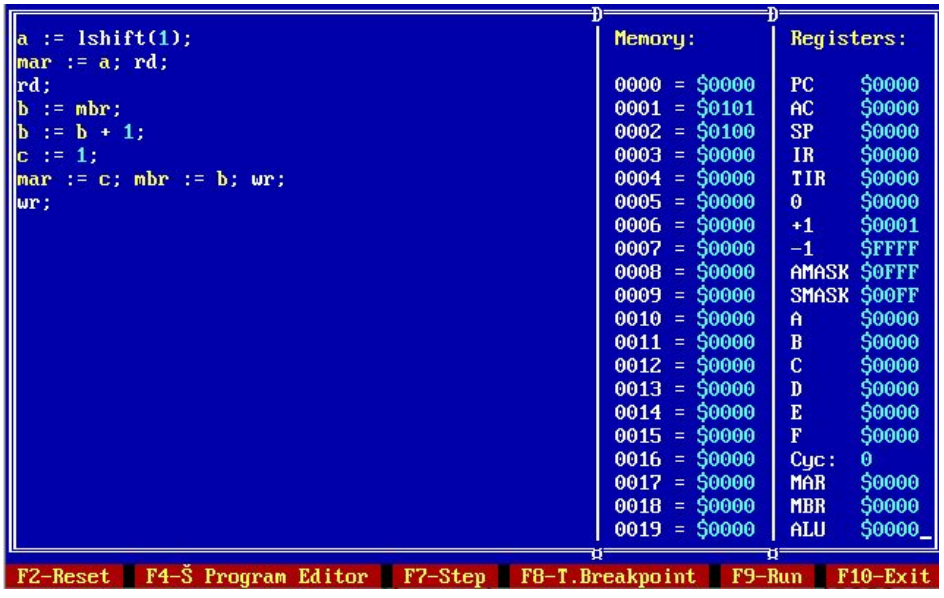
V tem oknu na levi strani vidimo kodo, v sredini je prikazan spomin po katerem se lahko premikamo s puščicama “levo” in “desno”. Na desni strani pa so prikazane vrednosti registrov.

Po programu se premikamo z F7, ter sproti gledamo vrednosti registrov, kjer preverjamo rezultate.



Ko ta program izvedemo dobimo takšne vrednosti registrov.
Register C ima vrednost 1 (vsota registrov “0” in “1”).
Register D ima vrednost 1.
Register E ima vrednost 256 (vsota registrov “d” in “smask”, d=c=1, smask=0x00FF=255).
Register F ima vrednost 512 (binarna operacija << nad registrom E, vsi biti se premaknejo za 1 v levo stran).
Register A ima vrednost 256 (binarna operacija >> nad registrom F, vsi biti se premaknejo za 1 v desno stran, 0x0100=256).
Register B ima vrednost 255 (od registra B smo odšteli vrednost registra “-1”).

Nato pa imamo še 2 posebni operaciji, operaciji branja in pisanja spomina. Poglejmo primer:

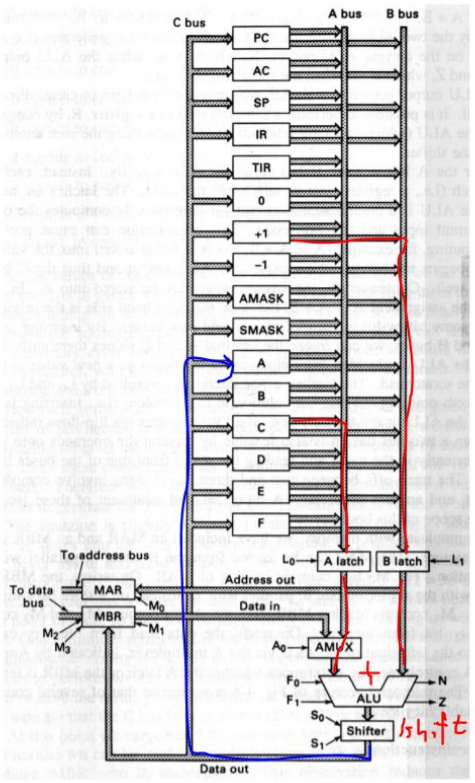


Program prebere vrednost z mesta v spominu 2 (nastavili smo MAR (memory address register)). To vidimo iz registra A, ki je nastavljen na 2 (1<<1=2).
Nato vrednost ki smo jo prebrali shranimo v register B, ter v naslednji vrstici registru B prištejemo 1. Nato na mesto “vrednosti registra A” zapišemo vrednost registra B.

Rezultat tega programa je lahko viden v spominu, originalna vrednost spomina na mestu 2 je bila 256, po koncu programa pa vidim da je v spominu vrednost na mestu 1 enaka 257.

Teorija

Tukaj je nepopolna razlaga, ki je v nekaterih delih lahko narobe.
Preden pa se lahko sploh lotimo te naloge, je potrebno razumeti kaj te operacije sploh delajo.
Primer: poglejmo operacijo "**a := lshift(c + 1);**"



Na začetku je potrebno gledat rdeči puščici iz registrov "1" in C, vrednosti teh dveh registrov gresta po A in B busu. Do dela "ALU", v katerem se izvede operacija "+", za tem pa se izvede operacija "lshift". Za tem se rezultat shrani nazaj v register A preko C busa.

Kako pa bi to inštrukcijo napisali v heksadecimalnem/binarnem zapisu?
0b00000100000110101100011 0xxxxxxxx = 0x041AC6XX

AMUX	COND	ALU	SH	MBR, MAR, RD, WR, ENC
0 = A latch 1 = MBR	0 = No jump 1 = Jump if N = 1 2 = Jump if Z = 1 3 = Jump always	0 = A + B 1 = A AND B 2 = Ā 3 = Ā	0 = No shift 1 = Shift right 1 bit 2 = Shift left 1 bit 3 = (not used)	0 = No 1 = Yes

Za primer "**a := lshift(c + 1);**"

amux	cond	alu	sh	mbr	mar	rd	wr	enc	C	A	B	address
0	0	00	10	0	0	0	0	1	1010	1100	0110	x

alu = 0, ker imamo operacijo "+" (glej spodnjo sliko)
sh = 2, ker imamo operacijo "lshift"
enc = 1, ker vrednost ki smo izračunali shranimo. (enc ~ enable C(C kot del inštrukcije, ne kot register C).
C = 10, v C imamo zaporedno številko registra, v katerega bomo shranili vrednost. V našem primeru je to register A.
A = 12, zaporedna številka registra, ki gre po A-busu, v našem primeru register C.
B = 6, zaporedna številka registra, ki gre po B-busu, v našem primeru register "1".

Zdaj ko smo pogledali primer se lahko lotimo naloge.
Psevdokoda naloge je podobna temu:

```
v0 = 512;                                   // 1
v1 = memory[v0];                            // 2
v2 = 520;                                   // 3
v3 = 0;                                     // 4
do {                                        // 5
        v3 = v3 + memory[v2];               // 6
        v2 = v2 + 1;                        // 7
        v1 = v1 - 1;                        // 8
} while(v1 > 0);                            // 9

v4 = 256;
memory[v4] = v3;
```

Torej če gremo lepo po vrsti, v vrstici 1 je treba v enega izmed registrov zapisati vrednost 512. To lahko naredimo na veliko različnih načinov. Jaz bom uporabil register A ter ga nastavil na vrednost 512.

Način 1:
A := 1 + 1; // A = 2
A := A + A; // A = 4
A := lshift(A + A); // A = 16
A := lshift(A + A); // A = 64
A := lshift(A + A); // A = 256
A := A + A; // A = 512
(OPOZORILO: ne uporabiti tega ali pa bo asistent podelil veliko plagiatov.)

Način 2:
Isto stvar ki je napisana v načinu 1 se da rešit v 1 sami vrstici. (namig: uporabi smask)

…

Način 999+:
Probajte sami napisat na svoj način.

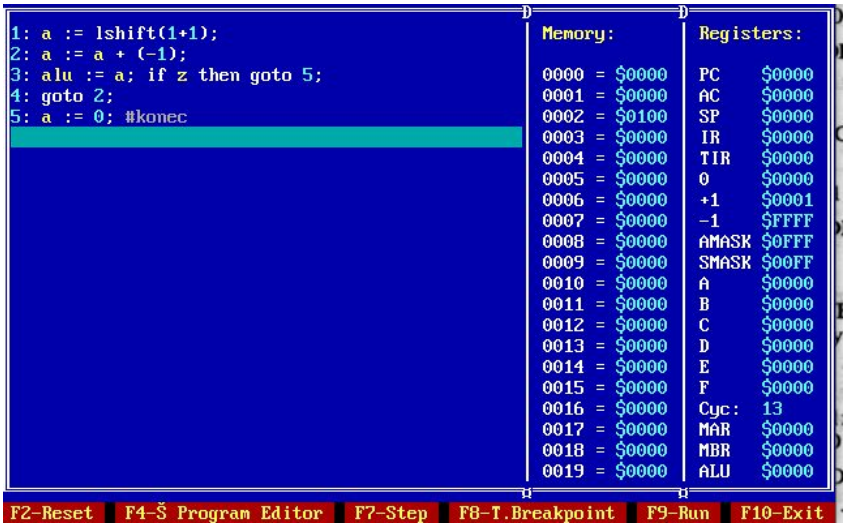
V vrstica 2 je treba prebrat vrednost iz mesta 512.

```
mar := a; rd;
rd;
b := mbr;
```

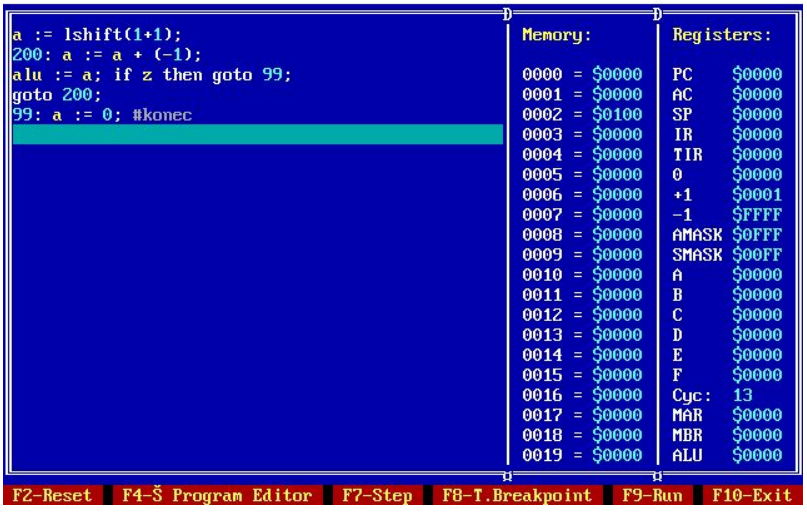
Zakaj je 2 krat napisan ukaz “rd;” in zakaj v 2 vrsticah probajte ugotoviti sami. Lahko pa samo zaupate ter uporabite to kar je napisano.

Vrstici 3 in 4 bi morali znati rešiti sami glede na primere ki smo jih že rešili.

Nato pa imamo zanko, ki jo implementiramo z pomočjo “if” ter “goto” stavka.



Ta program ima na začetku vrednost registra “A = 4”. Nato pa sledi “for” zanka, v kateri pogledamo če je vrednost registra A še manjša ali enaka 0 (if stavek v vrstici 3. v programu), če pogoj drži skočimo na 5(then goto 5), v nasprotnem primeru pa se izvede koda v 4. vrstici ki pravi “goto 2”, ki pa program izvajanje pomakne v 2. vrstico.
Na začetku vsake vrstice pa vidite zaporedno številko z dvopičjem, tega ni treba pisati vedno, lahko si poljubno izmislite sami.
Na primer tole je tudi možno.



Na koncu pa še samo vrednost vsote števil, ki ste jo izračunali, shranite v spomin na mesto 256.
Koda:
 # predpostavljam da boste imeli shranjeni vrednosti 256 in vsoto v 2 registrih
A = 256;
B = vsota;
mar := A; mbr := B; wr;
wr;

1.2. Vertikalna organizacija

Če primerjate izgled kode v pdfu Model_HO in Model_VO, boste videli zelo podobno kodo.

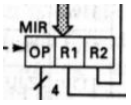
Horizontalna

```
0: mar := pc ; rd ;
1: pc := pc + 1; rd ;
2: ir := mbr ; if n then goto 28;
3: tir := lshift (ir + ir ); if n then goto 19;
4: tir := lshift (tir ); if n then goto 11;
5: alu := tir ; if n then goto 9;
6: mar := ir ; rd ;
7: rd ;
8: ac := mbr ; goto 0;
9: mar := ir ; mbr := ac ; wr ;
10: wr ; goto 0;
11: alu := tir ; if n then goto 15;
12: mar := ir ; rd ;
13: rd ;
14: ac := mbr + ac ; goto 0;
15: mar := ir ; rd ;
16: ac := ac + 1; rd ;
17: a := inv (mbr );
18: ac := ac + a ; goto 0;
19: tir := lshift (tir ); if n then goto 25;
20: alu := tir ; if n then goto 23;
21: alu := ac ; if n then goto 0;
22: pc := band (ir , amask ); goto 0;
23: alu := ac ; if z then goto 22;
24: goto 0;
25: alu := tir ; if n then goto 27;
26: pc := band (ir , amask ); goto 0;
27: ac := band (ir , amask ); goto 0;
28: tir := lshift (ir + ir ); if n then goto 40;
29: tir := lshift (tir ); if n then goto 35;
30: alu := tir ; if n then goto 33;
31: a := ir + sp ;
32: mar := a ; rd ; goto 7;
```

Če primerjamo kodo, vidimo da je koda “enaka”, edina razlika je v tem, da vertikalna organizacija ne omogoča, da imamo več stvari v isti vrstici.

Na primer vrstici 8:
v horizontalni je lahko napisano “ac=mbr;goto 0”,
v vertikalni pa to ni mogoče, zato vidimo da sta vrstici zapisani v različnih vrsticah.

To pa se zgodi, ker so inštrukcije v vertikalni orientaciji veliko samo 12 bitov. 4 biti za operacijsko kodo, potem pa 2 krat po 4 biti za zaporedni številki posameznih registrov.



Vertikalna

```
0:  mar := pc ; rd ;
1:  rd ;
    pc := pc + 1;
2:  ir := mbr ;
    tir := lshift (ir );
    if n then goto 28;
3:  tir := lshift (tir );
    if n then goto 19;
4:  tir := lshift (tir );
    if n then goto 11;
5:  alu := tir ;
    if n then goto 09;
6:  mar := ir ; rd ; {LODD}
7:  rd ;
8:  ac := mbr ;
    goto 0;
9:  mar := ir ; mbr := ac ; wr ; {STOD}
10: wr ;
    goto 0;
11: alu := tir ;
    if n then goto 15;
12: mar := ir ; rd ; {ADDD}
13: rd ;
14: a := mbr ;
    ac := ac + a ;
    goto 0;
15: mar := ir ; rd ; {SUBD}
16: rd ;
99: ac := ac + 1;
17: a := mbr ;
    a := inv (a );
18: ac := ac + a ;
    goto 0;
19: tir := lshift (tir );
    if n then goto 25;
20: alu := tir ;
    if n then goto 23;
21: alu := ac ; {JPOS}
    if n then goto 0;
22: pc := ir ;
    pc := band (pc , amask );
    goto 0;
```

Torej kako rešite nalogo z vertikalno organizacijo.
Samo vzamete svojo kodo z horizontalno orientacijo, spremenite vrednosti od kot začnete branje števil na 516 in kam shranite končno vrednost na 508.
Nato pa še samo spremenite kodo na mestih, kjer je to potrebno:

- pred “if” in “goto” stavki
- shift(X+Y) spremenite v “v=X+Y; shift(v);” (iz 1 operacije v 2)

2. MAC

Torej med inštrukcije je treba dodat operacijo “mul”.

Kako se lotiti te naloge?
Prvo je potrebno analizirati testni program.

2. 1. “Analiza”

```
LOCO 4          # v register AC smo naložili vrednost 4
SWAP            # zamenjali smo vrednosti registrov SP in AC
LODL 30         # v register AC smo naložili memory[SP+30]
MUL 6           # to želimo naredit, naj bi pomnožilo vrednost AC z 6
STOL 30         # na memory[SP+30] se shrani vrednost registra AC
DESP 1          # vrednost registra SP se zmanjša za 1
SWAP            # zamenjamo vrednosti registrov AC in SP
JZER 10         # če je AC == 0, skočimo v 10 vrstico
SWAP            # menjamo AC in SP
JUMP 2          # skočimo v drugo vrstico
```

Iz tega je nekako razvidno da želimo vrednosti v spominu od mesta 31 do 34 pomnožit z vrednostjo 6 ter jo shranit na enako mesto v spominu.

Preden nadaljujemo z naslednjim korakom si poglejmo kako program MAC sploh zaženemo. Za potrebo testa bomo nadomestili operacijo “MUL 6” nadomestili z “ADDD 6” (namesto da bomo vsako vrednost pomnožili z 6, ji bomo prišteli memory[6]”.
V micsim.exe je potrebno naložit datoteko inter.mp, ki je interpreter za MAC. Nato pa moremo inštrukcije pretvoriti v heksadecimalne kode:

```
LOCO 4      = 0x7004
SWAP        = 0xFC00
LODL 30     = 0x801E
…
```

Te inštrukcije potem zapišete v spomin od mesta 0 naprej.

Od kod pa so prišle te heksadecimalne kode?
Preračunali smo jih iz te tabele.

Binary	Mnemonic	Instruction	Meaning
0000xxxxxxxxxxxx	LODD	Load direct	$ac := m[x]$
0001xxxxxxxxxxxx	STOD	Store direct	$m[x] := ac$
0010xxxxxxxxxxxx	ADDD	Add direct	$ac := ac + m[x]$
0011xxxxxxxxxxxx	SUBD	Subtract direct	$ac := ac - m[x]$
0100xxxxxxxxxxxx	JPOS	Jump positive	**if** $ac \geq 0$ **then** $pc := x$
0101xxxxxxxxxxxx	JZER	Jump zero	**if** $ac = 0$ **then** $pc := x$
0110xxxxxxxxxxxx	JUMP	Jump	$pc := x$
0111xxxxxxxxxxxx	LOCO	Load constant	$ac := x\ (0 \leq x \leq 4095)$
1000xxxxxxxxxxxx	LODL	Load local	$ac := m[sp + x]$
1001xxxxxxxxxxxx	STOL	Store local	$m[x + sp] := ac$
1010xxxxxxxxxxxx	ADDL	Add local	$ac := ac + m[sp + x]$
1011xxxxxxxxxxxx	SUBL	Subtract local	$ac := ac - m[sp + x]$
1100xxxxxxxxxxxx	JNEG	Jump negative	**if** $ac < 0$ **then** $pc := x$
1101xxxxxxxxxxxx	JNZE	Jump nonzero	**if** $ac \neq 0$ **then** $pc := x$
1110xxxxxxxxxxxx	CALL	Call procedure	$sp := sp - 1; m[sp] := pc; pc := x$
1111000000000000	PSHI	Push indirect	$sp := sp - 1; m[sp] := m[ac]$
1111001000000000	POPI	Pop indirect	$m[ac] := m[sp]; sp := sp + 1$
1111010000000000	PUSH	Push onto stack	$sp := sp - 1; m[sp] := ac$
1111011000000000	POP	Pop from stack	$ac := m[sp]; sp := sp + 1$
1111100000000000	RETN	Return	$pc := m[sp]; sp := sp + 1$
1111101000000000	SWAP	Swap ac, sp	$tmp := ac; ac := sp; sp := tmp$
11111100yyyyyyyy	INSP	Increment sp	$sp := sp + y\ (0 \leq y \leq 255)$
11111110yyyyyyyy	DESP	Decrement sp	$sp := sp - y\ (0 \leq y \leq 255)$

xxxxxxxxxxxx is a 12-bit machine address; in column 4 it is called x.
yyyyyyyy is an 8-bit constant; in column 4 it is called y.

2. 2. Razširjanje interpreterja

Ko smo končali z analizo moremo preučiti, je potrebno v zgornji tabeli najti prosto operacijsko kodo.
Na primer vidimo da je ena izmed prostih operacijski kod enaka 0b11111101yyyyyyyy.

Ko smo našli kodo je potrebno ustrezno modificirati datoteko inter.mp, tako da bo najprej podpirala operacijsko kodo 0b11111101, nato pa dodamo kodo za operacijo MUL.

Kako bomo dodali operacijsko kodo? Potrebno je ugotoviti kako si sledijo “if” stavki v datoteki “inter.mp”.

V sledečih korakih bom predstavil **enega izmed možnih načinov** spremembe datoteke “inter.mp”.

```
 0: mar := pc ; rd ;                                  {main loop}
 1: pc := pc + 1; rd ;                                {increment pc}
 2: ir := mbr ; if n then goto 28;                    {save, decode mbr}
 3: tir := lshift (ir + ir ); if n then goto 19;
 4: tir := lshift (tir ); if n then goto 11;          {000x or 001x?}
 5: alu := tir ; if n then goto 9;                    {0000 or 0001?}
 6: mar := ir ; rd ;                                  {0000 = LODD}
 7: rd ;
 8: ac := mbr ; goto 0;
 9: mar := ir ; mbr := ac ; wr ;                      {0001 = STOD}
10: wr ; goto 0;
11: alu := tir ; if n then goto 15;                   {0010 or 0011?}
12: mar := ir ; rd ;                                  {0010 = ADDD}
13: rd ;
14: ac := mbr + ac ; goto 0;
```

Začetek programa inter.mp je sledeč. Iz kode razberemo da se v prvih 2 vrsticah izvede branje vrednosti iz spomina na mestu vrednosti registra PC(program counter). Za tem pa se začnejo izvajati if stavki ki preverijo katero operacijsko kodo trenutno imamo.

V kodo želimo dodati operacijsko kodo 0b11111101(bin) oz. 0xFD(hex). To bomo naredili na sledeč način:

```
2: ir := mbr;
100: a := ir;                       # v register A shranimo vrednost celega ukaza
101: b := band(a,smask);            # z operacijo band izluščimo spodnjih 8 bitov
102: c := inv(b);                   # v naslednjih 3 operacijah izvedemo A-B na način
103: a := a + c;                    # A - B = A + 1 + not B
104: a := a + 1;
{
tukaj dodate 5 vrstic kode          # skratka vrednost registra D more bit enaka 0xFD00
}
110: d := inv(d);
111: a := a + 1;
112: a := a + d; if n then go to 199; # preverili smo če je ukaz enak naši operacijski kodi
                                    # v naslednjih vrsticah napišete kodo za operacijo MUL
                                    # ki pa je v bistvu samo 1 for zanka
```

113: # v registru B je vrednost 8 bitnega števila
114: # v registru AC je vrednost s katero jo motere zmnožiti
{
tukaj dodate 7 vrstic kode
}

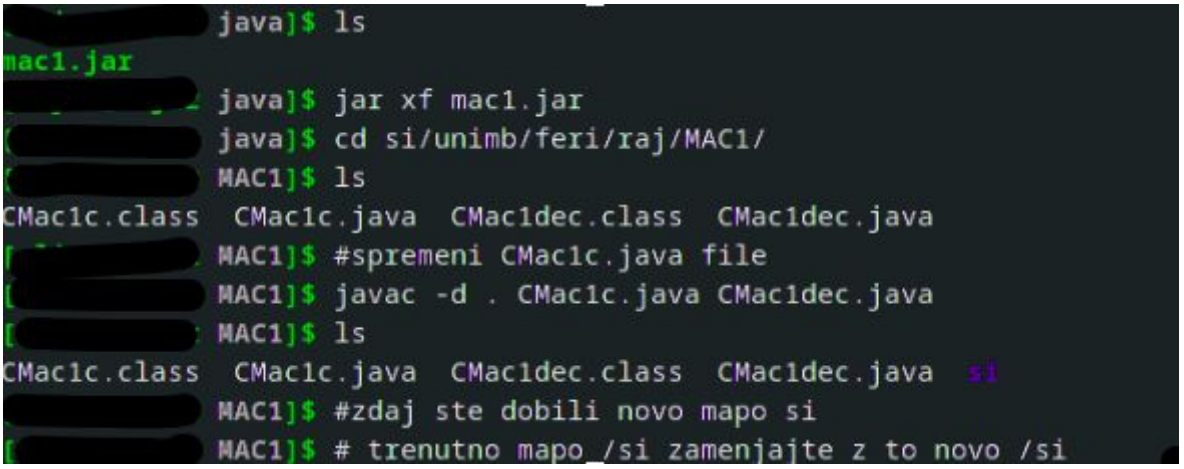
198: goto 0;
199: alu := ir; if n then goto 28; # če ukaza nismo našli se vrnemo nazaj na program

3: …
4: …. # program se normalno nadaljuje

Zakaj so številke pred kodo med 100 in 199? Lahko so poljubne, razen vrstice od 0 do 78 so že zasedene.

2. 3. Spreminjanje MAC prevajalnika

Ko ste spremenili inter.mp datoteko je potrebno še popraviti še CMac1.java file.



To je postopek v cmd.

Verzija 0 - nepopravljeno

V CMac1.java spremenite 3 stvari:

```
// simbolne tabele (mnemoniki)
protected final static String strMnemoniki[] =
    {
        "LODD",
        "STOD",
        "ADDD",
        "SUBD",
        "JPOS",
        "JZER",
        "JUMP",
        "LOCO",
        "LODL",
        "STOL",
        "ADDL",
        "SUBL",
        "JNEG",
        "JNZE",
        "CALL",
        "PSHI",
        "POPI",
        "PUSH",
        "POP",
        "RETN",
        "SWAP",
        "INSP",
        "DESP",
        "END",
        "MUL"   // MUL
        };
```

```
// maske argumentov
protected final static int iArgumentMaske[] =
    {
        0x0FFF,
        0x0FFF,
        0x0FFF,
        0x0FFF,
        0x0FFF,
        0x0FFF,
        0x0FFF,
        0x0FFF,
        0x0FFF,
        0x0FFF,
        0x0FFF,
        0x0FFF,
        0x0FFF,
        0x0FFF,
        0x0FFF,
        0x0FFF,
        0x0000,
        0x0000,
        0x0000,
        0x0000,
        0x0000,
        0x0000,
        0x00FF,
        0x00FF,
        0x0000,
        0x00FF // MUL
        };
```

```
// operacijska koda za mnemonike
protected final static int iOperacijskeKode[] =
    {
        0x0000,
        0x1000,
        0x2000,
        0x3000,
        0x4000,
        0x5000,
        0x6000,
        0x7000,
        0x8000,
        0x9000,
        0xA000,
        0xB000,
        0xC000,
        0xD000,
        0xE000,
        0xF000,
        0xF200,
        0xF400,
        0xF600,
        0xF800,
        0xFA00,
        0xFC00,
        0xFE00,
        0xFFFF,
        0xFD00 // MUL
        };
```

Dodali smo string ime operacije, kateri del inštrukcije predstavlja argument, ter kakšna je operacijska koda.

Za tem je potrebno mape že spet prevesti v jar file, kar pa se naredi ukazom (v cmd):
jar cvf IME.jar *

Ko smo modificirali interpreter(inter.mp) ter dopolnili prevajalnik, program izvedemo na sledeč način.
Prvo v “IME.mac” damo testno kodo ki je na eštudiju. Nato izvedemo spremenjeno jar datoteko ter kot prvi argument dodamo “IME.mac”.

java -jar mac1.jar
Premalo argumentov (Podajte vhodno in izhodno datoteko)!
Uporaba:java asm <vhod> <izhod> [-v]

V micsim.exe v RAM editorju naložimo izhodno datoteko iz jar programa, v program editor pa naložimo spremenjeno inter.mp datoteko.

3. Zaključek

Ta dokument je namenjen za osvežitev znanja in samo predlaga način reševanja. Rešitve so samo ene izmed možnih načinov reševanja.
Ne priporočam kopiranje iz dokumenta, saj obstaja verjetnost da bo več ljudi skopiralo enako stvar.