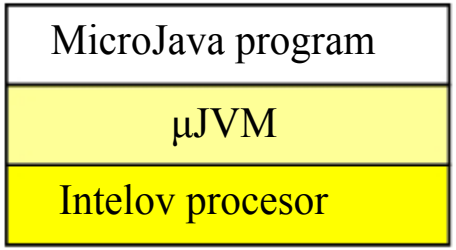
Arhitektura MikroJava virtuelne mašine (μJVM)

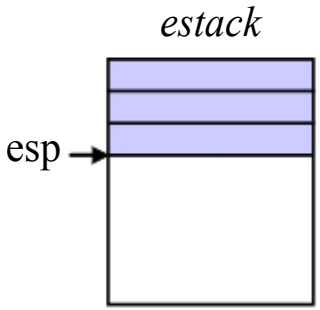
Šta je virtuelna mašina (VM)?

- CPU implementiran u softveru
- instrukcije se interpretiraju (ili "jit-uju")
- primeri: Java VM, Microsoft CLR, Paskalski P-kod



μJVM je stek mašina

- nema registara
- umesto njih ima *stek izraza* (koji čuva vrednosti koje se računaju)



niz mem. reči (1 reč = 4 bajta)
ne mora biti veliki (npr. 32 words ≈ 32 registra)

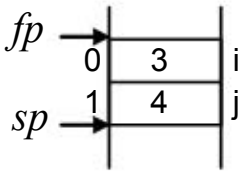
esp ... expression stack pointer

Kako radi Stek Mašina

Primer

iskaz **i = i + j * 5;**

i i *j su npr. lokalne promenljive*



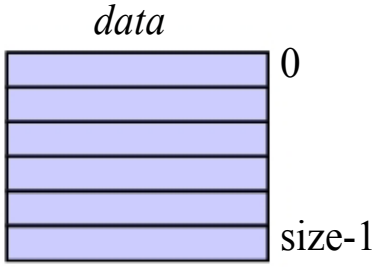
Izvršavanje mjvm programa

maš.instrukc	*e-stek*	
load0	3	load variable from address 0 (i.e. *i*)
load1	3 4	load variable from address 1 (i.e. *j*)
const5	3 4 5	load constant 5
mul	3 20	multiply the two topmost stack elements
add	23	add the two topmost stack elements
store0		store the topmost stack element to address 0

Na kraju svakog iskaza e-stek treba da je prazan!

Oblasti podataka μJVM

Globalni podaci

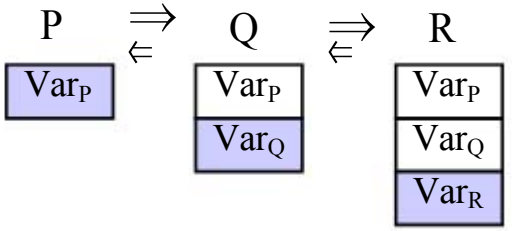


implementirano kao niz reči u VM

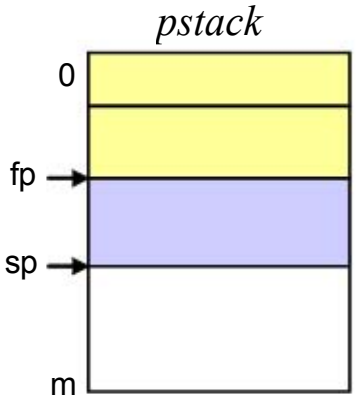
- oblast fiksne veličine
- globalne promenljive postoje tokom čitavog programa
- svaka promenljiva zauzma 1 memorijsku reč (4 bajta)
- adresiraju se uz navođenje indeksa u nizu data
  npr. *getstatic 2* učitava vrednost sa adrese 2 niza *data* na *e-stek*

Oblasti podataka μJVM

Lokalne promenljive



- alociraju se unutar okvira **programskog** steka
- svaki poziv metoda dobija sopstveni okvir na steku
- okvirima se stavljaju na stek pri pozivu, a skidaju sa steka pri povratku iz potprograma



lokalne promenljive P

lokalne promenljive Q

lokalne promenljive tekućeg metoda R

fp ... *frame pointer*

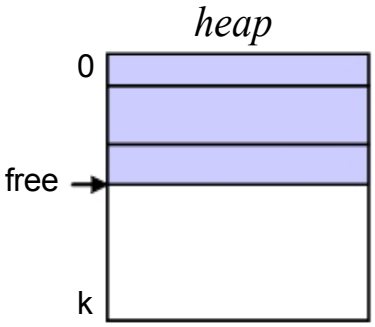
sp ... *stack pointer*

- lokalne promenljive se adresiraju relativno u odnosu na *fp*
- svaka promenljiva zauzima po 1 reč (4 bajta)
- npr. *load0* učitava vrednost promenljive sa adrese *fp+0* na *e-stek*

Oblasti podataka μJVM

Dinamički podaci - Heap

- za čuvanje instanci klasa i nizova



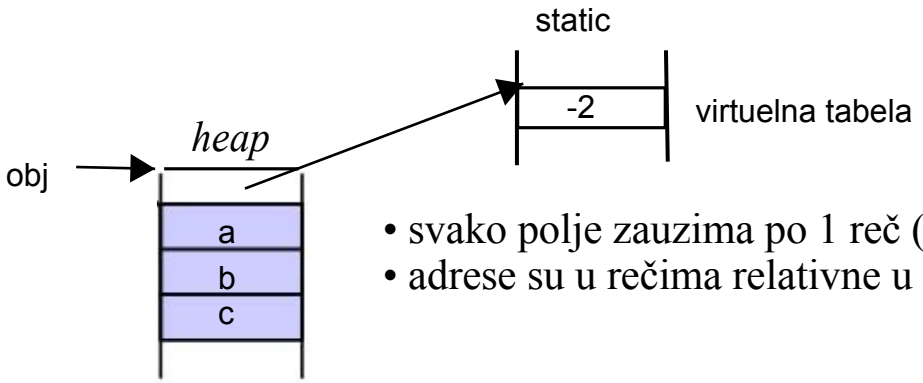
niz reči u VM

- Novi objekti se alociraju na poziciji *free* (pa se *free* uvećava); ovo se radi VM instrukcijama *new* i *newarray*
- Objekti se nikad ne dealociraju (nema “sakupljanja smeća”- garbage collection)
- Pokazivači su adrese na nivou reči

Oblasti podataka μJVM

Objekat klase

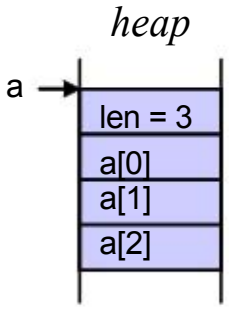
```
class X {
    int a, b;
    char c;
}
X obj = new X;
```



- svako polje zauzima po 1 reč (4 bajta)
- adrese su u rečima relativne u odnosu na *obj*

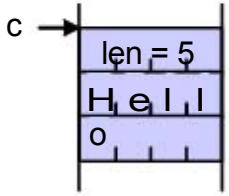
Niz

```
int[] a;
a = new int[3];
```



- dužina niza se pamti na početku
- svaki element zauzima po 1 reč

```
char[] c = new char[5];
```

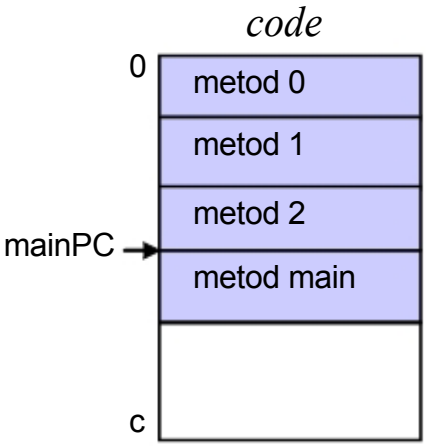


- *char* nizovi su nizovi bajtova
- alociranje prostora je na nivou reči

Oblast koda μJVM

Programski kod

- niz bajtova fiksne veličine
- metodi se alociraju jedan za drugim
- *mainPC* ukazuje na *main()* metod



niz bajtova VM

posebni registeri VM

fp frame pointer
sp stack pointer (pstack)
esp stack pointer (estack)
pc program counter

Skup instrukcija μJVM

Bajtkod (slično bajtkodu Java virtuelne mašine)

- Kompaktan: većina istrukcija od po 1 bajt
- ne pamte se tipovi (kod Java VM tipovi operanada su kodirani u instrukcijama)

MicroJava

```
load0
load1
add
```

Java

```
iload0        fload0
iload1        fload1
iadd          fadd
```

razlog: Java verifikator bajtkoda
može na osnovu tipova operanada proveriti integritet programa

Format μJVMinstrukcija

```
Code =          {Instruction}.
Instruction =   opcode {operand}.
```

opcode ... 1 bajt
operand ... 1, 2 ili 4 bajta

Primeri

0 operanada	`add`	ima 2 implicitna operanda na steku
1 operand	`load 7`	
2 operanda	`enter 0, 2`	početak metoda

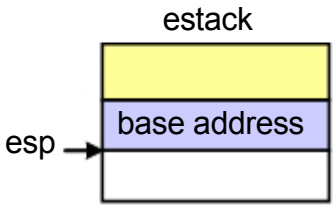
Skup instrukcija μJVM

Načini Adresiranja

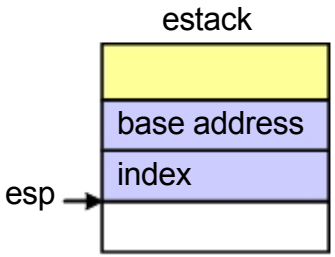
Kako se može pristupiti operandima?

način adresiranja	*primer instrukcije*	
• **Neposredno**	const 7	za konstante
• **Lokalno**	load 3	za lokalne promenljive na *psteku*
• **Statičko**	getstatic 3	za globalne promenljive u *data* oblasti
• **Stek**	add	za vrednosti na *esteku*
• **Relativno**	getfield 3	za polja objekata (load *heap[pop() + 3]*)
• **Indeksno**	aload	za elemente nizova (load *heap[pop() + 1 + pop()]*)

Relativno



Indeksno



Skup instrukcija μJVM

Load/store lokalnih promenljivih

load b	..., val	Load push(local[b]);
load<n>	..., val	Load (n = 0..3) push(local[n]);
store b	..., val	Store local[b] = pop();
store<n>	..., val	Store (n = 0..3) local[n] = pop();

dužine operanada

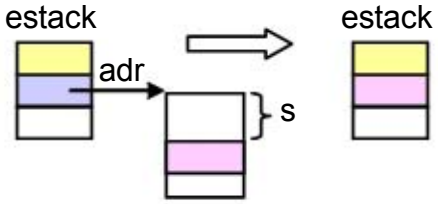
b ... bajt
s ... short (2 bajta)
w ...word (4 bajta)

Load/store globalnih promenljivih

getstatic s	..., val	Load static variable push(data[s]);
putstatic s	..., val	Store static variable data[s] = pop();

Skup instrukcija μJVM

Load/store polja objekata



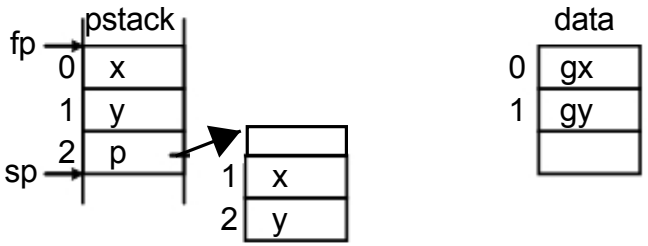
getfield s	..., adr ..., val	Load object field adr = pop(); push(heap[adr+s]);

putfield s	..., adr, val	Store object field
		val = pop(); adr = pop();
		heap[adr+s] = val;

Load konstante

const w	..., val	Load constant push(w);
const<n>	..., val	Load constant (n = 0..5) push(n);
const_m1	..., val	Load minus one push(-1);

Primeri: Load i Store



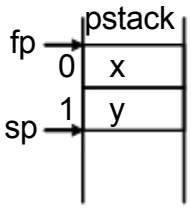
	code	*bytes*	*stack*
x = y;	load1 1		y
	store0 1		-
gx = gy;	getstatic 1	3	gy
	putstatic 0	3	-
p.x = p.y;	load2 1		p
	load2 1		p p
	getfield 2	3	p p.y
	putfield 1	3	-

Skup instrukcija μJVM

Aritmetičke instrukcije

add	..., val1, val2 ..., val1+val2	Add push(pop() + pop());
sub	..., val1, val2 ..., val1-val2	Subtract push(-pop() + pop());
mul	..., val1, val2 ..., val1*val2	Multiply push(pop() * pop());
div	..., val1, val2 ..., val1/val2	Divide x = pop(); push(pop() / x);
rem	..., val1, val2 ..., val1%val2	Remainder x = pop(); push(pop() % x);
neg	..., val ..., -val	Negate push(-pop());
shl	..., val, x ..., val1	Shift left x = pop(); push(pop() << x);
shr	..., val, x ..., val1	Shift right x = pop(); push(pop() >> x);

Primeri: Aritmetičke operacije



	code	*bytes*	*stack*
x + y * 3	load0 1		x
	load1	1	x y
	const3	1	x y 3
	mul	1	x y*3
	add 1		x+y*3

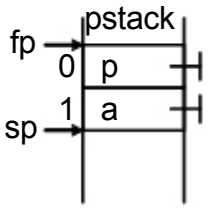
Skup instrukcija μJVM

Kreiranje objekata

new s	..., adr	New object allocate area of s words; initialize area to all 0; push(adr(area));

newarray b	..., n
	..., adr
		New array
		n = pop();
		if (b == 0)
		allocate array with n elements of byte size;
		else if (b == 1)
		allocate array with n elements of word size;
		initialize array to all 0;
		store n as the first word of the array;
		push(adr(array));

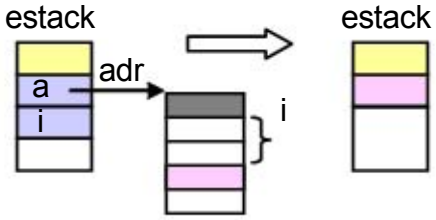
Primeri: Kreiranja objekata



	code	*bytes*	*stack*	
Person p = new Person;	new 4	3	p	// assume: size(Person) = 4 words
	store0 1		-	
int[] a = new int[5];	const5 1		5	
	newarray 1	2	a	
	store1 1		-	

Skup instrukcija μJVM

Pristup nizovima



aload	..., adr, i
	..., val
		Load array element
		i = pop(); adr = pop();
		push(heap[adr+1+i]);

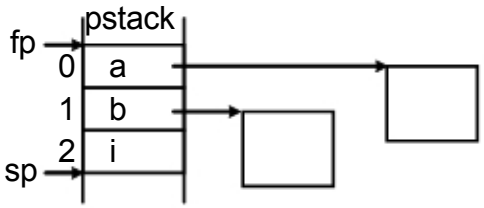
astore	...,adr, i, val	Store array element
		val = pop(); i = pop(); adr = pop();
		heap[adr+1+i] = val;

baload	..., adr, i
	..., val
		Load byte array element
		i = pop(); adr = pop();
		x = heap[adr+1+i/4];
		push(byte i%4 of x);

bastore	...,adr, i, val	Store byte array element
		val = pop(); i = pop(); adr = pop(); x = heap[adr+1+i/4];
		set byte i%4 in x;
		heap[adr+1+i/4] = x;

arraylength	..., adr
	..., len
		Get array length
		adr = pop();
		push(heap[adr]);

Primer: Pristup članu niza



	code	*bytes*	*stack*
a[i] = b[i+1];	load0 1		a
	load2	1	a i
	load1	1	a i b
	load2	1	a i b i
	const1 1		a i b i 1
	add	1	a i b i+1
	aload	1	a i b[i+1]
	astore 1		-

Skup instrukcija μJVM

Manipulacija stekom

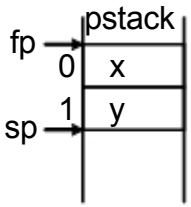
pop	..., val	Remove topmost stack element dummy = pop();

Skokovi

jmp s		Jump unconditionally pc = s;

j<cond> s	..., x, y	Jump conditionally (eq,ne,lt,le,gt,ge)
		y = pop(); x = pop();
		if (x cond y) pc = s;

Primer: Skok



	code	*bytes*	*stack*
if (x > y) ...	load0 1		x
	load1	1	x y
	jle ...	3	-

Skup instrukcija μJVM

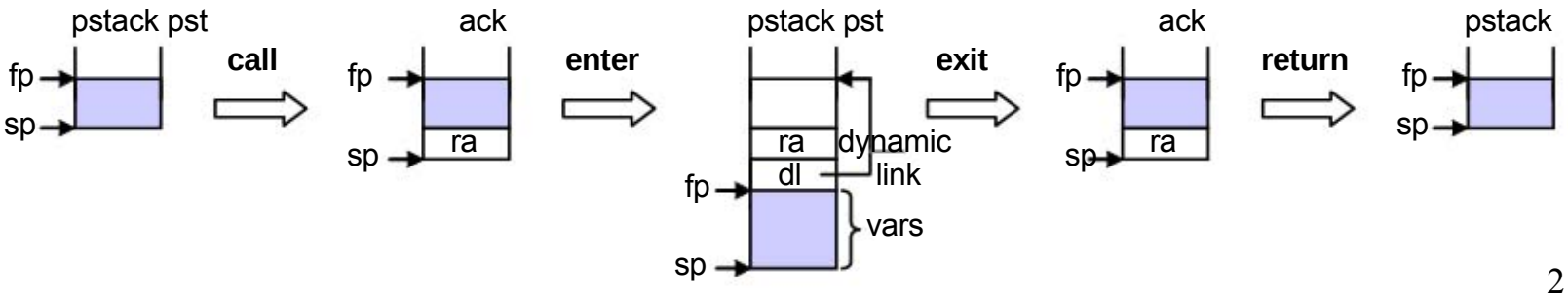
Poziv metoda

call s		Call method PUSH(pc+3); pc = s;

enter b1, b2		Enter method
		pars = b1; vars = b2; // in words
		PUSH(fp); fp = sp; sp = sp + vars;
		initialize frame to 0;
		for (i=pars-1; i>=0; i--) local[i] = pop();

exit		Exit method sp = fp; fp = POP();
return		Return pc = POP();

PUSH i POP rade nad *pstack-om*



Poziv statičke funkcije

Primer

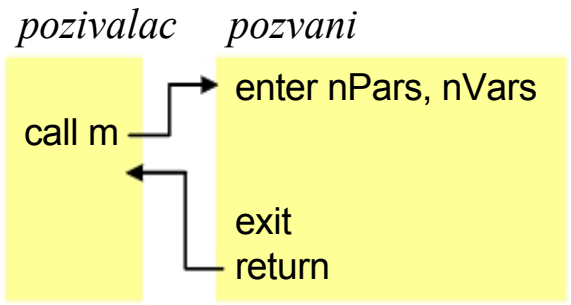
```
c = m(a, b);
```

```
load a
load b
call m
store c
```

parameters are passed on the *estack*

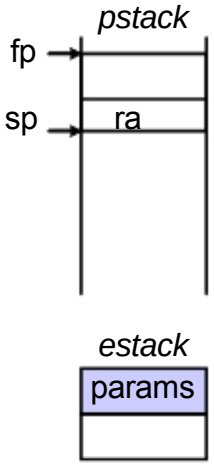
function value is returned on the *estack*

Okviri na programskom steku



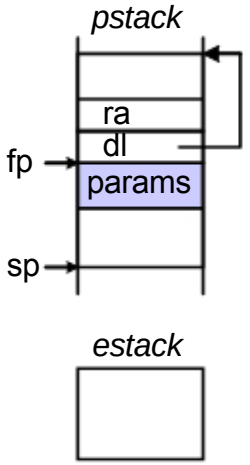
enter ... kreira okvir na psteku
exit ... uklanja okvir sa psteka

Enter instrukcija

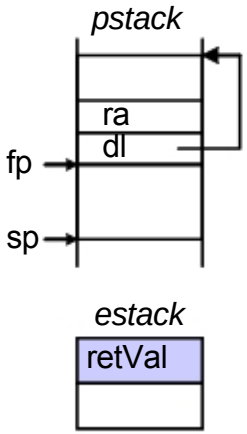


enter nPars, nVars

```
PUSH(fp); // dynamic li   nk
fp = sp;
sp = sp + nVars;
initialize frame to 0;
for (i=nPars; i>=0; i--)
 local[i]    = pop();
```

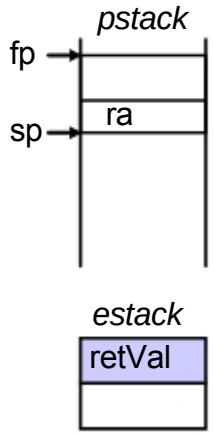


Exit instrukcija



exit

```
sp = fp;
fp = POP();
```



Skup instrukcija μJVM

Ulaz/izlaz

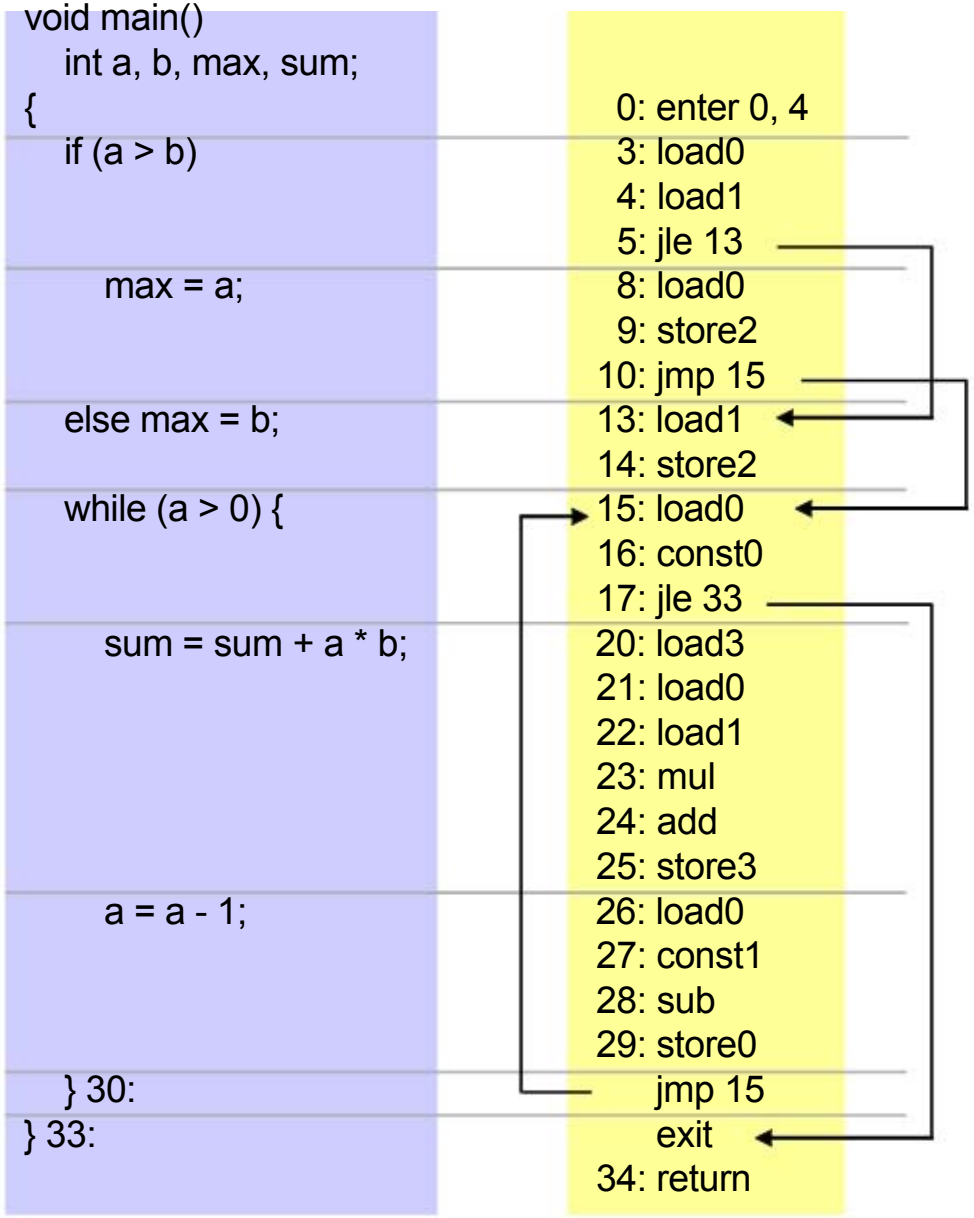
read	..., val	Read x = readInt(); push(x);
print	..., val, width	Print w = pop(); writeInt(pop(), w);
bread	..., val	Read byte ch = readChar(); push(ch);
bprint	..., val, width	Print w = pop(); writeChar(pop(), w);

ulaz sa System.in
izlaz na System.out

Ostalo

trap b		Throw exception print error message b; stop execution;

Primer prevođenja statickog metoda



adrese

a ... 0
b ... 1
max ...2
sum ...3

Format objektnog fajla

Sadržaj objektnog fajla u Mikrojavi

- informacije za punioca
  - veličina koda (u bajtovima)
  - veličina globalnih podatak (u rečima)
  - adresa *main* metoda
- programski kod