ARHITECTURA CALCULATOARELOR 2003/2004
CURSUL 14

5.2.2 Modelul de memorie IJVM

Să începem să vorbim despre arhitectura IJVM. În esență constă dintr-o memorie care poate fi văzută în două feluri diferite astfel: o zonă (un şir, de fapt) de 4.294.967.296 octeți (4 GB) sau o zonă (un şir) de 1.073.741.824 cuvinte, fiecare conținând câte 4 octeți. Spre deosebire de multe ISA, maşina virtuală Java nu face adresele direct vizibile la nivelul ISA, dar sunt câteva adrese implicite care oferă baza pentru un pointer. Instrucțiunile IJVM pot accesa memoria numai indexând de la aceşti pointeri.

În orice moment sunt definite următoarele zone de memorie:

- **Zona Constantelor** (Constant Pool). Această zonă nu poate fi scrisă de programul IJVM, şi constă din constante, şiruri şi pointeri către alte zone ale memoriei care se pot referi. Se încarcă când programul este adus în memorie şi nu se schimbă ulterior. CPP este un registru implicit care conține adresa primului cuvant al zonei constantelor.

- **Zona Variabilelor Locale** (Local Variable Frame). Pentru fiecare apel al unei proceduri, este alocată o zona de memorie pentru memorarea variabilelor pe timpul apelului. Această zonă poartă numele de **structură a variabilelor locale**. La începutul acestei structuri se află parametrii (numiți argumente) cu care procedura a fost apelată. Structura variabilelor locale nu include stiva de operanzi care este separată. Oricum, din motive de eficiență, arhitectura propusă a aleas implementarea stivei de operanzi imediat deasupra structurii variabilelor locale. Există un registru implicit care conține adresa primei locații a structurii variabilelor locale. Vom numi acest registru, registru LV. Parametrii vehiculați la apelul procedurii sunt memorați la începutul structurii variabilelor locale.

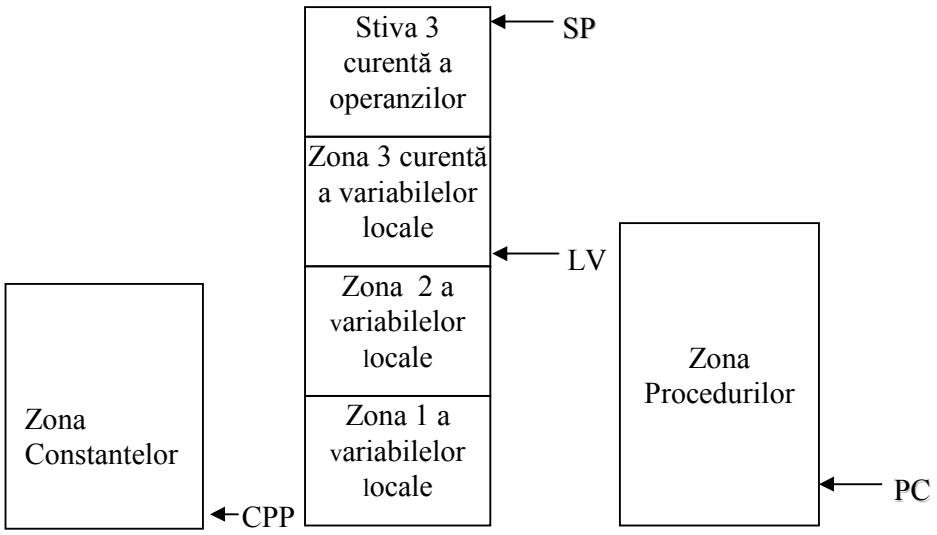

Figura 5.10 Zonele memoriei IJVM

- **Stiva Operanzilor** (Operand Stack). Se garantează că dimensiunea stivei nu va depăşi o anumită dimensiune, calculată în avans de compilatorul JAVA. Stiva operanzilor are un spațiu alocat imediat deasupra structurii variabilelor locale, ca în figura 5.10. În implementarea noastră, este de preferat să privim stiva operanzilor ca o parte a structurii variabilelor locale. În orice caz, există un registru care conține adresa cuvântului din vârful stivei. De notat că, spre deosebire de CPP şi LV, pointerul SP se schimbă în timpul execuției procedurii pe măsură ce operanzii sunt introduşi (push) sau extraşi (pop) din stivă.

- **Zona Procedurilor** (Method Area). În sfârşit, există o regiune a memoriei conținând programul, referit ca "text area" într-un proces UNIX. Există un registru implicit ce conține adresa instrucțiunii ce urmează să fie extrasă în continuare. Acest pointer este referit ca Program Counter sau PC. Spre deosebire de celelalte regiuni ale memoriei, zona procedurilor este tratată ca un şir de bytes.

Se cuvine facută o precizare referitoare la pointeri. Regiştrii CPP, LV şi SP sunt toți pointeri la cuvinte (words) nu la bytes, şi sunt decalate cu un număr de cuvinte. Pentru varianta pe întregi abordată, toate referințele la articolele din zona constantelor, zona variabilelor locale şi stiva sunt referințe la cuvinte, iar decalajele (offset) utilizate pentru indexare în aceste zone sunt decalaje numarate în cuvinte. De exemplu LV, LV+1 şi LV+2 se referă la primele trei cuvinte din zona variabilelor locale. În contrast, LV, LV+4 şi LV+8 se referă la cuvinte decalate cu câte 4 cuvinte, 16 bytes.

Utilizarea PC este diferită, se adresează la nivel de byte, incrementarea sau decrementarea PC schimbă adresa cu un număr de bytes, nu cu un număr de cuvinte. Această diferență este vizibila în portul special de memorie furnizat pentru PC în arhitectura Mic-1. Amintiți-vă că are un singur byte. Incrementând PC cu 1 şi inițiind o citire conduce la extragerea următorului byte. Incrementând SP cu 1 şi inițiind o citire conduce la extragerea următorului cuvânt.

5.2.3 Setul de instrucțiuni IJVM

Setul de instrucțiuni IJVM este prezentat în tabelul următor figura 5.11. Fiecare instrucțiune conține (opcode) şi uneori un operand, cum ar fi un decalaj în memorie (offset) sau o constantă. Prima coloană dă codarea hexazecimală a instrucțiunii. Cea de-a doua dă mnemonicele limbajului de asamblare. Cea de-a treia dă o explicație sumară a efectului său.

Există instrucțiunile care introduc un cuvânt din surse variate în stivă. Aceste surse includ zona constantelor (LDC_W), zona variabilelor locale (ILOAD), şi instrucțiunea însăşi (BIPUSH). O variabilă poate fi scoasă din stivă şi memorată în zona variabilelor locale (ISTORE). Două operații aritmetice (IADD şi ISUB) şi de asemenea două operații logice (IAND şi IOR) pot fi realizate folosind cele două cuvinte din vârful stivei ca operanzi. În toate operațiile aritmetice şi logice două cuvinte sunt extrase din stivă şi rezultatul introdus înapoi în stivă. Sunt oferite patru instrucțiuni de ramificare, una necondiționată (GOTO) şi trei condiționate (IFEQ, IFLT şi IF ICMPEQ). Toate aceste instrucțiuni, dacă sunt executate, ajustează valoarea PC-ului cu mărimea offsetului (16-biți cu semn) care urmează opcodul în instrucțiune. Acest offset este adăugat la adresa opcodului din instrucțiune. Exista deasemenea instrucțiuni IJVM pentru a schimba între ele cele două cuvinte din vârful stivei (SWAP), pentru a dubla cuvântul din vârful stivei (DUP) şi pentru a-l scoate din stiva (POP).

Hex	Mnemonic	Înțeles
0x10	BIPUSH byte	Introduce byteul în stivă
0x59	DUP	Copiază cuvântul din vârful stivei şi-l introduce în stivă
0xA7	GOTO offset	Salt necondiționat
0x60	IADD	Scoate două cuvinte din stivă şi introduce suma lor
0x7E	IAND	Scoate două cuvinte din stivă şi introduce AND-ul lor
0x99	IFEQ offset	Scoate un cuvânt din stivă şi salt dacă el e zero
0x9B	IFLT offset	Scoate un cuvânt din stivă şi salt dacă e mai mic ca zero
0x9F	IF_ICMPEQ offset	Scoate două cuvinte din stivă şi salt dacă sunt egale
0x84	IINC varnum const	Adaugă o constantă unei variabile locale
0x15	ILOAD varnum	Introduce variabila locală în stivă
0xB6	INVOKEVIRTUAL disp	Apel procedura
0x80	IOR	Scoate două cuvinte din stivă şi introduce OR-ul lor
0xAC	IRETURN	Întoarcere din procedura cu o valoare întreagă
0x36	ISTORE varnum	Scoate un cuvânt din stiva şi-l stocheaza într-o variabilă locală
0x64	ISUB	Scoate două cuvinte din stivă şi introduce diferența lor
0x13	LDC_W index	Introduce o constantă din zona constantelor în stivă
0x00	NOP	Nu face nimic
0x57	POP	Şterge cuvântul din vârful stivei
0x5F	SWAP	Schimbă între ele primele două cuvinte din vârful listei
0xC4	WIDE	Instrucțiunea prefix; următoarea instrucțiune are un index de 16 biți

Figura 5.11 Setul de instrucțiuni IJVM. Operanzii *byte, const, varnum* sunt pe un octet. Operanzii *disp, index* şi *offset* sunt pe doi octeți.

Unele instrucțiuni au formate multiple permițând o formă scurtă pentru versiunile des folosite.

În sfârşit, există o instrucțiune (INVOKEVIRTUAL) pentru a apela o altă procedură şi o altă instrucțiune pentru a determina revenirea din procedura apelată şi redarea controlului procedurii din care s-a facut apelul (IRETURN). Având în vedere complexitatea mecanismului am simplificat uşor definiția făcând posibil un mecanism de apelare şi revenire directe. Restricția este acea că spre deosebire de JAVA, noi permitem unei proceduri să apeleze numai o procedură existentă înăuntrul propriului obiect. Această restricție diminuează drastic orientarea pe obiecte, dar permite prezentarea un mecanism mult mai simplu prin ocolirea cererii de localizare dinamică a procedurii. S-a transformat aici limbajul Java într-un limbaj neorientat pe obiecte!! La toate calculatoarele exceptând JVM-ul, adresa proceduri apelate este determinată direct de instrucțiunea CALL, aşa că abordarea de aici este de fapt, din acest punct de vedere, un caz normal, nu o excepție.

Mecanismul pentru apelarea unei proceduri este următorul. Mai întâi, procedura care apelează introduce în stivă un pointer spre obiectul care urmează să fie apelat. Această referință nu este necesară în IJVM atât timp cât alt obiect nu poate fi specificat, dar a fost păstrată pentru că există în JVM. În figura 5.12 (a) această referință este indicată de OBJREF. Apoi apelantul introduce în stivă parametrii procedurii, în acest exemplu Parametrul 1, Parametrul 2, Parametrul 3. În sfârşit INVOKEVIRTUAL este executată.

Instrucțiunea INVOKEVIRTUAL include un deplasament ce indică poziția în zona constantelor a locației care conține adresa de început, din zona procedurilor, pentru procedura apelată. Oricum, dacă, codul procedurii se află în locul indicat de acest pointer, primi 4 octeți din zona procedurilor conțin date speciale:

- Primi doi octeți sunt interpretați ca un întreg pe 16 biți indicând numărul de parametrii pentru procedură (ei au fost deja plasați în stivă). Pentru această numărătoare, OBJREF este numărat ca parametru: parametrul 0. Acest întreg pe 16 biți, împreună cu valoarea SP-ului, furnizează localizarea OBJREF. De notat că LV indică spre OBJREF şi nu spre primul parametru. Într-un fel este arbitrar ce indică LV.
- Următorii doi octeți în zona procedurilor sunt interpretați ca un alt întreg pe 16 biți indicând mărimea zonei variabilelor locale pentru procedura apelată. Acest lucru este necesar deoarece o nouă stivă va fi stabilită pentru procedură, începând imediat deasupra zonei variabilelor locale.
- În sfârşit, al 5 lea octet în zona procedurilor conține primul opcode care va fi executat.

Secvență ce rezultă pentru INVOKEVIRTUAL este prezentată în figura 5.12. Cei doi octeți index fără semn care urmează opcodului sunt folosiți pentru construirea unui index în zona constantelor. Instrucțiunea calculează adresa de bază a noii zone a variabilelor locale prin scăderea numărului de parametrii din pointerul stivei şi setează LV astfel încât să indice OBJREF. La această locație, rescriind peste OBJREF, implementarea memorează adresa locației unde vechiul PC urmează să fie salvat. Această adresă este calculată adunând mărimea zonei variabilelor locale (parametri + variabile locale) cu adresa conținută în LV. Imediat deasupra acelei adrese unde vechiul PC va fi salvat este adresa unde va fi salvat vechiul LV. Imediat deasupra acestei adrese este începutul stivei pentru noua procedură apelată. SP este setat să indice spre vechiul LV, care este adresa imediat sub prima locație liberă din stivă.

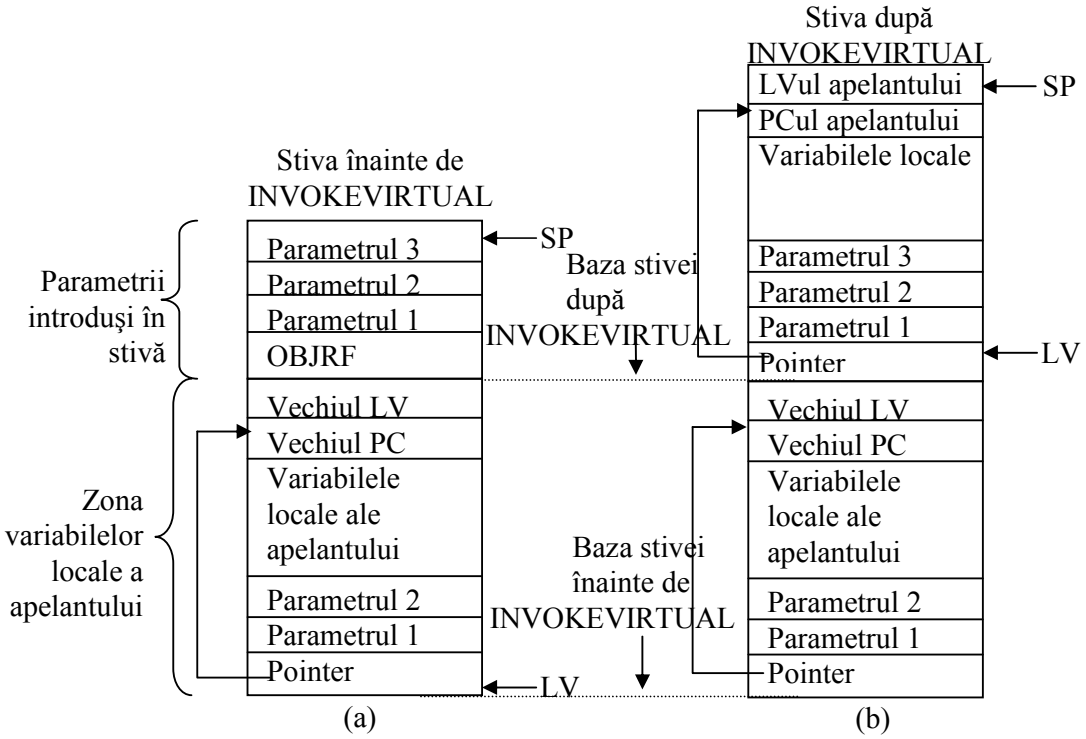

Figura 5.12 (a) Memoria înaintea execuției INVOKEVIRTUAL. (b) După execuție

Reamintim că SP-ul indică întotdeauna cuvântul din vârful stivei. Dacă stiva este goală, el indică prima locaţie sub sfârşitul stivei deoarece stiva noastră creşte în sus spre adresele înalte. În figurile noastre stivele întotdeauna cresc în sus spre cea mai înaltă adresă, din partea de sus a pagini.

Ultima operaţie pe care INVOKEVIRTUAL trebuie să o execute este setarea PC astfel încât să indice spre al cincilea byte din spaţiul de cod alprocedurii. Instrucţiunea IRETURN reface efectele realizate de operaţiile instrucţiunii INVOKEVIRTUAL, ca în figura 5.13. Ea eliberează spaţiul folosit de procedura care se termină (return). De asemenea reface stiva la stadiul anterior exceptând:

- cuvântul OBJREF (acum suprascris) şi toţi parametrii care au fost extraşi (pop) din stivă şi
- valoarea returnată a fost plasată în vârful stivei în locaţia ocupată anterior de OBJREF.

Pentru a restaura vechea stare, instrucţiunea IRETURN trebuie să fie capabilă să restaureze pointeri PC şi LV la vechile valori. El face acest lucru accesând pointerul de legătură (este cuvântul identificat de LV curent). În această locaţie, unde OBJREF a fost memorat la început, instrucţiunea INVOKEVIRTUAL a memorat adresa care conţine vechiul PC. Acest cuvânt şi cel de deasupra lui sunt utilizate pentru a restaura PC şi LV la vechile lor valori. Valoarea returnată, care este stocată în vârful stivei pentru procedura terminată, este copiată în locaţia unde a fost memorat iniţial OBJREF şi SP este refăcut să indice spre această locaţie. Controlul este deci redat instrucţiunii imediat următoare instrucţiunii INVOKEVIRTUAL.

Până acum, maşina noastră nu are instrucţiuni de intrare/ieşire şi nici nu o să adăugăm unele.

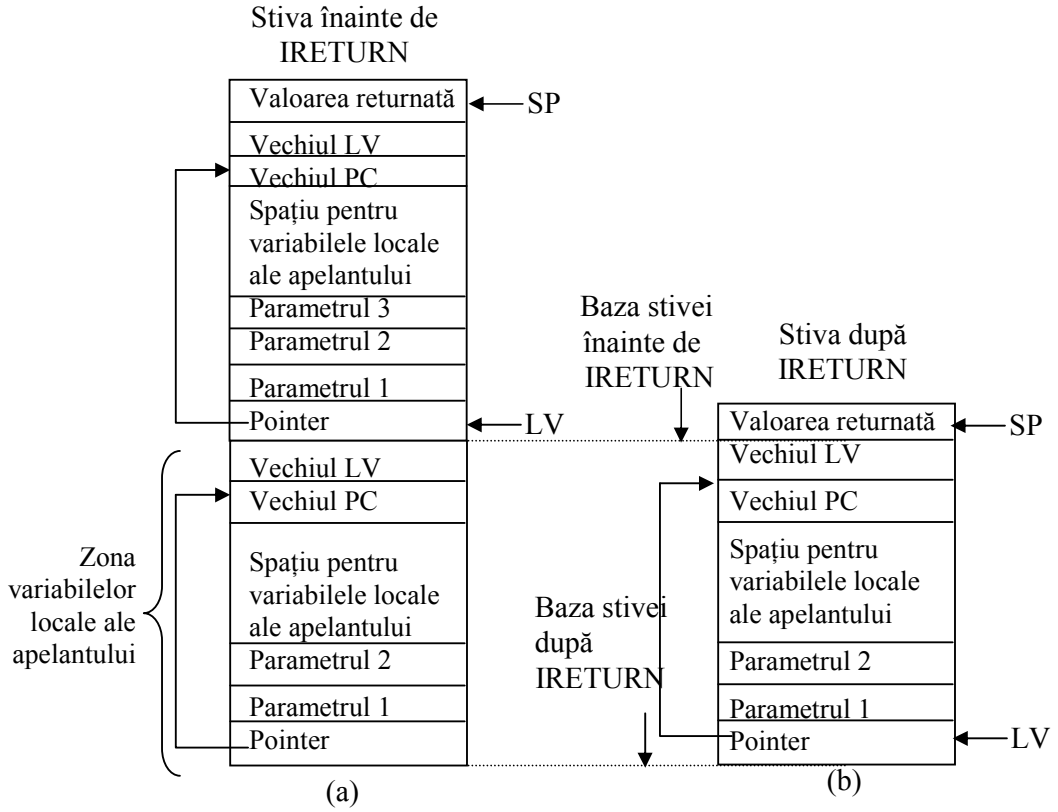

Figura 5.13 (a) Memoria înaintea execuţiei IRETURN. (b) După execuţia ei.

Nu are nevoie de ele cum nici JVM nu are, iar specificațiile oficiale nu menționează niciodată I /O. Teoria este aceea că dacă o maşinărie nu are I/O este "sigură". Citirea şi scrierea sunt executate in JVM prin intermediul apelării unor proceduri speciale care execută ele I/O.

3.3 Implementarea IJVM folosind Mic-1

Prezentăm relația care există între JAVA şi IJVM. În figura 5.14 (a) este prezentat un fragment simplu de cod JAVA. Când se oferă acest program unui compilator JAVA, acesta va produce un cod în limbajul de asamblare IJVM de tipul celui din figura 5.14 (b). Numerele de linie de la 1 la 15 din stânga listingului programului în limbaj de asamblare nu sunt oferite de compilator. Nici comentariile care urmează după //. Ambele au fost introduse pentru a explicita figura.

Asamblorul JAVA va traduce apoi programul de asamblare în programul binar din figura 5.14 (c). De fapt compilatorul JAVA face propria asamblare şi produce direct programul binar. Pentru acest exemplu am presupus ca i este variabila locală 1, j este varabila locală 2 iar, k este variabila locală 3.

```
i = j + k;              1        ILOAD j          // i = j + k    0x15 0x02
if (i= = 3)             2        ILOAD k                       0x15 0x03
            k=0;        3        IADD                          0x60
else                    4        ISTORE i                      0x36 0x01
          j = j-1;      5        ILOAD i          // if (i<3)    0x15 0x01
                        6        BIPUSH 3                      0x10 0x03
                        7        IF_ICMPEQ L1                  0x9F 0x00 0x0D
                        8        ILOAD j          // j = j-1     0x15 0x02
                        9        BIPUSH 1                      0x10 0x01
                        10       ISUB                          0x64
                        11       ISTORE j                      0x36 0x02
                        12       GO TO L2                      0xA70x00 0x07
                        13  L1:  BIPUSH 0         // k = 0       0x10 0x00
                        14       ISTORE k                      0x36 0x03
                        15  L2:
(a)                             (b)                            (c)
```

Figura 5.14 (a) Program în JAVA. (b) Programul corespunzător în ansamblare JAVA. (c) Program IJVM în hexazecimal.

Am ajuns în sfârşit în acel punct în care putem pune toate piesele împreună. În figura 5.15 este prezentat un microprogram care rulează pe Mic-1 şi interpretează IJVM-ul. Este un microprogram surprinzător de scurt – doar 112 microinstrucțiuni în total. Există trei coloane pentru fiecare microinstrucțiune: eticheta, microcodul şi un comentariu. De notat este că microinstrucțiunile consecutive nu sunt neapărat localizate în adrese consecutive în control store, cum am mai spus.

Până acum alegera numelor multor registre din figura inițială a arhitecturii ar trebui să fie evidentă: CPP, LV, şi SP sunt folosiți pentru a ține minte pointeri din zona constantelor, zona

variabilelor locale şi pentru vârful stivei, respectiv, iar PC-ul ţine minte adresele pentru următorul byte care va fi extras din fluxul de instrucţiuni. MBR-ul este un registru de un byte care secvenţial ţine minte octeţii din fluxul de instrucţiuni în ordinea care vin din memorie pentru a fi interpretaţi.

TOS-ul şi OPC-ul sunt registre suplimentare. Folosirea lor este descrisă mai jos.

La un anumit moment fiecare registru e garantat că ţine minte o anumită valoare, dar fiecare poate fi folosit ca un registru temporar dacă este nevoie. La începutul şi la sfârşitul fiecărei instrucţiuni TOS-ul conţine valoarea locaţiei de memorie indicate de către SP, cuvântul din vârful stivei. Valoarea este redundantă deoarece poate fi mereu citită din memorie, dar având pentru ea un registru adesea se salvează o referire la memoriei. De exemplu, instrucţiunea POP elimină vârful stivei şi din această cauză trebuie să extragă noul cuvânt din vârful stivei din memorie în TOS.

Registrul OPC este un registru temporar. Nu are nici o folosire dinainte prescrisă. Este folosit de exemplu, pentru a salva adresele de opcode pentru o instrucţiune de ramificare în timp ce PC-ul este incrementat pentru accesul la parametri. Este de asemenea folosit ca un registru temporar în instrucţiunea IJVM de ramificare condiţionată.

Ca toate interpretoarele, microprogramul din figura amintită are o buclă principală care extrage, decodifică şi execută instrucţiunile din programul care este interpretat, în acest caz instrucţiunile IJVM. Bucla principală începe la linia etichetată Main1. Porneşte cu invariantul cu care PC-ul a fost încărcat dinainte, o adresă a locaţiei de memorie care conţine opcode-ul. Mai mult, opcode-ul este extras în MBR. De notat este că în această implementare, când se va reveni în acest punct este nevoie ca PC-ul sa fi fost actualizat cu valoarea adresei locatiei care contine noul opcode care trebuie interpretat, iar acest opcode să fi fost deja extras în MBR.

Această secvenţă iniţială este executată la începutul fiecărei instrucţiuni, deci este imporatant să fie cât mai simplă posibil. Printr-o atentă proiectare a lui Mic-1, hardware şi software, s-a reuşit să se reducă bucla principală la o singură microinstrucţiune. În momentul în care maşina a pornit, de fiecare dată când microinstrucţiunea este executată, opcode-ul IJVM-ului care va fi executat este deja prezent în MBR. Ceea ce microinstrucţiunea face este o ramificare la microcodul care execută această instrucţiune IJVM, şi de asemenea extragerea următorului byte de după opcode, care ar putea fi fie un byte operand fie următorul opcode.

Acum putem spune adevăratul motiv pentru care fiecare microinstrucţiune îşi precizează numele succesorului ei în loc ca ele să fie executate secvenţial. Toate adresele din control store corespunzătoare opcode-urilor trebuie să fie rezervate pentru primul cuvânt al interpretării instrucţiunii corespunzătoare. De exemplu, codulul pentru interpretarea POP începe la 0x57 şi codul pentru interpretarea DUP începe la 0x59. (Cum ştie MAL-ul să pună POP-ul la 0x57 este unul dintre misterele universului – presupunem că există un fişier undeva care să-i spună acest lucru).

Din păcate codul pentru POP este lung de trei microinstrucţiuni, deci dacă va fi plasat în cuvinte consecutive va interfera cu începutul lui DUP. Din momentul în care toate adresele control store corespunzătoare opcode-ului sunt efectiv rezervate, microinstrucţiunile, altele decât aceea iniţială, din fiecare secvenţă trebuie să fie puse în golurile dintre adresele

rezervate. Din acest motiv, pentru că este multă muncă să sari de colo colo, a avea o microramificare explicită (o microinstrucțiune care ramifică) la fiecare câteva microinstrucțiuni, pentru de a sări din gaură în gaură, va fi foarte eficient.

Etichetă	Operație	Comentariu
Main1	PC = PC + 1; fetch; goto (MBR)	MBR conține opcode-ul; ia următorul byte; furnizează
nop1	goto Main1	Nu face nimic
iadd1	MAR = SP = SP – 1; rd	Citeşte primul cuvânt de sub vârful stivei
iadd2	H = TOS	H = vârful stivei
iadd3	MDR = TOS = MDR + H; wr; goto Main1	Adună cele 2 cuvinte; scrie în vârful stivei
isub1	MAR = SP = SP – 1; rd	Citeşte primul cuvânt de sub vârful stivei
isub2	H = TOS	H = vârful stivei
isub3	MDR = TOS = MDR - H; wr; goto Main1	Scade cele 2 cuvinte; scrie în vârful stivei
iand1	MAR = SP = SP – 1; rd	Citeşte primul cuvânt de sub vârful stivei
iand2	H = TOS	H = vârful stivei
iand3	MDR = TOS= MDR AND H; wr; goto Main1	Execută AND; scrie în vârful stivei
ior1	MAR = SP = SP – 1; rd	Citeşte primul cuvânt de sub vârful stivei
ior2	H = TOS	H = vârful stivei
ior3	MDR = TOS = MDR OR H; wr; goto Main1	Execută OR; scrie în vârful stivei
dup1	MAR = SP = SP + 1	Incrementează SP şi copiază în MAR
dup2	MDR = TOS; wr; goto Main1	Scrie noul cuvânt în stivă
pop1	MAR = SP = SP – 1; rd	Citeşte primul cuvânt de sub vârful stivei
pop2		Aşteaptă ca noul TOS să fie citit din memorie
pop3	TOS = MDR; goto Main1	Copiază noul cuvânt în TOS
swap1	MAR = SP – 1; rd	Setează MAR la SP – 1; citeşte al doilea cuvânt din stivă
swap2	MAR = SP	
swap3	H = MDR; wr	Setează MAR la vârful stivei;
swap4	MDR = TOS	Salvează TOS în H; scrie al doilea cuvânt în vârful stivei
swap5	MAR = SP – 1; wr	Copiază vechiul TOS în MDR
swap6	TOS = H; GOTO Main1	Setează MAR la SP–1; scrie ca al doilea cuvânt din stivă Reactualizează TOS
bipush1	SP = MAR = SP + 1;	MBR = byteul care trebuie memorat în stivă
bipush2	PC = PC + 1; fetch	Incrementează PC-ul, extrage următorul opcode
bipush3	MDR = TOS = MBR; wr; goto Main1	Extensia semnului şi memorează în stivă
iload1	H = LV	MBR conține indexul; copiază LV-ul în H
iload2	MAR = MBRU + H; rd	MAR = adresa variabilei locale care trebuie pusă în stivă
iload3	MAR = SP = SP + 1	SP indică noul vârf de stivă; pregăteşte scrierea
iload4	PC = PC + 1; fetch; wr	Inc PC-ul; ia următorul opcode; scrie noul vârf al stivei
iload5	TOS = MDR; goto Main1	Reactualizează TOS
istore1	H = LV	MBR conține indexul; copiază LV-ul în H
istore2	MAR = MBRU + H	MAR = adresa variabilelor locale în care trebuie stocat
istore3	MDR = TOS; wr	Copiază TOS în MDR; scrie cuvântul
istore4	SP = MAR = SP - 1; rd	Citeşte primul cuvânt de sub vârful stivei
istore5	PC = PC + 1; fetch	Incrementează PC-ul, extrage următorul opcode
istore6	TOS = MDR; goto Main1	Reactualizează TOS
wide1	PC = PC + 1; fetch; GOTO (MBR OR 0x100)	O ramificare multiplă cu cel mai semnificativ bit setat
wide_iload1	PC = PC + 1; fetch	MBR conține primul byte de index; extrage pe al 2-lea
wide_iload2	H = MBR << 8	H = primul byte de index deplasat la stânga cu 8 biți
wide_iload 3	H = MBR OR H	H = un index pe 16 biți al variabilelor locale
wide_iload4	MAR = LV + H; rd; goto iload3	MAR = adresa variabilei locale care trebuie pusă în stivă

wide_istore1	PC = PC + 1; fetch	MBR conţine primul byte de index; extrage pe al 2-lea
wide_istore2	H = MBRU << 8	H = primul byte de index deplasat la stânga cu 8 biţi
wide_istore3	H = MBRU OR H	H = un index pe 16 biţi al variabilelor locale
wide_istore4	MAR = LV + H; goto istore3	MAR = adresa variabilei locale care trebuie stocată
ldc_w1	PC = PC + 1; fetch	MBR conţine primul byte de index; extrage pe al 2-lea
ldc_w2	H = MBRU << 8	H = primul byte de index deplasat la stânga cu 8 biţi
ldc_w3	H = MBRU OR H	H = un index pe 16 biţi în zona constantelor
ldc_w4	MAR = H + CPP; rd; goto iload3	MAR = adresa constantei
iinc1	H = LV	MBR conţine indexul; copiază LV-ul în H
iinc2	MAR = MRBU + H; rd	Copiază LV + index în MAR; citeşte variabilele
iinc3	PC = PC + 1; fetch	Extrage constanta
iinc4	H = MDR	Copiază variabila în H
iinc5	PC = PC + 1; fetch	Extrage următorul opcode
iinc6	MDR = MBR + H; wr; goto Main1	Pune sum în MBR; reactualizează variabila
goto1	OPC = PC – 1	Salvează adresa opcode-ului
goto2	PC = PC + 1; fetch	MBR = primul byte de offset; extrage al doilea byte
goto3	H = MBR << 8	Deplasează şi salvează primul byte cu semn în H
goto4	H = MBRU OR H	H = offset pe16 biţi al ramificării
goto5	PC = OPC + H; fetch;	Adună offset-ul la OPC
goto6	goto Main1	Aşteaptă extragerea următorului opcode
iflt1	MAR = SP = SP - 1; rd	Citeşte primul cuvânt de sub vârful stivei
iflt2	OPC = TOS	Salvează temporar TOS în OPC
iflt3	TOS = MDR	Pune noul vârf de stivă în TOS
iflt4	N = OPC; if(N) goto T; else goto F	Ramificare după bitul N
ifeq1	MAR = SP = SP - 1; rd	Citeşte primul cuvânt de sub vârful stivei
ifeq2	OPC = TOS	Salvează temporar TOS în OPC
ifeq3	TOS = MDR	Pune noul vârf de stivă în TOS
ifeq4	Z = OPC; if(Z) goto T; else goto F	Ramificare după bitul Z
if_icmpeq1	MAR = SP = SP - 1;rd	Citeşte primul cuvânt de sub vârful stivei
if_icmpeq2	MAR = SP = SP – 1	Setează MAR pentru a citi noul vârf al stivei
if_icmpeq3	H = MDR; rd	Copiază al doilea cuvânt din stivă în H
if_icmpeq4	OPC = TOS	Salvează temporar TOS în OPC
if_icmpeq5	TOS = MDR	Pune noul vârf al stivei în TOS
if_icmpeq6	Z = OPC - H; if(Z) goto T; else goto F	Dacă primele 2 cuvinte sunt egale, goto T, altfel goto F
T	OPC = PC - 1; fetch; goto goto2	La fel ca goto1; este necesar pentru ţinta adreselor
F1	PC = PC + 1	Sare primul byte de offset
F2	PC = PC + 1; fetch	PC-ul indică către următorul opcode
F3	goto Main1	Aşteaptă să se extragă opcode-ul
invokevirtual1	PC = PC + 1; fetch	MBR = indexul primului byte; inc PC; extrage al doilea
invokevirtual2	H = MBRU << 8	Deplasează şi salveză primul byte în H
invokevirtual3	H = MBRU OR H	H = offset la pointerul procedurii din CPP
invokevirtual4	MAR = CPP + H; rd	Ia pointerul la procedură din zona CPP
invokevirtual5	OPC = PC + 1	Salvează temporar Return PC în OPC
invokevirtual6	PC = MDR; fetch	PC indică către noua procedură; ia numărul de param.
invokevirtual7	PC = PC + 1; fetch	Extrage al 2 lea byte din numărătorul parametrilor
invokevirtual8	H = MBRU << 8	Deplasează şi salveză primul byte în H
invokevirtual9	H = MBRU OR H	H = numărul de parametri
invokevirtual10	PC = PC + 1; fetch	Extrage primul byte al # locale
invokevirtual11	TOS = SP – H	TOS = adresa lui OBJREF – 1
invokevirtual12	TOS = MAR = TOS + !	TOS = adresa lui OBJREF (noul LV)
invokevirtual13	PC = PC + 1; fetch	Extrage al 2 lea byte al # locale
invokevirtuall4	H = MBRU << 8	Deplasează şi salveză primul byte în H
invokevirtual15	H = MBRU OR H	H = # locale

invokevirtual16	MDR = SP + H + 1; wr	Scrie peste OBJREF pointerul de legătură
invokevirtual17	MAR = SP = MDR;	Setează SP, MAR la locaţia care reţine vechiul PC
invokevirtual18	MDR = OPC; wr	Salvează vechiul PC deasupra variabilelor locale
invokevirtual19	MAR = SP = SP + 1	SP indică locaţia care reţine vechiul LV
invokevirtual20	MDR = LV; wr	Salvează vechiul LV deasupra PC salvat
invokevirtual21	PC = PC + 1; fetch	Extrage primul opcode al noii proceduri
invokevirtual22	LV = TOS; goto Main1	Setează LV-ul pentru a pointa către zona LV
ireturn1	MAR = SP = LV; rd	Resetează SP, MAR pt. a primi pointerul de legătură
ireturn2		Aşteaptă citirea
ireturn3	LV = MAR = MDR; rd	Setează LV cu pointerul de legătură; ia vechiul PC
ireturn4	MAR = LV + 1	Setează MAR pentru a citi vechiul LV
ireturn5	PC = MDR; rd; fetch	Reface PC; extrage următorul opcode
ireturn6	MAR = SP	Setează MAR pentru a scrie TOS
ireturn7	LV = MDR	Reface LV
ireturn8	MDR = TOS; wr; goto Main1	Salvează valoarea returnată în vârful stivei originale

Figura 5.15 Exemplu de microprogram

Pentru a vedea cum lucrează interpretorul, să presupunem de exemplu că MBR-ul conţine valoarea 0x60 care este opcode pentru IADD. Într-o singură microinstrucţiune bucla principală realizează trei lucruri:

1. Incrementează PC-ul, făcându-l să conţină adresa primului byte după opcode.

2. Iniţiază o extragere a următorului byte în MBR. Acest byte va fi folositor mai devreme sau mai târziu, fie ca un operand pentru instrucţiunea curentă IJVM sau ca următorul opcode (ca în cazul instrucţiunii IADD, care nu are bytes de operand).

3. Face o ramificare către adresa conţinută în MBR la începutul lui Main1. Adresa este egală cu valoarea numerică a opcode-ului care este executat acum. A fost plasată aici de microinstrucţiunea anterioară. De notatat este că valoarea extrasă în această microinstrucţiune nu joacă nici un rol în ramificarea multiplă.

Extragerea următorului byte începe aici pentru a fi disponibilă la începutul celei de a treia microinstrucţiuni. S-ar putea să avem sau să nu avem nevoie de ele, dar eventual vor fi folosite, deci începerea extragerii lor de pe acum nu poate cauza nici un rău.

Dacă se întâmplă ca biţii din MBR să fie toţi 0, upcode-ul pentru instrucţiunea NOP, următoarea microinstrucţiune este aceea etichetată nop1, extrasă dinn locaţia 0. Deoarece instrucţiunea nu face nimic se va realiza doar legătura înapoi la bucla main, unde secvenţa este repetată, dar cu un nou opcode care este potrivit în MBR.

Încă o dată accentuăm faptul că microinstrucţiunile din figura anterioară nu sunt consecutive în memorie şi că Main1 nu se află la adresa 0 (deoarece nop1 trebuie să fie la adresa 0). Este treaba asamblorului să pună microinstrucţiunile la adrese convenabile şi să le lege împreună între ele într-o scurtă secvenţă folosind câmpul NEXT_ADDRESS. Fiecare secvenţă începe la adresa corespunzătoare valorii numerice a opcode-ul IJVM pe care îl interpretează (de exemplu POP începe de la 0x57) dar restul secvenţei poate fi oriunde în control store şi nu necesar în adrese consecutive.