CURS 7

ARHITECTURA SETULUI DE INSTRUCŢIUNI

Conţinut:

I. Introducere în arhitectura setului de instrucţiuni
II. Formatul instrucţiunilor
III. Interdependenţa set de instrucţiuni - organizare internă a procesorului

I. INTRODUCERE ÎN ARHITECTURA SETULUI DE INSTRUCŢIUNI

Arhitectura setului de instrucţiuni specifică unui procesor cuprinde:

⇒ setul de instrucţiuni (instrucţiuni specifice la nivel maşină) recunoscute de procesor,
⇒ tipurile de date care pot fi manipulate cu aceste instrucţiuni şi
⇒ contextul în care aceste instrucţiuni operează.

Arhitectura setului de instrucţiuni (ASI) este o componentă a maşinii de calcul, vizibilă programatorului la nivel de limbaj de asamblare, componentă care realizează interfaţa între software şi hardware (figura 4.1.).

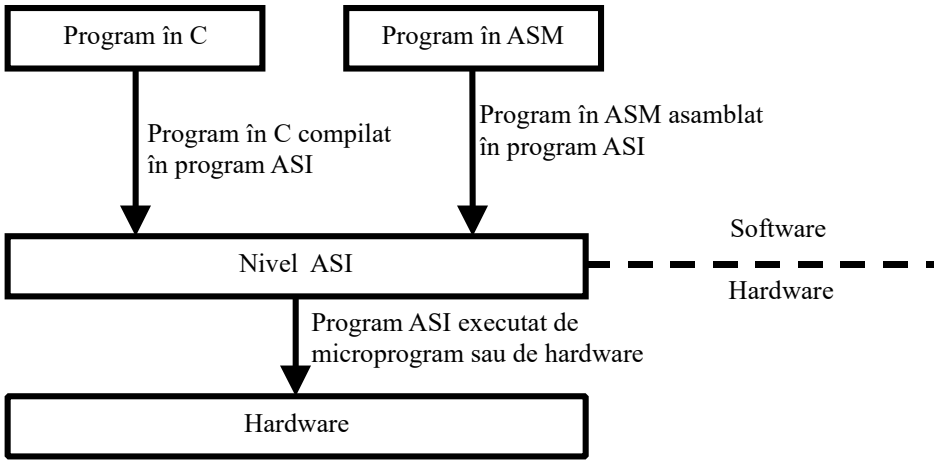

Figura 4.1. Arhitectura setului de instrucţiuni ca interfaţă între hardware şi software (adaptată după [Patterson90])

Pentru a putea comanda componenta hardware a unui calculator trebuie cunoscute instrucţiunile recunoscute de acesta la nivelul maşinii de bază. Instrucţiunile implementate hardware pe o anumită maşină formează setul de instrucţiuni, sau limbajul inclus în ASI. De aceea, în figura 4.1. programele în limbaj de nivel înalt (C în figură) sunt compilate pentru a se genera un program recunoscut de maşină, numit în figură program ASI. Acesta este programul în cod binar recunoscut de unitatea de control (UC) şi interpretat de către o maşină cu algoritm de tranziţie a stărilor internă UC; algoritmul poate fi implementat ca microprogram sau poate fi cablat în hardware. În mod similar programele scrise în limbaj de asamblare (ASM) trebuie translatate cu ajutorul unui program translator numit asamblor în programe executabile, conform ASI specific maşinii.

Setul de instrucţiuni implementat pe un anumit microprocesor poate fi clasificat după mai multe criterii, dintre care menţionăm [Patterson90]:

- Numărul de *adrese de operanzi pe instrucţiune*. Numărul de adrese pe instrucţiune afectează în mod direct complexitatea şi dimensiunile (în număr de biţi) ale unei instrucţiuni. Indiferent de tipul de format ales şi de tipul operaţiilor numărul maxim de operanzi /adrese) este 3, iar numărul minim este 0.
- *Locul de stocare a operanzilor* pentru instrucţiunile executate de CPU. Operanzii operaţiilor aritmetice şi logice pot fi imediaţi, sau se pot găsi în registrele interne UCP, în memoria principală, în spaţiul de I/O, sau în memorie de tip stivă. Există arhitecturi care efectuează operaţii aritmetice şi logice doar cu operanzi din registrele interne, iar cu exteriorul permit doar operaţii de încărcare şi stocare a operanzilor (arhitectură load / store[1]). Alte tipuri permit operanzi doar în memoria stivă (stack architecture[2]). Locul de stocare al operanzilor afectează nu numai formatul instrucţiunilor, dar şi modurile de adresare ce trebuiesc codificate.
- *Tipul operaţiilor* efectuate de setul de instrucţiuni. La implementarea unui set de instrucţiuni se urmăreşte optimizarea operaţiilor cele mai frecvent efectuate pe o anumită maşină. Astfel că tipul instrucţiunilor depinde în special de aplicaţiile pentru care este destinat procesorul. De exemplu, un microprocesor destinat aplicaţiilor de tip DSP (prelucrare a semnalelor digitale) are instrucţiuni ce optimizează operaţiile de: adunare, înmulţire, stocare temporară, shift[3]-are şi manipulare de matrice.
- *Tipul datelor prelucrate* de unităţile aritmetice şi logice. Diferitele maşini pot lucra cu numere în virgulă fixă (întregi sau fracţionare) şi numere în virgulă mobilă (numere reale).

Setul de instrucţiuni al unui procesor poate avea două caracteristici principale:

1. **regularitate (ortogonalitate)** - această caracteristică impune ca fiecare instrucţiune din setul de instrucţiuni să poată opera cu orice tip de date, conţinute în oricare registru sau oricare locaţie de memorie, şi să poată fi adresate prin oricare mod de adresare.
2. **completitudine** - această cerinţă ar impune ca setul de instrucţiuni să cuprindă tot setul de operatori disponibili într-un limbaj. Dar cuprinderea întregului set de operatori, operatori cu frecvenţe foarte diferite de utilizare, ar duce la implementări şi performanţe departe de cele optime. În consecinţă se introduce o cenzură pentru operatorii cu frecvenţă scăzută de apariţie în programe, în avantajul operatorilor cu frecvenţă ridicată. Uneori se introduce regula (cam generală dar practică): "*Execută rapid operaţiile frecvente şi corect pe cele rare*". În faza de proiectare a unui procesor, setul de instrucţiuni se croieşte anticipând spectrul de aplicaţii ale

[1] load / store = încarcă / stochează

[2] stack architecture = arhitectură pe bază de stivă

[3] shift = deplasare a informaţiei dintr-un registru la stânga sau la dreapta cu un anumit număr de poziţii binare.

procesorului, astfel ca acestea să poată fi uşor exprimate sau programate. Aceasta este faza în care se pot determina operaţiile ce vor fi folosite frecvent.

II. FORMATUL INSTRUCŢIUNILOR

Formatul instrucţiunilor se reprezintă de obicei în formă grafică sub forma unor dreptunghiuri ce simbolizează câmpurile binare incluse în instrucţiuni. Din punctul de vedere al lungimii instrucţiunilor pot exista două tipuri de ASI:

- arhitecturi la care ***lungimea*** instrucţiunilor este ***fixă***, pentru tot setul recunoscut
- arhitecturi cu ***lungime variabilă*** a instrucţiunilor recunoscute

La microprocesoarele cu lungime fixă a instrucţiunilor (de exemplu arhitecturile RISC[4], dar şi arhitecturi CISC cum ar fi cea a calculatorului Felix C256) unitatea de control, care interpretează instrucţiunile şi lansează comenzile pentru execuţia acestora, este mai simplă, dar programele ocupă un spaţiu mai mare în memorie. Formatul de lungime variabilă a instrucţiunilor (în general microprocesoare cu arhitectura CISC[5]) implică o complexitate mai mare a unităţii de control. Lungimea variabilă a instrucţiunilor (lungime mică pentru operaţii simple şi mare pentru cele complexe) permite programe cu cod mai compact (ocupă în general o zonă mai mică de memorie) şi de asemenea permite extinderea uşoară a setului de instrucţiuni, pentru o familie de procesoare cu aceeaşi ASI, pentru că nu mai există limitarea unui număr fix de octeţi.

Instrucţiunile se reprezintă în memoria principală ca octeţi succesivi (unul sau mai mulţi octeţi în funcţie de tipul de procesor). Scopul fiecărei instrucţiuni recunoscute de procesor este să specifice operaţia ce trebuie executată de hardware, operanzii utilizaţi şi locul unde se stochează rezultatul operaţiei. Formatul unei instrucţiuni include două tipuri principale de informaţii:

- codul operaţiei (Op-Code în figura 4.2.) pe care o comandă instrucţiunea respectivă
- câmpuri de adrese pentru specificarea modului de referire la operanzi şi la locul unde se stochează rezultatul. Vom numi aceste câmpuri specificatori de operanzi, pentru că ei indică adresa sau modul de calcul al adresei efective a operanzilor (în figură s-au considerat trei specificatori de adresă de operanzi: OP1, OP2, OP3).

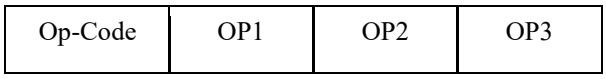

Figura 4.2. Exemplu al formatului general pentru o instrucţiune ASI

Primul câmp binar al instrucţiunii reprezintă codul numeric identificator al instrucţiunii (OpCode = codul operaţiei efectuate de instrucţiune). Lungimea acestuia este în general diferită în funcţie de complexitatea operaţiei şi eventualele informaţii suplimentare pe care le cuprinde; aceste informaţii pot fi de exemplu: modul de adresare al operanzilor sursă, un indicator de condiţie testat

[4] RISC = Reduced Instruction Set Computer
[5] CISC = Complex Instruction Set Computer

pentru a lua o decizie la execuţia instrucţiunii, un prefix al codului operaţiei care modifică modul de lucru al acelei instrucţiuni. Prefixul poate avea efecte diverse, cum ar fi:

- repetare de un număr de ori a instrucţiunii cu prefix,
- indicaţia că se lucrează cu operanzi sau adrese extinse (dimensiune dublă faţă de cea normală)
- identifică o instrucţiune specială destinată coprocesorului aritmetic.

Apoi urmează specificatorii pentru operanzi (modul de găsire al operanzilor se numeşte *mod de adresare* al operanzilor). Ordinea în care sunt dispuşi specificatorii de operanzi în formatul instrucţiunii diferă de la procesor la procesor. Vom considera, doar pentru exemplificare, un format de instrucţiune cu trei operanzi, ca cel din figura 4.2, şi la care specificatorul OP1 se referă la destinaţie (aceasta poate fi un registru intern, o locaţie de memorie sau porturi de I/O[6]), iar OP2 şi OP3 sunt specificatori de operanzi sursă (operanzii putând fi date imediate, sau stocaţi în registre, memorie, ori porturi de I/O).

Instrucţiunea recunoscută de maşină este scrisă deci în cod binar care codifică o acţiune a procesorului. Specificatorii de operand ce apar *explicit* în formatul instrucţiunii după codul operaţiei se referă de obicei la adrese din memoria principală, adrese de porturi, operanzi imediaţi (date sau adrese imediate), registre interne de uz general ale UCP. Unele din registrele interne ale UCP au funcţii specifice în cazul anumitor instrucţiuni, iar prin decodificarea codului operaţiei acestor instrucţiuni, registrele respective fiind recunoscute *implicit* ca sursă sau destinaţie de operanzi.

După modul în care se stochează în memoria principală adresele operanzilor imediaţi din câmpul de specificatori, există două convenţii. Convenţia "Little Endian" (aluzie la şirul indian care are în faţă pe cel mai mic) plasează întotdeauna pe prima poziţie, la adresa cea mai mică, cuvântul cel mai puţin semnificativ al unei adrese multi - cuvânt. Stocarea în memorie se face în ordine, de la cuvântul cel mai puţin semnificativ (adresa mică), către cuvântul cel mai semnificativ stocat la adresa cea mai mare. În figura 4.3. se prezintă un exemplu de stocare în memorie, în convenţia "Little Endian", a unui cuvânt de 32 biţi. Memoria din figură este organizată pe octet, având deci alocată câte o adresă fiecărui cuvânt de 8 biţi. Valorile numerice au fost scrise în hexazecimal.

[6] I/O = intrare / ieşire

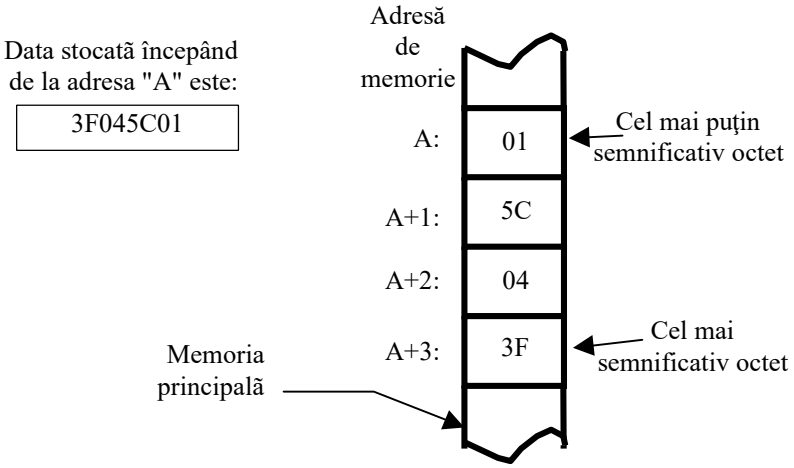

Figura 4.3. Exemplu de stocare a unui cuvânt de 32 de biţi, într-o memorie organizată pe octet, convenţia utilizată fiind Little Endian. Într-o instrucţiune ce adresează această dată pe 32 de biţi se va specifica adresa "A", unde este stocat octetul cel mai puţin semnificativ.

Pentru cealaltă convenţie, numită "Big Endian" (marele indian), ordinea este inversă, adică "în faţă", la adresa cea mai mică se plasează octetul cel mai semnificativ al cuvântului multi-octet.

Legat de convenţia de stocare în memorie a informaţiilor (instrucţiuni, date, adrese) trebuie menţionat aici că la unele microprocesoare accesul la obiecte mai mari decât un octet se poate face doar dacă informaţia este aliniată. Alinierea se referă la faptul că accesul la un obiect de dimensiunea *D* octeţi se face prin specificarea adresei *A* primului octet ocupat, unde adresa A respectă ecuaţia:

$$A \ mod \ D = 0$$

Considerând o memorie adresabilă la nivel de octet şi un microprocesor de 32 biţi, alinierea şi nealinierea informaţiilor pe 32 de biţi se exemplifică grafic în figura 4.4.

Nealinierea produce complicaţii la citirea memoriei şi conduce la pierdere de timp. În exemplul din figura 4.4. pentru informaţia aliniată citirea se poate face într-un singur ciclu de acces la memorie. Pentru informaţie ne-aliniată trebuiesc cel puţin două cicluri de acces la memorie. Dar situaţia de ne-aliniere prezentată nu este atât de critică cât ar fi dacă nealinierea s-ar realiza prin stocarea informaţiei pe 32 de biţi la adrese succesive, de exemplu în ordinea: 0B, 0C, 0D şi 0E hex, când citirea ar trebui făcută pentru fiecare octet în parte.

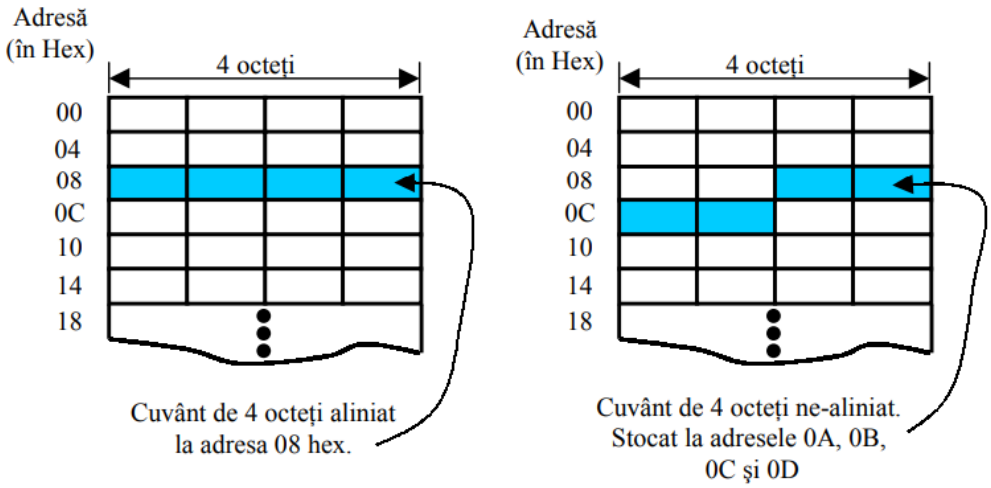

Figura 4.4. Exemplu de adrese de acces la obiecte aliniate şi nealiniate. Cuvântul pe 32 de biţi nealiniat poate fi citit în cel mai bun caz prin două accesări succesive la memorie pentru citirea a câte 2 octeţi.

III. INTERDEPENDENŢA
SET DE INSTRUCŢIUNI - ORGANIZARE INTERNĂ A PROCESORULUI

O sursă de controverse începând de la prima generaţie de calculatoare a fost: câte *adrese explicite* pentru operanzi să includă o instrucţiune? Cu cât sunt mai puţine adrese cu atât e mai scurtă instrucţiunea. Dar limitând numărul de adrese se limitează domeniul de funcţii pe care fiecare instrucţiune le poate realiza. În general, mai puţine adrese înseamnă mai multe instrucţiuni primitive şi deci programe mai lungi pentru o anumită sarcină de rezolvat.

În timp ce necesităţile de stocare în memorie pentru instrucţiuni mai scurte şi programe mai lungi tind să se echilibreze cu instrucţiunile lungi şi programe scurte, programe mai lungi cer un timp mai lung de execuţie. Pe de alta parte, instrucţiunile lungi, cu adrese multiple, cer de obicei circuite mai complexe pentru decodare şi prelucrare. De multe ori procesoarele sunt clasificate după numărul maxim de adrese de memorie principală în câmpurile instrucţiunilor lor. Calculatoarele conţinând un număr diferit de adrese au în general instrucţiuni de lungimi diferite. Instrucţiunile cu doi operanzi necesită cel mult trei adrese. De exemplu, pentru o operaţie de adunare, simbolizată prin mnemonica “add” pot fi implementate instrucţiuni cu minim 0 şi maxim 3 operanzi, asa cum se observă în tabelul 4.1.

Numărul de câmpuri de adresă în formatul instrucţiunii unui procesor depinde organizarea internă a registrelor sale, dar în acelaşi timp influenţează modul în care este proiectat procesorul ca organizare, din punctul de vedere al registrelor interne. Numărul de adrese poate fi diferit, de la instrucţiune la instrucţiune la acelaşi procesor. De exemplu instrucţiunea de scădere poate avea trei adrese de operanzi (doi sursă şi unul destinaţie), iar instrucţiunea de complementare poate avea un singur operand (registrul sursă este şi destinaţie).

Tabelul 4.1.

Număr adrese	Format instrucţiune	Comentariu
3	**add z,x,y**	; z ← x+y , unde x,y,z, sunt locaţii de ;memorie, nume de registre, sau date imediate
2	**add x,y**	; x ←x+y sau Ac ← x+y , în funcţie de procesor ; (Ac = registru acumulator)
1	**add x**	; Ac ← Ac+x
0	**add**	; adună primele două articole din vârful stivei şi ; stochează rezultatul în noul vârf al stivei

Conform legăturii dintre organizarea UCP şi numărul de adrese pe instrucţiune, vom putea încadra cele mai multe procesoare într-una din următoarele trei categorii:

1. *organizare pe bază de registru acumulator (Ac)* la care pentru majoritatea operaţiilor unul dintre operanzi se găseşte implicit în acumulator. De asemenea rezultatul operaţiei se 'acumulează' în acest registru cu funcţie specială.
2. *organizare pe bază de registre de utilizare generala (RG)*. La acest tip de organizare operanzii sunt explicit localizaţi fie în registrele interne (fără restricţii), fie în registrele din locaţiile memoriei principale.
3. *organizare de tip stivă (ST)* la care operanzii se găsesc implicit în vârful stivei şi tot aici se stochează rezultatul. Există unele procesoare de tip stivă care folosesc două stive: una pentru operaţii asupra datelor şi alta pentru operaţii asupra adreselor.

Tabelul 4.2. Exemple de instrucţiuni

Tip arhitectură	Instrucţiuni	Comentarii
Ac	add x	;Ac ← Ac + x sau Ac ← Ac + @x ;adună la acumulator conţinutul registrului x sau al ;memoriei de la adresa dată de x

RG	add r1,r2,r3
	add r1,r2
	mov r1,r2
	add r1,x
		; r1 ← r2 + r3
		; r1 ← r1 + r2 sau r2 ← r1 + r2
		; r1 ← r2 sau invers
		; r1 ← r1 + @x

ST	push x add	;trimite cuvântul de adresă x în vârful stivei, ;VS ← @x ;aduce primele 2 articole din stivă, le adună şi ; stochează rezultatul în stivă

Fiecare dintre cele trei tipuri de arhitecturi au avantaje şi dezavantaje. Astfel arhitectura pe bază de acumulator conduce la instrucţiuni scurte, pentru că acumulatorul este referit ca registru implicit la multe dintre instrucţiunile maşinii. Conduce şi la o densitate relativ mare a codului (programelor) stocate în memorie, dar acumulatorul constituie un puternic factor de limitare al vitezei de lucru al procesorului. Maşinile pe bază de registre de

uz general prezintă avantaje din punctul de vedere al uşurinţei generării codului de către programe compilatoare. Dacă nu ar fi existat aceste registre, foarte multe din rezultatele intermediare ar fi trebuit stocate în memoria principală ceea ce ar fi micşorat viteza de prelucrare. Dezavantajul arhitecturilor de tip RG este dat de faptul că fiecare registru trebuie adresat de către instrucţiunea ce-l utilizează, astfel că lungimea instrucţiunilor este relativ mare. Ultimul tip de arhitectură permite o evaluare extrem de uşoară a operaţiilor pe baza notaţiei inverse (notaţia "poloneză" - vezi exemplul de mai jos). Aceasta conduce şi la instrucţiuni cu lungime foarte redusă. Din păcate o stivă este mai dificil de adresat în mod aleator, iar codul rezultat este relativ ineficient. Stiva introduce şi o limitare în viteză şi de aceea acest tip de arhitectură este rar implementat la procesoare de uz general.

În tabelul 4.2. s-au prezentat câteva exemple de instrucţiuni pentru cele trei tipuri de arhitecturi. Multe calculatoare pot fi încadrate în cele 3 categorii de mai sus, dar cele mai multe combină caracteristici din mai multe moduri de organizare.
Pentru a ilustra influenţa numărului de adrese în programele executate de procesor vom evalua egalitatea:

$$x = a \times (b + c) \qquad (4.1)$$

utilizând exemple de instrucţiuni cu 0, 1, 2, şi 3 adrese. Vom folosi simbolurile (mnemonicele):

- add, mul, - pentru operaţiile aritmetice elementare de adunare şi înmulţire
- load, store - transfer de la/la memorie spre/dinspre acumulator
- push, pop - înscriere respectiv extragere din stivă.
- r# - registre interne ale UCP, unde # este un indice

Presupunem că operanzii a, b şi c din relaţia (4.1) sunt stocaţi în memorie, în locaţiile de adrese a, b, c, iar rezultatul se va introduce la adresa x în memorie.

a) pentru arhitecturi pe baza de registre de uz general:

- *instrucţiuni cu 3 adrese care pot efectua operaţii aritmetice sau logice cu operanzi din memorie (CISC):*

```
add    r1,b,c      ;r1 ← @b + @c
mul    x,r1,a      ;@x ← r1 * @a
```

Se observă că acest format poate utiliza fiecare câmp de adresă operand pentru a specifica un operand dintr-un registru sau din memorie. Deşi programul este foarte scurt, el conţine instrucţiuni cu format de lungime mare şi se desfăşoară relativ lent. Astfel la prima instrucţiune se fac două accesări la memorie pentru citirea operanzilor b şi c. La fel pentru a doua instrucţiune sunt necesare două accesări succesive la memorie (operand şi rezultat).

- *instrucţiuni cu 3 adrese (RISC);*

```
load    r1,a        ; r1 ← @a
load    r2,b        ; r2 ← @b
load    r3,c        ;
add     r2,r2,r3    ; r2 ← (r2 + r3)
mul     r1,r1,r2    ; r1 ← (r1 * r2)
store   x,r1        ; @x ← r1
```

La acest tip de arhitecturi se folosesc doar instrucţiuni de *load (încarcă) şi store (stochează)* când se comunică cu memoria, iar toate celelalte instrucţiuni se referă la operanzi din registrele UCP (deci prelucrare foarte rapidă)

- *instrucţiuni cu 2 adrese (CISC)*

```
load    r1,b        ;r1 ← @b
add     r1,c        ;r1 ← r1 +@c
mul     r1,a        ;r1 ← r1 * @a
store   x,r1        ;@x ← r1
```

b) pentru arhitecturi pe baza de acumulator (instrucţiuni cu o adresa, acumulatorul fiind adresat implicit):

```
load    b           ; ac ← @b
add     c           ; ac ← ac + @c
store   t           ; @t ← ac
load    a           ; ac ← @a
mul     t           ; ac ← ac * @t
store   x           ; @x ← ac
```

unde s-a notat cu “t” adresa unei locaţii de memorie folosita pentru stocarea temporara a rezultatelor intermediare

c) pentru arhitecturi pe bază de stivă (instrucţiuni cu zero adrese)

```
push    a       ;vs ← @a, fie “adr” adresa curentă a vârfului stivei
                (vs) ; iar stiva creşte spre adrese mici
push    b       ;vs ← @b, la adr-1
push    c       ;vs ← @c, la adr-2
add             ;vs ← (b + c), stocat la adr-1          zero adrese
mul             ;vs ← a*(b+c), stocat la adr            zero adrese
pop     x       ;@x ← vs, vs pointează la adr+1
```

Pentru a evalua expresiile aritmetice într-un calculator stivă, e necesar să se convertească expresia în notaţia poloneză (inversă). Ideea a pornit de la matematicianul polonez Lukasievics J, care a notat o expresie de forma "a + b" ca "ab+".

După acelaşi principiu putem exemplifica notaţia poloneză şi pentru alte ecuaţii:

x = a* (b + c) = abc+*.
x = (a+b)*(c+d) = ab+cd+*
x = (a+b)*(c+d)*(e+f) = ab+cd+ef+**

După cum se vede, notaţia poloneză are avantajul că nu foloseşte paranteze. Ba mai mult, pornind de la stânga la dreapta se indică toate operaţiile ce trebuiesc efectuate, în ordine, pe o arhitectură de tip stivă.