Acta et Commentationes, Sciences of Education, nr. 2(36), 2024
p. 78-86

ISSN 1857-0623
E-ISSN 2587-3636

CZU: 004.3/.4

DOI: 10.36120/2587-3636.v36i2.78-86

# MOTIVE PENTRU CARE LIMBAJUL DE ASAMBLARE RĂMÂNE FUNDAMENTAL ÎN PROGRAMAREA MODERNĂ

**Ala GASNAŞ**, doctor, conferenţiar universitar
https://orcid.org/0000-0002-7174-7027
**Angela GLOBA**, doctor, conferenţiar universitar
https://orcid.org/0000-0002-2653-0320
Universitatea Pedagogică de Stat „Ion Creangă" din Chișinău

**Rezumat**. În acest articol este scoasă în evidență importanța limbajului de asamblare în programarea modernă. Deși limbajele de nivel înalt domină pe scară largă dezvoltarea software-ului, limbajul de asamblare rămâne esențial în anumite domenii. Acesta oferă un control precis și direct asupra hardware-ului și resurselor sistemului, permițând optimizarea performanței și accesul la caracteristici specifice ale procesorului. Dezvoltatorii de drivere, sisteme în timp real și aplicații încorporate de nivel jos beneficiază în mod special de cunoașterea limbajului de asamblare. Astfel, limbajul de asamblare rămâne un instrument fundamental pentru cei care doresc să înțeleagă în profunzime funcționarea hardware-ului și să optimizeze performanța sistemelor informatice.
**Cuvinte cheie**: limbaj de asamblare, optimizarea performanţei, limbajele de programare de nivel înalt.

# REASONS WHY ASSEMBLY LANGUAGE REMAINS FUNDAMENTAL IN MODERN PROGRAMMING

**Abstract**. This article explores the importance of assembly language in modern programming. While high-level languages dominate software development widely, assembly language remains essential in certain domains. It provides precise and direct control over hardware and system resources, allowing performance optimization and access to specific processor features. Developers of drivers, real-time systems, and low-level embedded applications particularly benefit from knowledge of assembly language. Thus, assembly language remains a fundamental tool for those seeking to deeply understand hardware operation and optimize the performance of computer systems.
**Keywords**: assembly language, performance optimization, high-level programming languages

## Introducere

Limbajul de asamblare este considerat un limbaj de programare de nivel scăzut, având o strânsă corelație cu instrucțiunile codului de mașină adaptate la arhitectura specifică. Fiecare arhitectură de mașină are propriul său limbaj de asamblare distinct. În acest tip de limbaj, operațiile și instrucțiunile sunt reprezentate prin intermediul simbolurilor. Din acest motiv, limbajul de asamblare mai este cunoscut și sub numele de cod de mașină simbolic.

Chiar dacă limbajele de programare de nivel înalt sunt predominante și sunt utilizate în principal pentru dezvoltarea de aplicații și programe software, importanța limbajului de asamblare în lumea modernă nu poate fi ignorată. Un programator poate obține beneficii semnificative învățând să programeze în limbajul de asamblare și să-l utilizeze în proiectele sale. Astăzi, limbajul de asamblare facilitează manipularea directă a componentelor

hardware, abordarea problemelor critice legate de performanță și oferă acces la instrucțiunile specifice pentru procesoare [1].

Domeniile de aplicare ale limbajului de asamblare includ:

- *Dezvoltarea de drivere pentru dispozitive*: programe software care permit sistemului de operare să comunice cu hardware-ul dispozitivelor atașate, cum ar fi imprimantele, plăcile grafice, tastaturile etc. Dezvoltarea de drivere pentru dispozitive implică adesea codificarea în limbajul de asamblare pentru a avea control fin asupra funcționalității hardware-ului. Limbajul de asamblare oferă acces direct la registrele hardware și la instrucțiunile specifice ale procesorului, ceea ce este esențial pentru a asigura o comunicare eficientă și corectă între sistemul de operare și dispozitivul hardware.
- *Sisteme în timp real*: sunt sisteme informatice care trebuie să răspundă la evenimente în timp real, cu o precizie specifică și într-un interval de timp determinat. Aceste sisteme sunt utilizate în domenii în care timpul de răspuns este crucial, cum ar fi controlul industrial, sistemele medicale, vehiculele autonome etc. Dezvoltarea de sisteme în timp real necesită adesea programare la niveluri foarte scăzute ale hardware-ului și optimizare pentru performanță și timp de răspuns. Limbajul de asamblare este preferat în acest context datorită controlului precis asupra resurselor hardware și a timpului de execuție rapid pe care îl oferă [1].
- *Sisteme încorporate (înglobate) de nivel scăzut*: sisteme integrate în dispozitive care au nevoie de funcționalități specifice, cum ar fi sistemele de control pentru automobile, electrocasnice inteligente, dispozitive medicale sau chiar drone. Limbajul de asamblare este utilizat adesea pentru a programa aceste sisteme încorporate deoarece oferă un control precis și eficient asupra hardware-ului. Acest nivel scăzut de programare este necesar pentru a asigura performanța și fiabilitatea dispozitivelor încorporate [1].
- *Coduri de pornire*: secvențe de instrucțiuni care sunt executate la pornirea unui computer sau a unui dispozitiv. La acest stadiu, sistemul trebuie să aibă un program minim care să inițializeze hardware-ul și să pregătească sistemul de operare sau alte programe pentru încărcare. Limbajul de asamblare este adesea folosit pentru a scrie aceste coduri de pornire, deoarece permite un control direct și eficient asupra resurselor hardware.
- *Activități de inginerie inversă*: procesul de analiză a unui program sau a unui dispozitiv pentru a înțelege cum funcționează și ce face. Limbajul de asamblare este util în aceste activități deoarece permite analizarea codului mașină și înțelegerea detaliilor interne ale unui program sau dispozitiv. Inginerii folosesc limbajul de asamblare pentru a dezvălui funcționalități ascunse, pentru a găsi erori sau vulnerabilități de securitate și pentru a realiza alte investigații tehnice detaliate.

Limbajul de asamblare este esențial în aceste domenii datorită controlului precis și direct pe care îl oferă asupra hardware-ului și a capacității sale de a accesa și manipula resursele hardware la nivel de instrucțiuni.

**Rolul limbajului de asamblare în programare**

Capacitatea limbajului de asamblare de a oferi un control precis și direct asupra hardware-ului și de a permite accesul și manipularea resurselor hardware la nivel de instrucțiuni este esențială în domenii specifice ale programării și dezvoltării software. Această caracteristică fundamentală a limbajului de asamblare îl face indispensabil în domenii precum dezvoltarea de drivere pentru dispozitive, sisteme în timp real și alte aplicații care necesită interacțiunea directă cu hardware-ul. Prin urmare, învățarea limbajului de asamblare devine esențială pentru programatorii care doresc să lucreze în aceste domenii, deoarece le oferă cunoștințele și abilitățile necesare pentru a lucra eficient și eficace cu resursele hardware. Argumente care susțin importanța și relevanța învățării limbajului de asamblare sunt reflectate în figura 1.

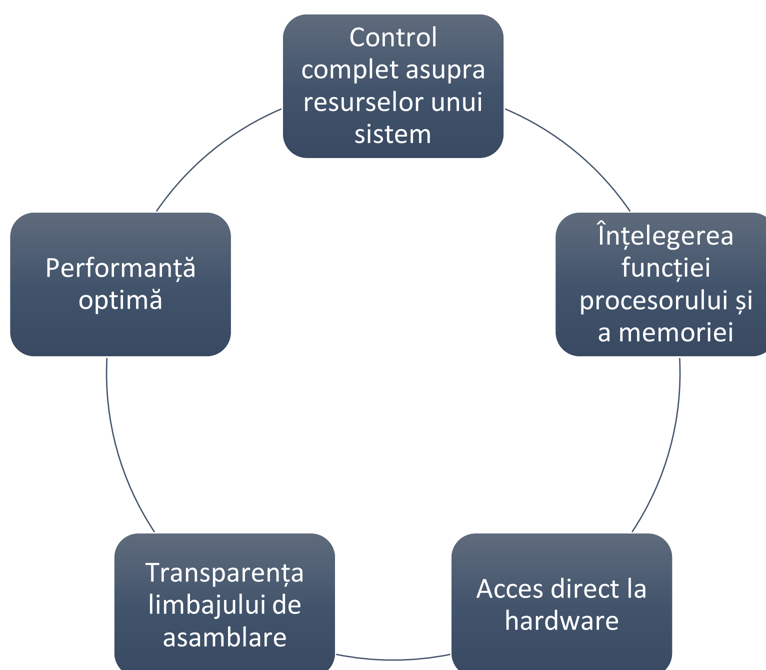


**Figura 1. Argumente privind importanța și relevanța învățării limbajului de asamblare**

*Control complet asupra resurselor unui sistem*: Limbajul de asamblare oferă programatorilor cea mai mare apropiere de procesorul unei mașini. În limbajul de asamblare se poate scrie cod pentru a accesa registrele și se pot gestiona adresele de memorie direct pentru a prelua valori. Astfel, scrierea unui program în limbaj de asamblare poate aduce beneficii semnificative, deoarece acesta oferă acces la optimizarea vitezei și la obținerea unei eficiențe și performanțe majore [2].

*Înțelegerea funcției procesorului și a memoriei*: În cazul în care se dezvoltă un compilator sau un driver, este important să se înțeleagă complet funcția procesorului. Scrierea codului în limbajul de asamblare permite programatorilor să studieze modul de funcționare a procesorului și a memoriei. Deși limbajul de asamblare poate părea greu de

înțeles și poate genera cod sursă mai mare decât limbajele de nivel înalt, investiția în învățarea limbajului de asamblare poate spori înțelegerea proceselor interne ale unui sistem informatic [2]. Aceasta înseamnă că programatorii vor avea o mai bună înțelegere a modului în care funcționează procesorul și memoria, cum sunt manipulate datele și instrucțiunile de programare la nivel de mașină, și cum se realizează comunicarea directă cu hardware-ul. Această înțelegere profundă poate ajuta la depistarea și remedierea problemelor de performanță sau de securitate, precum și la optimizarea codului pentru o mai bună eficiență.

În același timp, învățarea limbajului de asamblare poate stimula dezvoltarea abilităților de programare și gândire analitică. Așa cum, limbajul de asamblare este un limbaj de nivel scăzut, scrierea și înțelegerea codului în acest limbaj necesită o gândire detaliată și riguroasă. Această experiență poate dezvolta capacități de analiză a algoritmilor și de înțelegere a conceptelor fundamentale ale programării. În plus, deoarece limbajul de asamblare permite accesul direct la resursele hardware, programatorii pot dezvolta aplicații care sunt mai eficiente și mai bine adaptate la nevoile specifice ale sistemelor lor. Astfel, investiția în învățarea limbajului de asamblare poate aduce beneficii semnificative în dezvoltarea abilităților de programare și în creșterea performanței și a eficienței aplicațiilor software.

*Acces direct la hardware*: Limbajul de asamblare este singurul limbaj care comunică direct cu hardware-ul unui sistem. Accesul direct la hardware, în contextul limbajului de asamblare, se referă la capacitatea programatorului de a interacționa direct cu componente hardware ale sistemului, cum ar fi procesorul, memoria, controlerele de periferice etc., fără a fi nevoie de un intermediar software complex. Acest lucru este posibil deoarece limbajul de asamblare oferă instrucțiuni care sunt directe și precise în ceea ce privește operațiunile efectuate de hardware.

Atunci când se utilizează limbajul de asamblare, programatorul poate accesa și manipula resursele hardware la nivel de instrucțiuni individuale. De exemplu, programatorul poate scrie cod care să încarce date în registrele procesorului, să efectueze operații aritmetice sau logice pe aceste date, să acceseze adrese specifice de memorie pentru a citi sau scrie date etc.

Accesul direct la hardware este esențial în dezvoltarea unor aplicații care necesită performanțe ridicate și control fin asupra resurselor sistemului. Acest lucru este valabil în special în domenii precum programarea sistemelor încorporate, unde eficiența și performanța sunt esențiale, și în dezvoltarea de drivere de dispozitive, unde este necesară interacțiunea directă cu hardware-ul dispozitivului.

*Transparența limbajului de asamblare*: În comparație cu limbajele de nivel înalt, care sunt adesea abstracte în ceea ce privește tipurile de date, limbajul de asamblare este mai simplu și mai transparent. Acest lucru se datorează, în mare parte, numărului redus de

operațiuni pe care le include. Astfel, limbajul de asamblare este util pentru analiza algoritmilor, deoarece oferă o semantică și un flux de control clar. De asemenea, facilitează depanarea, deoarece este mai puțin complex decât limbajele de nivel înalt. În general, acesta implică mai puține cheltuieli generale în comparație cu limbajele de nivel înalt.

În anumite situații, limbajul de asamblare este preferat față de alte limbaje de programare, cum ar fi C++ sau C, deoarece oferă un control direct și precis asupra hardware-ului și a resurselor sistemului. În aceste cazuri limbajul de asamblare oferă:

– *Performanță optimă*: Codul scris în limbajul de asamblare este adesea mult mai eficient, din punct de vedere al performanței, decât echivalentul său în C++ sau C. Acest lucru se datorează controlului direct asupra instrucțiunilor procesorului și a modului în care sunt utilizate resursele sistemului.
– *Controlul precis al resurselor*: În limbajul de asamblare, programatorul are control total asupra resurselor hardware-ului, cum ar fi registrele, memoria și instrucțiunile procesorului. Acest lucru poate fi util în aplicații care necesită manipularea directă a hardware-ului sau optimizarea resurselor pentru performanță maximă [3].
– *Cerințe stricte de performanță*: În anumite aplicații, cum ar fi sistemele înglobate sau aplicațiile cu cerințe stricte de performanță în timp real, utilizarea limbajului de asamblare poate fi necesară pentru a îndeplini aceste cerințe [3].
– *Învățare a conceptelor de bază*: Scrierea de cod în limbajul de asamblare poate ajuta la înțelegerea mai profundă a funcționării hardware-ului și a conceptelor de bază ale programării, cum ar fi gestionarea memoriei, manipularea datelor și controlul fluxului.

Cu toate acestea, este important să subliniem că utilizarea limbajului de asamblare poate fi dificilă și consumatoare de timp, iar codul rezultat poate fi mai greu de întreținut și de înțeles decât echivalentul său scris într-un limbaj de nivel înalt, cum ar fi C++ sau C. De aceea, utilizarea limbajului de asamblare este recomandată doar în anumite situații unde performanța maximă sau controlul precis al hardware-ului sunt prioritare.

Un exemplu de utilizare a limbajului de asamblare în scop de performanță este prezentat printr-un program de calcul al sumei elementelor dintr-un vector de numere întregi. Mai întâi, vom prezenta o implementare în limbajul de asamblare, iar apoi în limbajul C++.

```
.model small
.stack 100h
.data
    vector dw 1, 2, 3, 4, 5  ; vectorul de numere întregi
.code
start:
     mov ax, @data      ; inițializăm segmentul de date
     mov ds, ax
     mov cx, 0          ; începem cu suma zero
```

```
        lea si, vector  ; încărcăm adresa de început a vectorului în registrul SI
     mov bx, [si]         ; încărcăm primul element al vectorului în registrul bx
sum_loop:
      add cx, bx          ; adăugăm elementul curent la suma totală
      add si, 2       ; trecem la următorul element al vectorului (int are 2 octeți)
      mov bx, [si]         ; încărcăm următorul element al vectorului în bx
      cmp bx, 0     ; verificăm dacă am ajuns la sfârșitul vectorului (0 este un marker de terminare)
      jnz sum_loop         ; dacă nu am ajuns la sfârșit, repetăm procesul
                           ; în acest moment, suma totală se află în registrul cx
 mov ax, 4c00h        ; cod de terminare a programului
       int 21h
end start
```

Exemplu în limbajul C++:

```
#include <iostream.h>
#include <vector>
using namespace std;

int main() {
    vector<int> nums = {1, 2, 3, 4, 5}; // vectorul de numere întregi

    int sum = 0;
    for (int i = 0; i < nums.size(); ++i) {
        sum += nums[i];
    }
cout<< "suma elementelor este: "<< sum <<endl;
    return 0;
}
```

Diferența principală constă în nivelul de abstractizare. Limbajul de asamblare este mai aproape de arhitectura hardware și oferă mai mult control asupra resurselor, ceea ce poate duce la performanțe mai bune.

În acest exemplu programul scris în limbajul de asamblare are potențialul de a fi mai performant decât echivalentul său în limbajul C++ din mai multe motive:

*Control total asupra hardware-ului*: În limbajul de asamblare, programatorul are control complet asupra instrucțiunilor procesorului și a modului în care acestea sunt executate. Acest lucru poate duce la optimizări fine și la evitarea costurilor suplimentare asociate cu codul generat automat de compilatoarele C++.

*Eficiența instrucțiunilor*: Codul de asamblare permite utilizarea unor instrucțiuni specifice și optimizate pentru sarcina curentă, fără a fi necesară adăugarea unui nivel suplimentar de abstractizare. Acest lucru poate duce la o utilizare mai eficientă a resurselor hardware și la o performanță îmbunătățită.

*Controlul direct al memoriei*: Programatorii în limbaj de asamblare au control complet asupra manipulării memoriei, permițându-le să optimizeze accesul la memorie și să evite direcționări inutile. În C++, unele operațiuni pot implica accesul la memorie suplimentar sau direcționări care pot afecta performanța.

*Reducerea overhead-ului*: Deoarece limbajul de asamblare este un limbaj de nivel scăzut, poate fi scris într-un mod care minimizează overhead-ul asociat cu gestionarea memoriei, apelurile de funcții și alte operațiuni. În C++ există adesea costuri suplimentare asociate cu gestionarea memoriei și abstracțiile limbajului.

Un alt exemplu de control strict, performanță și învățare a conceptelor de bază este explicarea conceptului de dată. Tipuri de date simple pot fi legate de limbajul de asamblare prin examinarea modului în care acesta gestionează datele și tipurile de date simple. În limbajul de asamblare, datele sunt reprezentate sub formă de octeți (byte) sau cuvinte (word), iar tipurile de date simple includ numere întregi, caractere și valori logice.

În continuare, propunem un exemplu de cod scris în limbajul de asamblare care demonstrează definirea și utilizarea unor tipuri de date simple:

```
.model small
.stack 100h
.data
  numar_int db 10  ; Definirea unui număr întreg de 2 octeți
  caracter db 'A'  ; Definirea unui caracter
     val_logica db 1 ; Definirea unei valori logice (0 sau 1)
.code
start:
  mov ax, @data
  mov ds, ax
  ; Afișarea numărului întreg
     mov dl,numar_int
  add dl,'0' ; Convertirea numărului la cod ASCII
  mov ah,02h ; Funcția de afișare a caracterului
  int 21h
  ; Afișarea caracterului
     mov dl, caracter
  mov ah, 02h
  int 21h
  ; Afișarea valorii logice
     mov dl, val_logica
  add dl, '0' ; Convertirea valorii la cod ASCII
  mov ah, 02h
  int 21h
     mov ah, 4Ch
  int 21h
end start
```

În C++, putem defini aceleași tipuri de date simple ca în exemplul de mai sus în limbajul de asamblare. Iată cum putem face acest lucru în C++:

```
#include <iostream>
int main() {
char numar_int = 10;// Definirea unui număr întreg de 1 octet (char)
char caracter = 'A';    // Definirea unui caracter
bool valoare_logica = true;    // Definirea unei valori logice (bool)
std::cout<<"Numar intreg:"<<static_cast<int>(numar_int) << std::endl;
std::cout << "Caracter: " << caracter << std::endl;
std::cout << "Valoare logica: " << valoare_logica << std::endl;
return 0;
}
```

În acest exemplu de cod C++, folosim tipurile de date simple din C++ pentru a defini și inițializa aceleași valori ca în exemplul de limbaj de asamblare.

Exemplul folosit pentru explicarea conceptului de dată prin intermediul limbajului de asamblare este util din mai multe motive:

*Înțelegerea mai profundă a funcționării hardware-ului*: Limbajul de asamblare oferă o perspectivă directă asupra funcționării hardware-ului computerului. Prin înțelegerea limbajului de asamblare, programatorii pot înțelege mai bine modul în care operațiunile simple sunt efectuate de către procesor, cum ar fi operațiile pe biți și bytes.

*Optimizarea performanței*: În anumite cazuri, scrierea unor părți critice ale programului în limbajul de asamblare poate duce la o performanță mai bună decât cea obținută folosind limbaje de nivel înalt, cum ar fi C++ sau Java. Aceasta se datorează faptului că programatorii pot controla direct instrucțiunile procesorului și pot realiza optimizări fine la nivel de cod.

*Învățarea conceptelor fundamentale*: În timp ce limbajul de asamblare poate părea mai complicat la început, învățarea sa poate ajuta programatorii să înțeleagă mai bine conceptele fundamentale ale programării și funcționării unui computer. Aceasta poate îmbunătăți înțelegerea altor limbaje de programare și a conceptelor software în general.

*Dezvoltarea abilităților de depanare*: Lucrul cu limbajul de asamblare poate dezvolta abilitățile de depanare ale unui programator. Înțelegerea modului în care codul de asamblare se traduce în instrucțiuni și date la nivel de hardware poate face mai ușor de depistat și depanat erorile și problemele de performanță.

În concluzie, limbajul de asamblare îi ajută pe programatori să înțeleagă mai bine funcționarea computerului și să optimizeze performanța aplicațiilor lor.

## Concluzii

Cu toate că limbajul de asamblare este un limbaj mai dificil de scris, de înțeles și de întreținut decât codul scris în limbaje de nivel înalt, limbajul de asamblare rămâne fundamental în programarea modernă din mai multe motive:

1. În primul rând, oferă un control precis și direct asupra hardware-ului și resurselor sistemului, permițând programatorilor să optimizeze eficient performanța și să acceseze caracteristici specifice ale procesorului.
2. Cunoașterea limbajului de asamblare poate fi esențială în dezvoltarea de drivere pentru dispozitive, sisteme în timp real și alte aplicații înglobate de nivel jos.
3. Capacitatea de a manipula hardware-ul direct face ca limbajul de asamblare să fie indispensabil în situații unde eficiența și controlul detaliat sunt prioritare.

*Articol realizat în cadrul proiectului de cercetări științifice „Metodologia implementării TIC în procesul de studiere a științelor reale din perspectiva conceptului STEAM și Inteligenței Artificiale", codul 040101, din cadrul Programului instituțional de cercetare (2024-2027), aprobat prin Ordin MEC nr. 102 din 01.02.2024*

## Bibliografie

1. CAREER GUIDE. Why are assembly languages useful? 23.11.20244. Disponibil la: https://uk.indeed.com/career-advice/career-development/what-is-assembly-language (accesat 15.03.2024).
2. PAL, Kaushik. Why is learning assembly language still important? În: *Techopedia*, 6.12.2022. Disponibil la: https://www.techopedia.com/why-is-learning-assembly-language-still-important/7/32268 (accesat 15.03.2024).
3. AI and the LinkedIn community. What are the benefits of using assembly language? Disponibil la: https://www.linkedin.com/advice/0/what-benefits-using-assembly-language-skills-computer-hardware-6h4he (accesat 15.03.2024).