**UNIVERSITATEA DE MEDICINĂ, FARMACIE, ȘTIINTE ȘI TEHNOLOGIE “GEORGE EMIL PALADE” DIN TÂRGU MUREȘ**

**FACULTATEA DE INGINERIE**

**Program de studiu: INFORMATICĂ**

**LUCRARE DE LICENȚĂ**

**Mediu de Simulare pentru Algoritmi în Limbaj de Asamblare**

**Coordonator științific:** Conf. dr. ing. Haller Piroska

**Asistent:** Asistent drd. Bolboacă Roland

**Absolvent:** Gorea Tudor-Andrei

?

**Cuprins:**

(completez la final cand stiu si paginile)

1. Abstract ....................................................................................
2. Introducere ...............................................................................
3. Aspecte Teoretice .....................................................................
4. Aspecte de Proiectare ...............................................................
5. Aspecte de Implementare .........................................................
6. Concluzii ..................................................................................
7. Bibliografie ..............................................................................

**ABSTRACT**

Limbajele de Asamblare sunt un instrument foarte puternic în programarea *low-level* a calculatoarelor. Oferind control la nivelul *hardware*, aceste limbajele sunt o opțiune foarte bună în optimizarea performanței aplicațiilor, în ingineria inversă a programelor, în scopul analizei securității acestora, și în înțelegerea arhitecturii calculatoarelor. Cu toate acestea, limbajele de Asamblare sunt foarte dificil de învățat, mai ales pentru începători, aceste limbaje având o sintaxă dificilă, lipsindu-le abstracția limbajelor de nivel mai înalt, fiind necesară implementarea manuală a mai multor sarcini care sunt gestionate automat în alte limbaje (ex: if, while, for, gestionare funcții), precum și îngreunarea depanării cauzată de lipsa constructelor de nivel înalt. În această lucrare vom prezenta un mediu de simulare pentru algoritmi în limbaj de asamblare, folosind limbajul de programare MyASM. Mediul este conceput pentru a simplifica experiența de învățare, fiind ușor de utilizat pentru scrierea și rularea codului, cu caracteristici precum rularea normală, cu posibilitatea de a fi întreruptă, rularea pas cu pas a instrucțiunilor, vizualizarea regiștrilor, a memoriei și a stivei în cadrul rulării pas cu pas și posibilitatea de a oferi date de intrare și de a vizualiza datele de ieșire. Limbajul de programare MyASM, bazat în mare pe limbajul de asamblare x86, a fost creat cu scopul de a ușura și de a scoate în evidență elementele de bază ale programării în limbaj de asamblare. Implementarea aplicației noastre este construită cu ajutorul .NET Framework, folosind limbajul de programare C#, conținând un modul WinForms pentru interfața vizuală, un modul pentru compilator, și un modul de Unit Testing pentru testarea corectitudinii compilatorului. Deși aplicația nu a fost testată de un public mai larg, credem că are potențialul de a fi un instrument valoros pentru învățarea limbajelor de asamblare.

**Cuvinte cheie:** Assembly, MyASM, Compilator, Simulator

**INTRODUCERE**

Limbajele de Asamblare sunt un tip important de limbaje de programare, fiind folosite pentru a scrie instrucțiuni care pot fi executate direct de procesorul unui computer. Aceste limbaje au o diverse aplicații, precum: programarea *low-level*, control la nivel *hardware*, optimizări de performanță, înțelegerea arhitecturii calculatoarelor și a procesoarelor, inginerie inversă pentru a găsi vulnerabilități de securitate, sau mentenanța sistemelor scrise în acest tip de limbaj. Cu toate acestea, procesul de învățare al unui limbaj de asamblare poate fi un proces dificil, mai ales pentru începători în domeniul programării, acestea având o sintaxă dificilă, precum și lipsa abstracției limbajelor de nivel mai înalt, dar și depanarea îngreunată din cauza acestui lips.

În această lucrare vom prezenta un mediu de simulare pentru algoritmi în limbaj de asamblare, folosind limbajul de programare MyASM. Mediul este conceput pentru a simplifica experiența de învățare, fiind ușor de utilizat pentru scrierea și rularea codului sursă MyASM, cu caracteristici precum rularea normală, cu posibilitatea de a fi întreruptă, rularea pas cu pas a instrucțiunilor, vizualizarea regiștrilor, a memoriei și a stivei în cadrul rulării pas cu pas și posibilitatea de a oferi date de intrare și de a vizualiza datele de ieșire. Limbajul de programare MyASM, bazat în mare pe limbajul de asamblare x86, a fost creat cu scopul de a ușura și de a scoate în evidență elementele de bază ale programării în limbaj de Asamblare. Implementarea aplicației noastre este construită cu ajutorul *Framework*-ului .NET, folosind limbajul de programare C#, folosind mediul de dezvoltare Visual Studio. Aplicația este împărțită în trei module: un modul WinForms pentru interfața vizuală, un modul pentru compilator, și un modul de *Unit Testing* pentru testarea corectitudinii compilatorului.

În capitolele următoare vom prezenta etapele dezvoltării și ciclul de viață al limbajului de programare MyASM, alături de mediul de simulare care permite rularea codului sursă MyASM și vizualizarea regiștrilor și a memoriei operative.

**ASPECTE TEORETICE**

(Assembly, Compilatoare, C#)

**ASPECTE DE PROIECTARE**

(Limbajul MyASM, IDE requirements)

Limbajul de programare MyASM este un limbaj de programare creat cu scopul de a fi ușor de folosit și foarte lizibil. Pentru a realiza acest lucru, sintaxa trebuie să fie intuitivă, ușor de înțeles și ușor de memorat. Limbajul are scopul de a familizariza utilizatorul cu bazele limbajelor de asamblare, astfel trebuind să conțină caracteristicile importante ale mai multor familii de limbaje de asamblare, precum: x86, x64, AMD, ARM, RISC-V, OpenRISC. Aceste limbaje au toate în comun următoarele tipuri de instrucțiuni: *instrucțiuni de memorie*, *instrucțiuni aritmetice*, *instrucțiuni la nivel de biți*, *instrucțiuni de salt* (condiționate sau necondiționate), *instrucțiuni pe stivă* și *instrucțiuni IO*.

În această lucrare am creat un set de instrucțiuni, denumit **MyASM** (de la My Assembly). Limbajul creat este asemănator în cea mai mare măsură cu limbajul **x86**, care conține categoriile de bază ale limbajelor de asamblare enumerate mai sus. O instrucțiune *MyASM* are următoarea formă generală:

eticheta: operatie operand\_1, operand\_2 # comentariu

unde:

* *eticheta* – reprezintă etichetele care se află în dreptul instrucțiunii la care se va sări prin *instrucțiuni de salt* sau prin instrucțiunea de *apel de funcție*, fiind denumite **etichete pentru instrucțiuni**,
* *operația* – reprezintă comanda care trebuie executată de către procesor sau simulator, aceasta poate avea zero, unul sau doi operanzi; comportamentul comenzii poate fi schimbat folosind diferite combinații de tipuri de operanzi,
* *operandul* – reprezintă argumentul comenzii, acestea fiind entitățile pe care / cu care lucrează comanda,
* *comentariul* – reprezintă o explicație lizibilă a codului, fiind ignorată la compilare

**Operanzii** sunt de mai multe tipuri:

* ***Regiștrii generali*** – similar limbajului de asamblare x86, vom avea 4 regiștrii generali: A, B, C, D, cu care se vor executa majoratatea instrucțiunilor.
* ***Valori numerice*** – reprezintă valori constante, pozitive sau negative, și pot fi scrise sub formă zecimală (ex: 12, 23, -4), binară (ex: 0b101, -0b1101), hexadecimală (ex: 0x1A2F, 0xe6a4, -0x123f, -0xEA0) sau sub formă de caracter (ex: ‘a’, ‘B’, ‘2’, ‘ ’, ‘.’), unde se va considera codul ASCII al caracterului. Caractere care **nu** sunt suportate: *‘:’, ‘#’, end-line*; pentru acestea se vor scrie valoarile numerice ale codului ASCII: 0x3A pentru ‘:’, 0x23 pentru ‘#’, iar pentru end-line se vor scrie în ordine 0x0D (“Carriage Return”), urmat de 0x0A (“Line Feed”).
* ***Adrese*** – reprezintă locația în memorie unde sunt salvate date, adresele pot fi *accesate direct* pentru scriere sau pentru citire la acea adresă. Adresa 0x00 se consideră adresa caracterului **null** și nu este permisă suprascrierea valorii numerice ale acestei adrese. Orice definire de date scrisă în codul sursă MyASM se va aloca memorie, prima definire pornind de la adresa 0x01, iar următoarele definiții pornind de la ultima adresă folosită de definirea precedentă plus unu. Adresele se scriu ca valori numerice între **[** și **]**, dar caracterele nu sunt permise (ex: [12], [0x123], [0b1011], ~~[‘C’]~~).
* ***Pointeri*** – similar cu adresele, pointerii reprezintă locația în memorie unde sunt salvate date, dar accesarea este *indirectă*, prin intermediul regiștrilor. Pointerii se scriu ca regiștrii între **[** și **]**, adică: [A], [B], [C] și [D].
* ***Etichete*** – reprezintă adresele unor date salvare în urma unor definiții, fiind denumite **etichete pentru date**. A nu se confunda cu *etichetele pentru instrucțiuni* care se pun în dreptul instrucțiunii și reprezintă zone către care se poate sări, *etichetele pentru date* vor fi scrise ca parametru (operand) pentru instrucțiuni, și sunt considerate ca valori numerice folosite pentru accesarea memoriei.
* ***String-uri*** – folosite doar în cadrul instrucțiunii de definire, oferă posibilitatea de a salva șiruri de caractere în memorie. **Atenție!** Operația de definire NU introduce caracterul *null* la finalul șirului de caractere, acesta trebuie introdus manual, folosind o instrucțiune separată de definire. De examplu:

DEF “Hello World!” # sirul de caractere

DEF 0 # caracterul null

Pentru a defini setul de instrucțiuni, vom folosi următoarele prescurtări pentru tipurile diferite de operanzi: **reg** pentru regiștri, **nr** pentru valori numerice, **adr** pentru adrese, **ptr** pentru pointeri, **lbl** pentru etichete (label-uri), **str** pentru șiruri de caractere (string-uri). Setul de instrucțiuni, împărțit în categorii este următorul:

* **Instructiuni de memorie:**
  + **DEF (Def**ine**)** – permite definirea datelor, și atribuirea de *etichete de date* acestora. Combinații posibile:  
    DEF str, lbl  
    DEF str  
    DEF nr, lbl  
    DEF nr  
    Alternativ operației **DEF** se poate folosi și operația: **DB (D**efine **B**ytes**)**. Exemple:  
    DEF “Hello World!”, hello1  
    DB “Hello World!”, hello2  
    DEF “No label here”  
    DEF 2, myNum  
    DEF 3  
    DB ‘ ’, space  
    DEF ‘C’  
    DB 0x86A, hexa  
    DB 0b1101
  + **MOV (Mov**e**)** – mută în primul operand valoarea din al doilea operand. Combinații posibile:  
    MOV reg, reg  
    MOV reg, nr  
    MOV reg, ptr  
    MOV reg, adr  
    MOV ptr, reg  
    MOV ptr, nr  
    MOV adr, reg  
    MOV adr, nr  
    Exemple:  
    MOV A, B  
    MOV C, 0x12  
    MOV B, [C]  
    MOV D, [23]  
    MOV [A], B  
    MOV [C], -24  
    MOV [0x2A], A  
    MOV [100], 0b11010  
    MOV A, [myNum] # reg, adr  
    MOV A, myNum # reg, nr
* **Instrucțiuni aritmetice:**
  + **ADD** – adună valoarea din primul operand, cu valoarea din cel de-al doilea operand și salvează rezultatul în primul operand. Combinații posibile:  
    ADD reg, reg  
    ADD reg, nr  
    ADD reg, ptr  
    ADD reg, adr  
    Exemple:  
    ADD A, B  
    ADD B, 100  
    ADD C, [A]  
    ADD D, [0x23]  
    ADD A, myNum # reg, nr  
    ADD A, [myNum] # reg, adr
  + **SUB (*Sub****tract***)** -
  + **MULT (*Mult****iply***)** -
  + **DIV (*Div****ide***)** -
  + **MOD (*Mod****ulo***)** -
  + **INC (*Inc****rement***)** -
  + **DEC (*Dec****rement***)** -
  + **NEG (*Neg****ate***)** -
* **Instrucțiuni logice:**
  + **AND** -
  + **OR** -
  + **XOR** -
  + **NOT** -
* **Instrucțiuni de salt:**
  + **Necondiționat:**
    - **JMP (*J****u****mp*)** -
  + **Condiționat:**
    - **JZ (*J****ump if* ***Z****ero***)** -
    - **JNZ (*J****ump if* ***N****ot* ***Z****ero***)** -
    - **JLZ (*J****ump if* ***L****ess than* ***Z****ero***)** -
    - **JLEZ (*J****ump if* ***L****ess than or* ***E****qual to* ***Z****ero***)** -
    - **JGZ (*J****ump if* ***G****reater than* ***Z****ero***)** -
    - **JGEZ (*J****ump if* ***G****reater than or* ***E****qual to* ***Z****ero***)** -
* **Instrucțiuni de setare:**
  + **SZ (*S****et if* ***Z****ero***)** -
  + **SNZ (*S****et if* ***N****ot* ***Z****ero***)** -
  + **SLZ (*S****et if* ***L****ess than* ***Z****ero***)** -
  + **SLEZ (*S****et if* ***L****ess than or* ***E****qual to* ***Z****ero***)** -
  + **SGZ (*S****et if* ***G****reater than* ***Z****ero***)** -
  + **SGEZ (*S****et if* ***G****reater than or* ***E****qual to* ***Z****ero***)** -
* **Instrucțiuni de shiftare**
  + **SHL (*Sh****ift* ***L****eft***)** -
  + **SHR (*Sh****ift* ***R****ight***)** -
* **Instrucțiuni pentru funcții și stivă**
  + **PUSH** -
  + **POP** -
  + **STR (*St****ack* ***R****ead***)** -
  + **STW (*St****ack* ***W****rite***)** -
  + **CALL** -
  + **RET (*Ret****urn***)** -
* **Instrucțiuni IO (Input-Output)**
  + **INP (*Inp****ut***)** -
  + **OUTI (*Out****put* ***I****nteger***)** -
  + **OUTC (*Out****put* ***C****haracter***)** -
* **Alte instrucțiuni:**
  + **HLT (H**a**lt)** -

**ASPECTE DE IMPLEMENTARE**

(MyASM IDE, ciclul compilarii, ciclul rularii, user manual?)

**CONCLUZII**

(nu am nici cea mai mica idee)

**BIBLIOGRAFIE**

(va contine multe link-uri de youtube)