Universitatea Politehnica din Bucuresti

Facultatea de Electronica, Telecomunicatii si Tehnologia Informatiei

Proiect la disciplina

Sisteme de Operare

Tema

Asamblorul si procesele de asamblare

 Studenti:

 Codreanu Radu Adrian

 Gheorghe Alin Valentin

 Grupa 432A

2014-2015

Cuprins :

1.Limbajul de asamblare

1.1 Introducere ( Codreanu Radu Adian )

1.2.1 Instructiuni ( Gheorghe Alin Valentin )

 1.2.2 Ciclul de executie al instructiunilor

 1.2.3 Executia programului

 1.2.4 Legaturi cu limbajele de programare de nivel inalt

1.3 Realizarea unui program de asamblare (Codreanu Radu Adrian)

1.4 Avantajele utilizarii limbajului de asamblare (Gheorghe Alin Valentin)

2. Procesul de asamblare Codreanu Radu Adrian

 2.1 Asamblarea in doua treceri

 2.2 Prima trecere

 2.3 A doua trecere

 2.4 Tabela de simboluri

3. Macroinstructiuni Gheorghe Alin Valentin

 3.1 Extinderi si efecte secundare

 3.2 Procesare macro

 3.3 Exemple de macro-uri

1.Limbajul de asamblare

1.1.Introducere. Codreanu Radu Adrian

Limbajul de asamblare este un limbaj de programare de nivel scazut utilizat pentru calculatoare sau alte dispozitive programabile specifice unei anume arhitecturi, in timp ce limbajele de programare de nivel inalt sunt portabile pentru o multitudine de systeme. Fiecare calculator personal contine un microprocesor care se ocupa de activitatile aritmetice, logice si de control ale calculatorului. Fiecare familie de porceosare are propriul set de instructiuni pentru gestionarea diferitelor operatii cum ar fi: obtinerea de intrari de la tastatura, afisarea de informatii pe ecran sau realizarea altor sarcini. Acest set de intructiuni poarta numele de intructiunile limbajului masina. [1]

Procesorul intelege doar instructiunile limbajul masina, care este format din siruri de 0 si 1. Cu toate acestea, limbajul masina este destul de greoi si complex pentru a fi utilizant in dezvoltarea software. Ca urmare, limabjul de asamblere este realizat pentru o familie specifica de procesoare care presupune o varietate de instructiuni in codul simbolurilor si o forma mai bine de inteles. [1]

Limbajul de asamblare utilizeaze mnemonice pentru a reprezenta intructiunile masina sau operatii ale acestora. Operatiile obisnuite necesita unul sau mai multi operanzi pentru a forma o intructiune completa, majoritatea asambloarelor pot luat etichete, simboluri sau expresii cum ar fi operanzii pentru a reprezenta adrese si alte constante, scutind programatorul de la o serie de calcule manuale. Asambloarele Macro contin macrointructiuni, astfel textul limbajului de asamblare poate fi reprezentat printr-un nume, acel nume putand fii utilizat pentru a introduce textul resprectiv intr-un alt cod. Multe asambloare ofera mecanisme aditionale pentru a usura dezvoltarea de programe, controlul procesului de asamblare si corectarea erorilor. [1]

1.2.1 Instructiuni. Gheorghe Alin Valentin

O instructiune este o declaratie care devine executabila cand un program este asamblat. Instructiunile sunt traduse de asamblor in limbaj masina, care sunt incarcate si executate de Unitatea Centrala de Procesare. [3]

O instructiune contine 4 parti de baza:

-Eticheta (Optional)

-Mnemonica instructiunii (Necesara)

-Operandul (sau mai multi operanzi) (De obicei necesar)

-Comentarii (Optional)

Sintaxa de baza arata in felul urmator: [eticheta:] mnemonic [operand] [;comentariu]

1.2.1.1 Eticheta:

O eticheta este un identificator care tine locul unui marcaj pentru instructiuni si date. O eticheta plasata chiar inaintea unei instructiuni implica adresa instructiunii.

 Etichete de date:

 O eticheta de date identifica locatia unei variabile, furnizand o referinta convenabila a unei variabile in cod.

 Asamblorul atribuie o adresa numerica fiecarei instructiuni.

 Etichete de cod:

 O eticheta in zona de cod a unui program ( unde se afla instructiunile ) trebuie sa se termine cu caracterul " : ". Etichetele de cod sunt utilizate ca target-uri pentru instructiuni de tip JUMP sau LOOP.

1.2.1.2 Mnemonica Instructiunilor

O mnemonica a unei instructiuni este un cuvant scurt care identifica o instructiune. Mnemonicele din limbajul de asamblare cum ar fi MOV, ADD si SUB furnizeaza indicii destre tipul de operatie pe care il efectueaza.

Exemple: [3]

MOV Muta ( Asigneaza ) o valoare

ADD Aduna doua valori

SUB Scade o valoare din alta

MUL Inmulteste doua valori

JMP Salt la o anumita locatie

CALL Apel de procedura

1.2.1.3 Operanzii [3]

Instructiunile in limbaj de asamblare pot avea intre 0 si 3 operanzi, fiecare poate fi un registru, operand de memorie, constanta, port de intrare-iesire.

Exemplu:

 Instructiunea MOV -are 2 operanzi

 MOV AX, BX ;Pune valoarea din registrul BX in registrul AX

 Instructiunea INC -1 operand

 INC AX ;Incrementeaza valoarea registrului AX

1.2.1.4 Comentarii:

 Comentariile reprezinta o cale importanta pentru un programator de a comunica informatii despre design-ul programului unei alte persoane care citeste codul sursa.

1.2.2 Ciclul de executie al instructiunilor

Executia unei singure instructiuni masina poate fi divizata intr-o secventa de operatii individuale numite cicu de executie al instructiunii. Inainte de executare, un program este incarcat in memorie. Instruction pointer-ul ( IP , cunoscut si ca numaratorul de program ) este un registru care retine adresa urmatoarei instructiuni. Instruction queue retine un grup de instructiuni care urmeaza sa fie executate. Executarea unei instructiuni masina necesita 3 pasi de baza: Fetch, Decode si Execute. [2]

Fetch: Unitatea de control aduce urmatoarea instructiune din coada de intructiuni (Instruction Queue) si incrementeaza Instruction Pointer-ul .

Decode: Unitatea de control decodeaza functia instructiunii pentru a determina ce face instructiunea. Operanzii de intrare ai instructiunii sunt trimisi catre ALU (Unitatea Aritmetico-Logica), si se trimit semnale catre ALU indicand operatia care trebuie efectuata.

Execute: ALU executa instructiunea folosind registrele ca operanzi si trimite rezultatul catre un registru sau catre memorie. ALU modifica fanioanele de stare aducand informatii despre starea procesorului.[2]

1.2.3 Cum este executat un program

Urmatoarea secventa de pasi descrie ce se intampla cand un utilizator executa un program:

- Sistemul de operare cauta numele programului in directorul curent.

Daca nu gaseste numele acolo, cauta intr-o lista predeterminata de directoare ( numite paths ). Daca sistemul de operare esueaza in a gasi numele programului, este afisat un mesaj de eroare.

- Daca programul este gasit, Sistemul de operare primeste informatiile de baza despre program , incluzand marimea si adresa fizica a locatiei de pe disc.

- Sistemul de operare determina urmatoarea locatie de memorie disponibila si incarca programul in memorie. Aloca un bloc de memorie programului si inscrie informatia despre marimea programului si locatia in tabel ( tabela de descriptori ). In plus , sistemul de operare poate ajusta valorile pointerilor din program pentru ca acestia sa contina adrese ale datelor de program.

- Sistemul de Operare incepe executia primei instructiuni masina a programului. Imediat ce programul incepe rularea, se numeste inceperea unui proces. Sistemul de operare atribuie procesului un numar de identificare ( process ID ), care este folosit pentru a-l putea urmari in timpul executiei.

- Procesul ruleaza de la sine. Este treaba sistemului de operare sa urmareasca executia procesului si sa raspunda la cereri de resurse de sitem.

- Cand procesul se termina, este inlaturat din memorie.

1.2.4 Legaturi cu limbaje de programare de nivel inalt [1]

Limbajele de programare de nivel inalt au legaturi de tipul one-to-many cu limbajul de asamblare si limbajul masina.

O instructiune in C++ se scrie in multiple instructiuni in limbaj de asambalre sau limbaj masina. Putem exemplifica o astfel de trecere.

Exemplu:

Instructiuni scrise intr-un limbaj de programare de nivel inalt (C++)

...

for( i=10, i>0, i--){

a=a\*2+b;

}

Echivalentul in limbaj de asamblare

 MOV CX, 10h ;Valoarea 10h este pusa in registrul CX (contor- se decrementeaza dupa fiecare intstructiune loop executata.

for: SHL AX, 1 ;Shift-are la stanga a registrului AX cu 1 ( inmultirea cu 2 )

 ADD AX,BX ;Adunam la registrul AX, registrul BX

 LOOP for ;Salt la eticheta for (cat timp CX este diferit de 0 )

Limbajul de asamblare spre deosebire de limbajele de nivel inalt nu este portabil, deoarece este specific unei familii de procesoare. Exista un numar de limbaje de asamblare diferite utilizate astazi, fiecare bazat pe o anumita familie de procesoare, cateva familii bine cunoscute sunt Motorola 68x00, x86, SUN Sparc, Vax si IBM-370.

1.3. Realizarea unui program in assembler Codreanu Radu Adrian

Realizarea unui program in assembler presupune urmatoarele etape:

 1. conceperea algoritmului

 2. scrierea programului in limbajul de asamblare corespunzator procesorului. Programul se scrie in editorul asamblorului. Sintaxa nu este case sensitive insa trebuie tinut cont de indentare intrucat o instructiune care nu este scrisa corespunzator este interpretata ca o eticheta si astfel programul va avea o executie defectuoasa.

 3. traducerea fisierului text care contine programul sursa, in fisiere obiect relocabile utilizand un asamblor.

 4. link-editarea fisierelor obiect intru-un singur fisier obiect executabil, prin reevaluarea atat a legaturilor interne cat si a legaturilor externe. Aceasta etapa de lucru se realizeaza cu ajutorul unui program numit linkeditor. Dupa asamblarea si link-editarea programului se obitne fisierul obj care va fi scris in memoria de program a sistemului. Iesirea asamblorului este redata in partea de jos a editorului asamblorului si putem observa pe prima coloana indicele instructiunii, apoi pe a doua coloana se gaseste adresa de memorie de unde incepe acea instructiune, si pe a treia, a patra se afla codul masina asociat intructiunii respective. Fiecare instructiune a limbajulu de asamblare are un cod de operatie (Opcode) fie in hexazecimal si se poate modifica doar in functie de tipul operanzilor sau de numarul operanzilor. Datorita faptului ca orice intructiune are un cod hexa in limbajul masina, care marcheaza tipul instructiunii si apoi operanzii, se poate observa apoi harta memorie care prezinta cum apare programul scris in memoria de program. Fiecare instructiune este scrisa succesiv in memorie avand un numar de octeti corespunzator.

 5. incarcarea programului in memoria de lucru a microsistemului: Programul asamblat si link-editat se poate incarca in memoria sistemului utilizand comanda load din meniul simulatorului sau al asamblorului. Dupa aceasta operatiune codul masina va aparea in memorie. Dupa incarcarea programului se poate selecta o rata de executie a programului care poate fi pas cu pas ( pentru analiza programatorului ) pana la viteze foarte mari care pot atinge aproape viteza reala a procesorului.

 6. rularea programului incarcat in memorie si evident verificarea rezultatului obtinut. Rularea se realizeaza utiliand comenzile din meniul Simulation al simulatorului si tinand cont de optinuea selectata la Rate. [1]

1.4. Avantajele utilizarii limbajului de asamblare Gheorghe Alin Valentin

 Limbajele de asamblare sunt apropiate de corespondenta unu la unu dintre instructiuni si codul executabil. De asemenea limbajul de asamblare include directive catre asamblor, directive pentru organizarea spatiului de date si macro-uri. Macro-urile pot fi folosite pentru a combina o serie de instructiuni ale limbajului de asamblare intr-un limbaj de nivel inalt. Exista cazuri in care o instructiune este transformata in mai multe instructiuni masina, dar in general instructiunile limbajului de asamblare corespund unei singure instructiuni executabile.

O serie de limbaje de nivel inalt mai vechi aveau o legatura mai apropiata intre instructiunile de nivel inalt si instructiunile limbajului masina. De exemplu, cea mai mare partea a instructiunilor COBOL au fost transferate intr-un mic si evident set de instructiuni masina. Moda de-a lungul timpului a fost ca limbajele de nivel inalt sa creasca nivelul de ambiguitate. Programarea orientata pe obiecte actuala este destul de abstracta, cu toata acestea, o serie de constructori cheie ai programarii orientate pe obiecte se traduc intr-un set compact de instructiuni masina.[1]

Programarea in limbaj de asamblare este mult mai dificila ca programarea intr-un limbaj de programare de nivel inalt. In cazul programarii in limbaj de asamblare, programatorul trebuie sa aiba in vedere mult mai multe detalii si trebuie sa aiba o serie de cunostinte aprofundate cu privire la procesorul pe care lucreaza. Avantajul unui limbaj masina de inalta calitate este acela ca poate rula programe mult mai rapid si are nevoie de foarte putin spatiu de memorie sau alte resurse fata de un program similar scris intr-un limbaj de nivel inalt. Programarea in limbaj de asamblare ofera de asemenea acces direct la o serie de proprietati cheie ale masini, esentiale pentru implementarea mai multor tipuri de rutine de nivel scazut, precum un sistem de oprerare kernel sau microkernel, drivere pentru dispozitive si control al masini.

Limbajele de programare de nivel inalt prezinta o varianta mai usoara de lucru pentru programatorii cu o mai mica experienta si pentru managerii tehnici de a o supraveghea. Tot odata limbajele de nivel inalt ofera un timp mai bun de developare decat limbajul de asamblare, chiar si in cazul unor programatori foarte bine pregatiti. Developarea poate creste de la 10 la 100 de ori. Programele scrise in limbaj de nivel inalt, in special programare orientata pe obiecte, sunt mult mai usoare si destul de ieftine de utilizat decat programe similare scrise in limbaj masina.[1]

2.Procesul de asamblare

2.1.Asamblare in doua treceri Codreanu Radu Adrian

 Asamblorul este un translator, care translateaza un program assembler intr-un program in limbaj masina. Asamblorul parcurge programul secvential, linie cu linie si genereaza cod masina pentru fiecare instructiune in parte. Pentru majoritatea instructiunilor acest proces functioneaza bine, cum ar fi instructiuni care fac referinta la registre, insa instructiunile de tip branch (salturi conditionate/neconditionate, apelari de subprograme,...) determina anumite probleme, deoarece asamblorul nu stie exact unde se afla eticheta la care trebuie sa faca salt , astfel nu stie daca saltul este unul scurt, apropiat sau departat (short, near, far jump). Asamblorul ar trebui sa aprecieze cat de departe este instructiunea pentru a genera instructiunea corecta, dar daca greseste atunci toate adresele pentru toate celelalte etichete din program o sa fie gresite. Solutia pentru aceasta problema o reprezinta asamblarea in doua treceri, astfel codul va fi scanat de doua ori. Prima data, codul va fi scanat pentru a putea afla adresele corecte ale tuturor etichetelor din program, de asemenea va fi creata o tabela care contine o lista a tuturor adreselor si unde vor fi acestea in program. Aceasta tabela se numste tabela de simboluri. A doua trecere (scanare), va genera codul masina, si va utiliza tabela de simboluri pentru a determina cat de departe se afla etichetele in program. [4]

2.2.Prima trecere:

In prima trecere, asamblorul realizeaza urmatoarele functii:

* Se verifica daca instructiunile din modul de asamblare curent sunt legale.
* Se aloca spatiu pentru instructiuni si zone de stocare cerute de programator.
* Unde este posibil, inscrie valorile constantelor.
* Se contruieste tabela de simboluri. [5]

Asamblorul citeste pe rand fiecare line din codul sursa. Daca codul sursa contine un simbol valid, asamblorul presupune ca acel simbol nu a fost inca folosit drept eticheta. Daca este prima data cand simbolul respectiv a fost folosit drept eticheta, asamblorul adauga eticheta respectiva la tabela de simboluri si ii atribuie valoarea numaratorului locatiei curente, simbolului. Daca simbolul a fost deja folosit ca eticheta, asamblorul returneaza un mesaje de eroare ( Redefinition of symbol) si reinitializeaza valoarea simbolului. In continuarea, asamblorul verifica mnemonicul instructiunii. Daca mnemonicul este al unei instructiuni masina care este legala pentru modul curent de asamblare, asamblorul determina formatul instructiunii (XO format). Atunci asamblorul aloca numarul necesar de biti pentru a pastra codul masina pentru instructiunea respectiva. Continutul numaratorului de locatii i se atribuie numarul respectiv de biti.

Cand asamblorul intalneste un comentariu sau un caracter de sfarsit de linie, asamblorul incepe scanarea urmatoarei instructiuni. Asamblorul continua scanarea de instructiuni si construieste tabela sa de simboluri pana cand nu mai exista instructiuni care trebuie citite.

La finalul primei treceri, tot spatiul necesar a fost alocat, si fiecare simbol definit in program a fost asociat cu o valoarea a numaratorul de locatii din tabela de simboluri. Cand s-au epuizat toate instructiunile care trebuiau citite, incepe a doua trecere, care porneste de la inceputul programului.

Daca in timpul primei treceri apare o eroare, procesul de asamblare se incheie si a doua trecere nu mai are loc. Daca acest lucru are loc, listarea asamblorului contine doar erori si avertizari generate in timpui primei treceri a acestuia. [5]

Exemplu:

Programul prezentat mai jos primeste ca parametri de intrare adresa de început şi numărul de elemente pentru un şir de caractere ce conţine numai litere şi returneze numărul de litere mari (prin intermediul registrului DX).

org 100h

Adrese Cod in limbaj de asamblare

100h jmp init

102h sir db 'SiRTesT'

109h init: lea BX, sir Se adauga in tabela 110h cld de simboluri adresele 111h mov DX, 0h efective ale etichetelor/ 112h mov CX, (init-sir) variabilelor declarate 113h mov AX, 0h in program

116h main: push BX 117h push CX 118h call subprog 11Bh add SP,4h 11Eh int 20h

120h subprog: mov BP,SP 122h mov SI, [BP+4h] Instructiunile se 125h mov CX, [BP+2h] parcurg secvential

128h count: lodsb 129h cmp AX, 5Ah 12Ch jbe incrementare

12Eh next: loop count 130h ret

131h incrementare: inc DX 132h jmp next

2.3.A doua trecere

In cea de a doua trecere, asamblorul :

 -Examineaza operanzii pentru referinte simbolice ale locatiilor de stocare si rezolva aceste referinte folosing informatii din tabela de simboluri

 -Se asigura ca nici o instructiune nu are o forma invalida.

 -Translateaza declaratiile sursa in cod masina si constante, astfel umple spatiul alocat cu cod obiect.

 -Produce un fisier care contine mesaje de eroare, daca este necesar.

 La inceputul celei de a doua treceri, asamblorul scaneaza fiecare declaratie a doua oara. Cat timp asamblorul translateaza fiecare instructiune incrementeaza valoarea continuta in numaratorul de locatii.

 Daca se gaseste un simbol in codul sursa, care nu se afla in tabela de simboluri, atunci simbolul nu a fost definit. Acest lucru se poate intampla daca, asamblorul nu a intampinat simbolul in campul de etichete din toate declaratiile scanate in prima trecere. Apoi programatorul trebuie sa corecteze diferitele erori de asamblare. [4]

 Daca se generaza doar avertizari in prima trecere, procesul de asamblare continua in a doua trecere. Listarea asamblorului contine erori si avertizari generate in timpul celei de a doua treceri. Orice avertizare generata in timpul primei treceri nu apare in listarea asamblorului.

2.4.Tabela de simboluri:

Majoritatea asambloarelor sunt asambloare cu doua treceri.

-In prima trecere, orice eticheta gasita in program e pusa in tabela de simboluri alaturi de valorile numerice corespunzatoare. [6]

 jbe incrementare

next: loop count ret

incrementare: inc DX jmp next

!!!Observam etichetele : next , count si incrementare.

 -etichetele de cod - locatia de memorie a instructiunii

 -etichete de variabile - locatia de memorie rezervata

 -constante- valoarea constantei

-In cea de a doua trecere toate etichetele din cod sunt inlocuite cu valorile lor din tabela de simboluri.

 JBE 0131h LOOP 0128h RET INC DX JMP 012Eh



Observam in tabela de simboluri, etichetele prezente in cod, precum si adresele fiecareia.

3.MACRO

Gheorghe Alin Valentin

Macro-urile sunt definite la inceputul unui program sursa, sau sunt scrise in fisiere separate si apoi copiate in program de o directiva INCLUDE.Procesorul citeste macro-ul, scaneaza codul sursa al programului si la fiecare apel al macro-ului in program, asamblorul adauga o copie a codului sursa al macro-ului apelat. Daca un macro este definit de program, dar nu este apelat, codul macro-ului respectiv nu apare in programul compilat. Majoritatea asambloarelor includ suport pentru macrou-uri. Macro-urile furnizeaza moduri de a reprezenta un bloc de text ( cod, date, etc.) printr-un nume (numele Macro-ului) ( mecanism sofisticat de substitutie a textului ) [7].

In vreme ce macro-urile definesc un sir de caractere(care poarta numele de valoare a macro-ului), care va inlocui apelul macro-ului, utilitatea macro-urilor depinde de 3 caracteristici: -Abilitatea lor de a reprezenta un sir de text folosind un sir mai scurt

 -Abilitatea lor de a fi folosite in diferite contexte, lucru ce denota flexibilitate.

 -Efectele secundare.

Folosirea unui macro intr-un program de asamblare presupune 2 pasi:

1.Definirea unui MACRO:

 %\*DEFINE (macro-call)(replacement-pattern)

2.Apelarea unui MACRO:

 %macro\_name (argument1, argument2)

Apeluri de macro instructiuni:

 Un apel de macro ia forma unei directive simbolice care incepe cu numele unui macro definit anterior, urmat de o list de parametri.

 <symbolic directive> : : = <identifier> [ <actual list> ] <actual list> : : = <actual parameter> { , <actual parameter> } <actual parameter> : : = [ <by name> | <by list > | <by value> ]

Primul pas in realizarea unui apel de macro, este de a salva continutul fiecarui parametru, in conformitate cu regulile de trecere ale parametrului asociate tipului parametrului, parametri individuali putand fi omisi. In toate cazurile, numarul parametrilor propriu-zisi trebuie sa fie mai mic sau cel putin egal cu numarul parametrilor formali. Dupa stocarea parametrilor, incepe asamblarea corpului macro-ului. In timp ce corpul macro-ului este in proces de asamblare, toti parametri formali care au fost identificati in momentul in care corpul era definit original se inlocuiesc cu continutul corespunzator al parametrilor actuali.

 Pentru parametrii formali declarati printr-un modul valoare, parametru actual trebuie sa fie o expresie.

< by value > : : = <expression >

Expresia va fi evaluata odata cu apelul macro-ului. Daca rezultatul este o valoare absoluta, reprezentarea in zecimal a valorii, va fi utilizata drept parametru. Daca valoarea nu este absoluta, se va obitine o expresie care asigura evaluarea valorii parametrului actual in timpul apelarii.

 Apeluri de macro nu se vor efectua cand se gasesc erori in lista de parametri. In timpul asamblarii corpului unui macro, alte apeluri de macro-uri pot fi evectuate.

Comparatie intre Macro-uri si Proceduri:

 -Un mare avantaj in folosirea procedurilor este acela ca codul masina pentru grupurile de instructiuni din proceduri, trebuie incarcate in memoria principala o singura data

 -Dezavantajul, cand se folosesc procedurile, il reprezinta nevoia pentru gramada (grupuri de instructiuni)

 -Un macro este un grup de instructiuni care sunt puse intre paranteze rotunde si pe care le denumim la inceputul programului.

 -Folosirea macro-urilor evita timpul de asteptare, care este prezent la proceduri, datorita apelului si returnarii.

 -Dezavantajul este ca astfel programul va avea nevoie de mai multa memorie decat in cazul folosirii procedurilor.

 -Macro-urile sunt preferabile cand se cer sarcini minore, in timp ce procedurile sunt utilizate in general pentru probleme mai serioase.

 Timpul de asamblare:

 Un program care contine macro-uri necesita un timp mai mare pentru asamblare.

 Timpul de executie:

 Un program care contine macro-uri prezinta o viteza mai mare decat un program fara macro-uri.

 Marimea programului:

 Un program ce contine macro-uri este in cele mai multe cazuri mai mare, de vreme ce fiecare apel de macro determina adaugarea codului in program.

Diferenta dintre macrou-uri si subrutine.

-Ambele permit unui grup de instructiuni sa fie definit ca o singura entitate cu o anumita eticheta/nume, unica, apelate cand este nevoie.

-O subrutina este apelata utilizand instructiuni specifice, cat timp un macro este apelat utilizand doar numele acestuia.

-Macrou-urile nu reprezinta un substituent pentru subrutine:

 - Din moment ce un macro este substituit cu codul care constituie corpul macro-ului, macro-urile de dimensiuni foarte mari care sunt folosite de mai multe ori pe parcursul programului vor mari considerabil dimensiunea codului.

 -In acest caz, o subrutina ar fi alegerea mai potrivita, deoarece codul din corpul subrutinei nu este inserat in codul sursa de fiecare data cand este apelat.

-Unitatea Centrala de Procesare furnizeaza suport pentru subrutine ca parte a setului de instructiuni, cat timp suportul pentru macro-uri este o parte din asamblor (similar cu directivele de asamblare).

- Din moment ce un macro poate fi invocat de mai multe ori in acelasi program, este esential sa nu existe conflicte datorate invocatiei multiple.[7]

3.1 Extinderi si efecte secundare

Trebuie facuta cu grija o distinctie intre valoarea unui macro sau functia reprezentata de aceasta si efectele secundare. La momentul apelarii, cand un macro sau o functie este apelata, procesorul macro inlocuieste apelul cu valoarea (un sir ASCII) macro-ului sau functia, precum si efectuarea operatiilor inerente in macro sau functie.

Valoarea lui DEFINE ( functie built-in ) este null string (sir nul ), astfel , cand se apeleaza DEFINE pentru a defini un macro, apelul este inlocuit de sirul nul.[7]

Daca, spre exemplu, programul scris contine mai multe apeluri catre o procedura numita SUBRUTINA, si se doreste sa se introduca/scoata din stiva (PUSH/POP) registrele ES, DS, AX, CX, DX, BX, SI, si DI inainte / dupa call, se poate defini un macro:

%\*DEFINE (APELSUBRUTINA)(

 PUSH ES POP DI

 PUSH DS POP SI

 PUSH AX POP BX

 PUSH CX POP DX

 PUSH DX POP CX

 PUSH BX POP AX

 PUSH SI POP DS

 PUSH DI POP ES

 CALL SUBRUTINA )

 Acum, oriunde va aparea apelul de macro %APELSUBRUTINA in cod, procesorul macro il va inlocui cu sirul de caractere definit, incluzand toate carriage-returns (trecerea dintre randuri), tab-urile si spatiile goale.

 Definitia unui macro incepe cu " %DEFINE ". ( asterixul (\*) este denumit caracterul "call-literally" si inseamna ca nici o expansiune macro nu este ceruta in acest moment. Procesorul macro este in literal mode.)[7]

3.2.Procesarea MACRO

 Procesorul macro, care face parte din Asamblorul MCS-86 , copiaza fisierul sursa intr-un fisier intermediar care urmeaza sa fie asamblat. In timpul procesului de copiere, procesorul macro examineaza fiecare caracter al fisierului sursa pentru a gasi metacaracterul '%'. Cand acest caracter este detectat, procesorul macro stie ca ce urmeaza, se foloseste in procesarea macro. [7]

 Metacaracterul semnalizeaza procesorului macro ca ce urmeaza este :

-Un macro definit de utilizator:

 %\*DEFINE(MACRO(NAME, TYPE, LENGTH)(%NAME D%TYPE %LENGTH DUP(?))

 -Se defineste un macro numit MACRO, cu trei parametri : NAME, TYPE, LENGTH, care, cand este apelat cu argumente valide, se extinde in directive de asamblare DB, DW, sau DD definind un sir cu %LENGTH unitati avand numele %NAME.

-O apelare a unui macro definit de utilizator:

 %MACRO(SIR, W, 100)

 Acest apel este inlocuit de valoarea sa, care, conform cu definitia precedenta este urmatorul sir:

 SIR DW 100 DUP(?)

-O apelare a unei functii built-in:

 %IF(%EQS(%ANSWER, YES))THEN(MACRO(SIR, W, 100)) FI

 Acest apel catre functia built-in IF, evalueaza definitia sirului prezentat mai sus daca valoarea lui ANSWER este egala cu sirul YES, si evalueaza null string in sens contrar. Observam: 1. Daca urmeaza o definire a unui macro, procesorul macro salveaza defnitia

2. Daca urmeaza un apel de macro, procesorul de macro primeste definitia macro-ului apelat , calculeaza valoarea ( un sir ASCII ) si il plaseaza fisierul intermediar. Acest lucru se numeste Extinderea macro-ului.

3. Daca urmeaza un apel catre o functie built-in, procesorul de macro inlocuieste apelul cu valoarea functiei, asemanator cu punctul 2.

3.3 Exemple de macro instructiuni [7]:

 Macro-urile pot fi folosite si pentru rutine. De exemplu, fisierul sursa poate necesita cele 3 variante ale aceluias cod pentru a muta un sir de cuvinte dintr-un segment DS-based, catre un segment ES-based:

 1.Mutarea a 5 cuvinte din TABLE, in FIELD

 MOV CX,5 ; Registru CX contine numarul cuvintelor care trebuie mutate. LEA SI, TABLE ; primul cuvant care trebuie mutat este la DS:TABLE. LEA DI, FIELD ; primul cuvant care trebuie mutat in ES:FIELD. REP MOVSW ; bucla, care dureaza pana CX scade la 0.

 2.Mutarea lungimi cuvintelor din ARRAY[BX], in ADTAB+8

 MOV CX, LENGTH ; Registul CX contine nr de cuvinte ce trebuie mutate LEA SI, ARRAY[BX] ; Cuvant ce trebuie mutat este la DS:ARRAY[BX] LEA DI, ADTAB + 8 ; Primul cuvat ce trebuie mutat in ES:FIELD REP MOVSW ; Bucla care dureaza cat timp CX>0

 3.Mutarea cuvintelor AX din STRUC.WDS, in [BX]

 MOV CX,AX ; Muta contorul in CX LEA SI, STRUC.WDS ;cuvant ce trebuie mutat se afla la DS:STRUC.WDS LEA DI, [BX] ; primul cuvant ce trebuie mutat in ES:[BX] REP MOVSW ; muta pe rand cuvintele pana CX = 0

Prin parametrizarea campurilor formate din cei trei operanzi care difera in cele trei exemple de mai sus, obtinem modelul de inlocuire a macro-ului care trebuie generat in cele trei cazuri:

 MOV CX, COUNT LEA SI, SOURCE LEA DI, DEST REP MOVSW

De unde rezulta un macro care reprezinta forma comuna a celor trei variante:

 %\*DEFINE(MOVE(COUNT, SOURCE, DEST))( MOV CX, %COUNT LEA SI, %SOURCE LEA DI, %DEST REP MOVSW )

Exemplu de definire a unui Macro:

 Display MACRO string1 MOV AH, 09 MOV DX, offset string1 INT 21H ENDM

Exemplu de macro repetitiv:

 Macro-ul REPT este folosit la repetarea unui set de instructiuni si are sintaxa: REPT expresie declaratii ENDM

 Cand asamblorul apeleaza macro-ul, setul de instructiuni continut este repetat de un numar de ori, care este selectat prin intermediul valorii setate in expresie.

Bibliografie:

Limbajul de asamblare:

[1] ＆[http://blog.hit.edu.cn/jsx/upload/ATHYPERLINK "TAssemblyLanguage.pdf%22http://blog.hit.edu.cn/jsx/upload/ATHYPERLINK%20%22http://blog.hit.edu.cn/jsx/upload/AT＆TAssemblyLanguage.pdf%22＆HYPERLINK%20%22http://blog.hit.edu.cn/jsx/upload/AT＆TAssemblyLanguage.pdf%22TAssemblyLanguage.pdf"＆HYPERLINK "TAssemblyLanguage.pdf%22http://blog.hit.edu.cn/jsx/upload/ATHYPERLINK%20%22http://blog.hit.edu.cn/jsx/upload/AT＆TAssemblyLanguage.pdf%22＆HYPERLINK%20%22http://blog.hit.edu.cn/jsx/upload/AT＆TAssemblyLanguage.pdf%22TAssemblyLanguage.pdf"TAssemblyLanguage.pdf](TAssemblyLanguage.pdf%22http%3A//blog.hit.edu.cn/jsx/upload/ATHYPERLINK%20%22http%3A//blog.hit.edu.cn/jsx/upload/AT%EF%BC%86TAssemblyLanguage.pdf%22%EF%BC%86HYPERLINK%20%22http%3A//blog.hit.edu.cn/jsx/upload/AT%EF%BC%86TAssemblyLanguage.pdf%22TAssemblyLanguage.pdf)

[2] <http://web.cse.ohio-state.edu/~dohm/CSE2421/Week11_1.pdf>

[3] <http://docs.oracle.com/cd/E19253-01/817-5477/817-5477.pdf>

Procesul de asamblare:

[4] <http://faculty.cs.niu.edu/~karonis/Classes/01spring/580M/sas/node4.html>

[5] <http://pages.cs.wisc.edu/~smoler/x86text/lect.notes/assembly.html>

[6] <http://users.cis.fiu.edu/~downeyt/cop3402/symbol.htm>

Macros:

[7] <http://ftp.df.lth.se/pub/bitsavers.org/pdf/intel/ISIS_II/9800640-02_MCS-86_Macro_Assembly_Language_Manual_Sep79.pdf>

[8] <http://faculty.cs.niu.edu/~byrnes/csci360/notes/360macro.htm>