ASAMBLOARELE SI PROCESUL DE ASAMBLARE

Plan:

1.Nivelul limbajului de asamblare (Viulet Tiberiu, 442A)

- definitia asamblorului
- limbaj de asamblare
- folosirea limbajului de asamblare
- formatul instructiunilor
- pseudoinstructiuni

2. Macro (Burloiu Grigore, 442A)

- macrodefinitie
- implementarea facilitatii de macrodefinite intr-un asamblor
- implementarea de conditionari in macro-uri

3. Procesul de asamblare (Stanescu Dragos, 442A)

- prima trecere
- a doua trecere
- tabela de simboluri

4. Editarea legaturilor si incarcarea programelor (Nemoianu Irina, 442A)

- activitatile editorului de legaturi
- structura unui modul obiect
- legarea si relocarea dinamica

5. Exemple de asambloare si caracteristicile lor distinctive (Istrate Radu, 443A)

- asamblorul ASEM-51
- MASM
- Netwide Assembler

- TASM
- SunOS 5.x Sparc Assembler

6. Proiectarea unui asamblor simplu (Marin Alexandru, 442A)

- identificarea construcţiilor din codul sursă
- descriere funcţională
- aspecte de implementare şi componente arhitecturale

Nivelul Limbajului de Asamblare

Nivelul limbajului de asamblare difera intr-o masura mare de nivelurile masina corespunzatoare microarhitecturii, ISA si a sistemului de operare. Programele scrise de utilizator sunt convertite din **limbaj sursa** in **limbaj tinta** folosind programe specializate, numite **translatoare**. Daca un procesor este capabil sa execute direct programe scrise in limbaj sursa, traducerea nu mai este necesara.

Traducerea se utilizeaza cand este disponibil un procesor pentru limbajul tinta dar nu si pentru limbajul sursa. Daca se realizeaza corect traducerea, rezultatul va fi acelasi cu rezultatul programului in cod sursa, daca s-ar fi utilizat un procesor corespunzator. Astfel se poate implementa un nou nivel, pentru care nu exista procesor, iar executia programului se va face prin traducerea in limbaj tinta a acestuia.

Exista o diferenta importanta intre traducere si interpretare. In traducere, programul scris original in limbaj sursa nu este direct executat. In schimb, acesta este convertit intr-n program echivalent, program obiect (binar), executabil a carui executie este realizata dupa ce traducerea s-a terminat. In cazul traducerii executia are loc in doua etape succesive:

1. Generarea programului echivalent in limbaj tinta

2. Executia programului generat

In cazul interpretarii exista o singura etapa: executia programului sursa initial. Asadar, generarea unu program echivalent nu mai este necesara, desi cateodata, programul sursa este convertit intr-o forma intermediara (ex: cod binar Java) pentru o mai usoara interpretare.

In timpul executiei programului obiect sunt implicate numai trei niveluri: nivelul microarhitecturii, nivelul ISA si nivelul sistemului de operare al masinii. In consecinta, in faza de executie se gasesc trei programe in memoria calculatorului: programul obiect, sistemul de operare si microprogramul (daca acesta exista). La executie, toate urmele programului sursa dispar. In acest mod, numarul de niveluri prezente in momentul executiei poate sa diefere de numarul de nivele prezente inainte de traducere.

Introducere in limbajul de asamblare

In functie de relatia dintre limbajul sursa si limbajul tinta, translatoarele se clasfica in doua categorii: **asambloare** si **compilatoare**. Cand limbajul sursa este o reprezentare simbolica pentru un limbaj masina numeric, translatorul se

numeste **asamblor** iar limbajul sursa este numit **limbaj de asamblare**. Cand limbajul sursa este de nivel inalt, ca Java sau C si limbajul destinatie este fie un limbaj masina numeric fie o reprezentare simbolica pentru astfel de limbaj, translatorul se numeste **compilator**.

Limbaje de asamblare

Intr-un limbaj de asamblare, fiecare instructiune produce o singura instructiune masina. Exista, astfel, o corespondenta *unu la unu* intre instructiunile masina si instructiunile din programul de asamblare. Desi nu pare o mare diferenta intre cele doua limbaje, aceasta devine evidenta daca se ia in calcul ca limbajul masina numeric este scris in hexazecimal pe cand limbajul de asamblare foloseste nume si adrese simbolice, mult mai usor de tinut minte decat reprezentarile octale. Asamblorul se ocupa de traducerea numelor simbolice si a adreselor simbolice in instructiuni masina respectiv adrese numerice.

O alta proprietate importanta a limbajelor de asamblare este aceea ca programatorul are acces la *toate* facilitatile si instructiunile disponibile pe masina tinta, facilitati pe care programatorul in limbaje de nivel inalt nu le are. De exemplu, daca masina tinta are un bit care testeaza depasirea superioara, un program scris in limbaj de asamblare poate testa acel bit direct, pe cand un program scris in Java nu are asemenea posibiltate. Astfel, tot ce se poate face in limbajul masina se poate face si in limbajul de asamblare. Limbajele avansate de nivel inalt, cum ar fi C, sunt de multe ori o combinatie intre aceste doua tipuri de limbaje de programare.

Totusi, un program scris in limbaj de asamblare este specific masinii pentru care a fost scris, pe cand un program scris intr-un limbaj de nivel inalt poate fi executat pe mai multe masini (proprietatea de portabilitate).

Ratiunea uilizarii limbajului de asamblare

Desi scrierea programelor in limbaj de asamblare este dificila, inceata si intretinerea acestora este mai greoaie, limbajul de asamblare ofera doua avantaje importante: **performanta** si *accesul la masina*.

Un programator expert in limbaj de programare va putea, in majoritatea cazurilor, sa produca un cod mai scurt si mai rapid decat daca ar fi folosit un limbaj de nivel inalt. In multe cazuri, viteza si dimensiunea sunt critice si de aceea limbajul de asamblare se foloseste la dezvoltarea procedurilor din BIOS, driverelor de echipamente, codul cartelelor inteligente etc.

In al doilea rand, anumite proceduri au nevoie de acces complet la hardware (exemplu, tratarea intreruperilor si a capcanelor de nivel scazut intr-un sistem de operare).

Cresterea performantelor utilizand limbajul de asamblare

In majoritatea programelor, doar un procent mic din tot codul utilizeaza o parte importanta din intregul timp de executie. Uzual, doar 1% din program dureaza 50% din timpul de executie si 10% din program utilizeaza 90% din timpul de executie.

Daca se presupune ca este necesar pentru 10 programaturi/an sa scrie un program intr-un limbaj de programare de nivel inalt, program ce are nevoie de 100 de secunde pentru a executa un task de benchmark, atunci scrierea

programului in limbaj de asamblare ar necesita 50 de programatori/an, dar acelasi program ar putea executa taskul de benchmark in 33 de secunde deoarece un programator inteligent poate fi de 3 ori mai eficient ca un compilator.

In practica se alege o solutie de mijloc. Programul este scris initial intr-un limbaj de nivel inalt si se analizeaza care parte din program consuma cel mai mult timp de executie (prin utilizarea ceasului de sistem pentru a determina timpul consumat de fiecare procedura, contorizarea numarului de executii ale ciclurilor etc). Portiunea critica de 10% care utilizeaza 90% din timpul de rulare al programului se va imbunatati prin rescriere in limbaj de asamblare. Acest proces se numeste tuning.

Exemplu:

	Nr. de programatori	Timpul de executie
Limbaj de asm.	50	33
Limbaj de niv. Inalt	10	100
Inainte de tuning		
Critic 10%	1	90
Altele 90%	9	10
Total	10	100
Dupa tuning		
Critic 10%	6	30
Altele 90%	9	10
Total	15	40

Daca se compara in exemplul de mai sus combinatia limbaj de nivel inalt / asamblare cu versiunea scrisa exclusiv in limbaj de asamblare se observa ca versiunea a doua este cu 20% mai rapida dar mai mult de trei ori mai scumpa. Mai mult, rescrierea unei proceduri in limbaj de asamblare, dupa ce aceasta a fost deja scrisa si depanata intr-un limbaj de nivel inalt este mai rapida decat conceperea acelei proceduri de la zero. De asemenea, un programator ce utilizeaza un limbaj de programare de nivel inalt capata o viziunea de ansamblu a problemei ce permite imbunatatirea performantelor. Devin astfel evidente avantajele aplicarii metodei combinate de concepere a programelor.

Rezultatele prezentate in exemplul de mai sus au o puternica sustinere practica, in urma mai multor experimente efectualte de Graham (1970) si Corbato (1969).

Succesul multor proiecte depind de posibilitatea introducerii unei imbunatatiri a performantei cu un factor de 2 sau 3 pentru o procedura critica si deci, cunoasterea limbajului de asamblare este importanta. De asemenea, se poate ajunge in situatii in care memoria este insuficienta si trebuie din nou apelat la programare in limbaj de asamblare (este cazul cartelelor inteligente, a procesoarelor incorporate in aparatura electronica obisnuita etc). Un compilator insusi realizeaza functia de asamblare si programatorulului trebuie sa ii fie familiar acest limbaj. In fine, cunoasterea unei masini este evidentiata de limbajul de asamblare folosit de aceasta. Pentru o intelegere completa a unei masini este necesara intelegerea completa a limbajului de ansamblare al acesteia.

Formatul unei instructiuni de asamblare

Structura unei instructiuni in limbaj de asamblare se aseamana foarte mult cu structura instructiunii masina pe care o reprezinta. Cu toate acestea, diferitele limbaje de anasamblare au suficiente asemanari pentru a se putea face o discutie generala.

Exemplu:

Mai jos sunt prezentate fragmente de program in limbaj de asamblare pentru Pentium II, Motorola 680x0 si Ultra SPARC, care realizeaza calculul N = I + J. Instructiunile care apar inainte de linia libera sunt cele care realizeaza calculul in sine, celelalte fiind instructiuni pentru ansamblor pentru rezervarea de memorie.

Pentium II

Eticheta	Cod Operatie	Operanzi	Comentarii
Formula	MOV	EAX , I	;registrul EAX = 3
	ADD	EAX , J	;registrul EAX = EAX + (J)
	MOV	N,EAX	; (N) = (I) + (J)
I	DW	3	;rezerva 4B initializati cu 3
J	DW	4	;rezerva 4B initializati cu 4
N	DW	0	;rezerva 4B initializati cu 0

Motorola 680x0

Eticheta	Cod Operatie	Operanzi	Comentarii
Formula	MOVE.L	I , D0	; (D0) = (I)
	ADD.L	J , D0	; (D0) = (D0) + (J)
	MOVE.L	D0 , N	; (N) = (D0)
I	DC.L	3	;rezerva 4B initializati cu 3
J	DC.L	4	;rezerva 4B initializati cu 4
N	DC.L	0	;rezerva 4B initializati cu 0

SPARC

Eticheta	Cod Operatie	Operanzi	Comentarii
FORMULA:	SETHI	%HI(I) , %R1	! R1 = bitii superiori din adresa I

	LD	[%R1+%LO(J)] , %R1	! R1 = J
	SETHI	%HI(J) , %R2	! R2 = bitii superiori din adresa J
	LD	[%R2+%LO(J)] , %R2	! R2 = J
	NOP		! Asteapta J din memorie
	ADD	%R1 , %R2 , %R2	! R2 = R1 + R2
	SETHI	%HI(N) , %R1	! R1 = bitii superiori din adresa N
	ST	%R2 , [%R1+ %LO(N)]	
I:	.WORD 3		;rezerva 4B initializati cu 3
J:	.WORD 4		;rezerva 4B initializati cu 4
N:	.WORD 0		;rezerva 4B initializati cu 0

Pentru Intel exista mai multe ansambloare diferite. In exemplul de mai sus s-a folosit limbajul MS MASM. Observatiile sunt valabile si pentru 386, 486, Pentium si Pentium Pro si versiuni ulterioare. Pentru SPARC s-a utilizat ansamblorul Sun. Observatiile sunt valabile si pentru versiunile anterioare. Sun utilizeaza caractere mici, dar pentru uniformitate s-au utilizat in exemplu caracterele mari.

Fiecare operatie are 4 campuri: **eticheta operatiei**, **codul operatiei**, **operanzi** si **comentarii**.

Etichetele se folosesc pentru a da nume simbolice adreselor din memorie sau referirea la date prin nume simbolice si permit, de asemenea satul la instructiuni. Micile diferente de sintaxa, cum ar fi necesitatea de a pune “ : “ dupa fiecare eticheta pentru limbajul Sun, depinde de gusturile celor ce au creat limbajul si nu releva nicio particularitate arhitecturala.

Fiecare masina are registre cu nume diferite. Pentru Pentium II acestea pot fi: EAX, EBX, ECX s.a. Motorola are registre denumite D0, D1, D2 s.a. SPARC are nume multiple pentru registrele sale.

Codul operatiei este o prescurtare simbolica reprezentativa pentru operatia desfasurata de instructiunea masina. Alegerea codului este tot o chestiune de gust. De exemplu, pentru Intel s-a preferat utilizarea MOV pentru incarcarea unui registru din/in memorie, pe cand proiectantii ansamblorului Motorola s-a ales MOVE pentru amble cazuri. Proiectantii limbajului Sun pentru SPARC au ales sa foloseasca LD pentru incarcarea unui registru din memorie si SD pentru incarcarea in memorie a unui registru.

Necesitatea utilizarii a doua instructiuni masina, incepand cu SETHI, pentru a acesa memoria, este o proprietate inerenta a arhitecturii SPARC, pentru ca adresele virtuale au 32b pentru SPARCv8 sau 44b pentru SPARCv9 iar instructiunile pot sa contina cel mult 22b de date imediate. Deci, vor fi necesare

doua instructiuni pentru a furniza toti bitii unei adrese virtuale complete:

SETHI %HI(I),%R1

pune pe zero cei mai semnificativi 32b si cei mai putini semnificativi 10b ai registrului R1(R1 are 64b), dupa care pune cei mai semnificativi 22b ai adresei variabilei I in pozitiile de bit de la 10 la 31 ai registrului R1.

LD [%R1+%LO(I)],%R1

aduna R1 si ceimai putin smnificativi 10b ai adresei I pentru a obtine adresa completa a lui I, aduce cuvantul din memorie si il pune in R1. Desi dificila, instructiunea este rapida.

Familiile Pentium, 680x0 si SPARC admit operanzi ce se executa pe B, pe W sau pe DW. Pentru ca ansamblorul sa isi dea seama ce lungime sa utilizeze s-au ales diverse solutii. Pentru Pentium II, registrele de lungime diferita au nume diferite: EAX este utilizat pentru a mult elemente pe 32b, AX pentru elemente pe 16b, AL si AH pentru elemente pe 8b.

Proiectantii Motorola au adoptat sufixul .L pentru cuvant lung, .W pentru cuvant (word) si .B pentru octet. SPARC utilizeaza coduri de operatie diferite pentru lungimi diferite (ex: LDSB, LDSH sau LDSW). Cele trei abordari sunt arbitrare si releva libertatea de care se bucura proiectantii de ansambloare.

Cele trei ansambloare difera si prin modul in care se rezerva spatii pentru date. Pentru Intel s-a ales DW (define word) si s-a adaugat .WORD ca alternativa. Pentru Motorola se foloseste DC (define constant) iar Sun utilizeaza .WORD.

Campul de operanzi se utilizeaza pentru a specifica adresele si registrele folosite ca operanzi de instructiunea masina. Pentru o instructiune de anunare intreaga, campul de operanzi indica ce cu ce se aduna. Campul de operanzi pentruo instructiune de salt indica eticheta la care se face saltul.

Campul de comentarii ofera programatorului optiunea de a-si comenta codul. Fara comentarii, un program in limbaj de asamblare este foarte dificil de inteles.

Pseudoinstructiuni

Pe langa specificare instructiunilor masina care trebuie executate, un program in limbaj de asamblare trebuie sa contina si comenzi pentru ansamblor, cum ar fi cerea pentru alocarea de memorie. Acestea se numesc pseudoinstructiuni sau directive de ansamblare. In sectiunea anterioara, deja s-a folosit o pseudoinstructiune: DW.

exemple de pseudoinstructiuni pentru MS MASM

Pseudoinstr.	Semnificatie
SEGMENT	Incepe un nou segment (text, date..) cu anumite atribute
ALIGN	Controleaza alinierea urmatoarei instructiuni sau date
EQU	Defineste un simbol nou egal cu o expresie data
DB	Aloca memore pentru unu sau mai multi B (initializati)
PROC	Incepe o procedura
ENDP	Incheie o procedura
MACRO	Incepe o macrodefinitie

ENDM	Incheie o macrodefinitie
INCLUDE	Include un fisier
IF	Incepe o ansamblare conditionata pe baza unei expresii date

Pseudoinstructiunea SEGMENT incepe un segment nou, iar ENDS termina un segment. Este permis sa se inceapa un segment de text de cod apoi sa se inceapa un segment de date si apoi sa se revina la segmentul de text s.a.

ALIGN forteaza ca urmatoarea linie, de obicei de date, sa inceapa la o adresa care este multiplu al argumentului sau. De exemplu, daca segmentul curent are deja 61B de date, atunci dupa ALIGN 4, urmatoare adresa alocata va fi 64.

EQU este utilizat pentru a da un nume simbolic unei expresii. De exemplu:

```
BASE    EQU    1000
```

simbolul BASE poate fi utilizat oriune in loc de valoare 1000.

DB, DD, DW si DQ aloca memorie pentru una sau mai multe variabile ocupand 1, 2, 4, sau 8 B. Exemplu:

```
TABLE DB 11, 23, 49
```

aloca spatiu pentru 3B pe care ii initializeaza cu 11, 23 si respectiv 49.

Pseudoinstructiunea INCLUDE produce includerea de catre ansambor a unui alt fisier in fisierul curent. Astfel de fisiere contin de obicei definitii, macro definitii si alte elemente necesare in mai multe fisiere.

Multe ansambloare suporta ansamblarea conditionata (MASM nu face exceptie). De exemplu:

```
WORDSIZE EQU 16
IF WORDSIZE GT 16
WSIZE:    DW   32
ELSE
WSIZE:    DW 16
ENDIF
```

se aloca un cuvant de 32b si eticheteaza adresa sa cu WSIZE. Cuvantul respectiv este initializat cu 32 sau 16, in functie de valoarea lui WORDSIZE. In acest caz, cu 16.De obicei, astfel de instructiune se foloseste pentru a putea fi ansamblata si pe o masina de 16b si pe una de 32b. Astfel, programul se poate utiliza pe mai multe masini tinta.

Nici pseudoinstruciunile nu sunt dictate de arhitectura masinii care le foloseste!

- Bibliografie:

Organizarea Structurata a Calculatoarelor – Andrew S. Tanenbaum (editia a IV-a)

MACRO

In cadrul programarii in limbaj de asamblare apare adesea nevoia de a refolosi anumite secvente de instructiuni in cadrul aceluiasi program. Pentru a evita scrierea acestora de cate ori este nevoie, se poate transforma secventa in procedura, care va fi apelata de fiecare data.

O alta solutie, care evita incetinirea programului cauzata de apelarea de proceduri, este folosirea de macro-uri, care inlocuiesc secventa de instructiuni cu o notatie aleasa de utilizator. Astfel este posibila programarea modulara, cu macro-uri definite la inceput, urmate de o formulare mai compacta a programului.

Macrodefinitie, apel si expandare

O macrodefinitie este un mod de a da un nume unei secvente de text. Odata definita, ea va functiona ca prescurtare pentru respectivul text. In urmatorul exemplu, scris in limbaj de asamblare pentru Pentium II, putem observa cum un macro poate fi folosit pentru a schimba (de doua ori) continutul a doua variabile intre ele.

(a)

```
MOV  EAX,P
MOV  EBX,Q
MOV  Q,EAX
MOV  P,EBX

MOV  EAX,P
MOV  EBX,Q
MOV  Q,EAX
MOV  P,EBX
```

(b)

```
SWAP   MACRO
 MOV  EAX,P
 MOV  EBX,Q
 MOV  Q,EAX
 MOV  P,EBX

    ENDM
    SWAP
    SWAP
```

In cazul (a) avem interschimbarea repetata in varianta clasica, in vreme ce in cazul (b) primele 6 randuri sunt rezervate definirii macro-ului, urmand ca interschimbarea repetata sa fie programata in doar 2 randuri, apeland de doua ori macro-ul `SWAP`.

O macrodefinitie contine urmatoarele trei componente de baza:

- antetul, care da numele macro-ului
- corpul macro-ului – in cazul nostru, secventa de 4 instructiuni
- o pseudoinstructiune ce marcheaza sfarsitul definitiei – aici, `ENDM`.

La fiecare macrodefinitie intalnita, asamblorul o va memora intr-o tabela de definitii, de unde ea va fi apelata la fiecare aparitie a macro-ului in corpul programului. Utilizarea macroului drept cod de operatie se numeste **apel de macro**, iar inlocuirea sa cu corpul macrodefinitiei se numeste **macroexpandare**.

Macroexpandarea intervine in timpul procesului de asamblare; in cod masina nu se observa diferente intre un program in care s-au folosit macro-uri si unul in care nu s-au folosit.

Macro-urile sunt deci doar un artificiu de programare – nu trebuiesc confundate cu apelurile de procedura, care apar in programul obiect si sunt executate odata cu programul. Conceptual, putem considera procesul de asamblare ca fiind executat in doua treceri: in prima au loc toate macroexpandarile, iar in a doua textul rezultat este prelucrat in mod normal de catre asamblor.

Trebuie insistat ca asamblorul nu este interesat de intelesul textului din corpul macrodefinitiei, iar design-ul procesorului de macro-uri este in principiu independent de masina de calcul folosita.

Macrodefinitii cu parametri

Cand un program contine mai multe secvente de instructiuni asemanatoare dar nu perfect identice, se pot folosi macro-uri cu parametri. In urmatorul exemplu, prima secventa interschimba P si Q iar a doua R si S.

(a)

```
MOV  EAX,P
MOV  EBX,Q
MOV  Q,EAX
MOV  P,EBX

MOV  EAX,R
MOV  EBX,S
MOV  S,EAX
MOV  R,EBX
```

(b)

```
CHANGE   MACRO P1,P2
     MOV  EAX,P1
     MOV  EBX,P2
     MOV  P2,EAX
     MOV  P1,EBX
         ENDM

     CHANGE  P,Q
     CHANGE  R,S
```

Observam cum **parametrii formali** (`P1`,`P2`) ai macrodefinitiei sunt utilizati ca parametri actuali odata cu apelarea macro-ului `CHANGE` si efectuarea macroexpandarii. Inca o data, programele (a) si (b) produc rezultate identice.

Implementarea facilitatii de macrodefinitii intr-un asamblor

Pentru a implementa facilitatea de macrodefinitie, un asamblor trebuie sa poata realiza functiile de memorare a macrodefinitiilor si de macroexpandare.

Memorarea se face cu ajutorul unei tabele de nume de macrodefinitii, memorata de asamblor. In tabela sunt memorati si indicatori catre definitiile respective, care ajuta la obtinerea definitiei cautate.

La fiecare intalnire a unei macrodefinitii se construieste o noua intrare in tabela – aceasta va contine numele respectiv, numarul de parametri formali, si indicatorul catre tabela macrodefinitiilor, in care se pastreaza corpul macrodefinitiei. In timpul memorarii corpului, parametrii formali sunt marcati prin simboluri speciale, cum ar fi caracterul ampersand (&). Iata un exemplu de reprezentare interna a macrodefinitiei `CHANGE`:

```
MOV EAX,&P1;MOV BX,&P2;MOV &P2,EAX;MOV &P1,EBX;
```

Asamblarea se poate face in una sau doua treceri, dupa cum vom vedea in cele ce urmeaza.

- **Asamblarea intr-o singura trecere**

Aceasta are loc cand asamblorul altereneaza intre definirea si expandarea macro-urilor in mod recursiv. Aceasta metoda este posibila presupunand ca toate macro-urile ar trebui sa fie definite in cadrul trecerii prin program, inaintea oricarei macroexpandari – in plus, corpul unei macrodefinitii ar putea contine definitii are altor macro-uri.

Aceasta restrictie nu pune nicio problema reala programatorului, fiindu-i usor sa defineasca orice macro inainte de a-l apela in cadrul programului.

Structura asamblorului in acest caz consta in trei tabele:

- **DEFTAB**, contine macrodefinitiile, inclusiv corpurile acestora
- **NAMTAB**, retine numele macro-urilor, are rol de index al DEFTAB, folosind pointeri catre ambele capete ale DEFTAB
- **ARGTAB**, contine parametrii (argumente) apelati, in ordinea pozitiei lor din lista de parametrii; odata cu expandarea macro-ului, parametrii actuali din ARGTAB se substituie parametrilor formali din corpul macrodefinitiei.

- **Asamblarea in doua treceri**

In acest caz, intai se cauta codurile de operatie si se expandeaza macrodefinitiile, care vor fi memorate in tabela de macrouri. Apoi, la fiecare apelare de macro, asamblorul va citi direct din tabela. De asemenea, parametrii

formali din corpul macroului vor fi inlocuiti cu parametrii actuali apelati in cadrul programului.

La asamblarea in doua treceri, definirile imbricate de macro-uri nu sunt permise.

Implementarea de conditionari in macro-uri

Macro-urile pot contine si structuri de forma IF-ELSE-ENDIF sau WHILE-ENDW. In cazul intalnirii unor asemenea instructiuni in timpul macroexpandarii, asamblorul va proceda in urmatoarele feluri:

- Structuri **IF-ELSE-ENDIF**.

Asamblorul trebuie sa aiba o tabela care sa contina valorile tuturor variabilelor temporale folosite. Aceste valori se schimba cand are loc procesarea de comenzi SET, si sunt verificate oricand e necesara aflarea starii temporale curente.

La intalnirea instructiunii IF, se evalueaza expresia asociata:

  - TRUE: Asamblorul continua sa proceseze randuri din DEFTAB pana cand intalneste urmatoarea instructiune ELSE sau ENDIF. Daca intalneste ELSE, sare la ENDIF.
  - FALSE: Asamblorul sare inainte in DEFTAB pana gaseste urmatoarea instructiune ELSE sau ENDIF.

- Structuri **WHILE-ENDW**

Cand o instructiune WHILE este intalnita in timpul macroexpandarii, se evalueaza respectiva expresie:

  - TRUE: Asamblorul continua sa proceseze randuri din DEFTAB pana cand intalneste urmatoarea instructiune ENDW. La atingerea ENDW, procesarea revine la WHILE, re-evalueaza expresia booleana, si actioneaza din nou in functie de noua valoare.
  - FALSE: Asamblorul sare inainte in DEFTAB pana la urmatoarea ENDW, si apoi continua expandarea in mod normal.

Caz particular: preprocesorul NASM

NASM, contine un puternic procesor de macro-uri, ce suporta asamblarea conditionata, includerea de fisiere multi-nivel, doua forme de macro (“single-line” si “multi-line”), si un mecanism “stiva de context”, care permite o folosire mai puternica a macro-urilor.

Toate directivele preprocesorului incep cu semnul `%`.

- Macro-uri “single-line”

Definitiile acestor macro-uri ocupa o singura linie, si se fac folosind directiva `%define`. Ele functioneaza intr-un mod similar limbajului C, dupa cum se poate vedea in urmatorul exemplu:

```
%define ctrl 0x1F &
%define param(a,b) ((a)+(a)*(b))
          mov byte [param(2,ebx)], ctrl   D
```

care expandat va arata:

```
          mov byte [(2)+(2)*(ebx)], 0x1F &   D
```

Cand expandarea unui macro single-line contine nume ale altor macro-uri, expandarea are loc in momentul invocarii, nu al definirii. Astfel, codul

```
%define a(x)   1+b(x)
%define b(x)   2*x
     mov ax,a(8)
```

va fi evaluat ca `mov ax,1+2*8`, desi macro-ul `b` nu fusese inca definit la momentul definitiei lui `a`.

Macro-urile single-line pot fi supraincarcate, de exemplu, in urma definitiilor

```
%define foo(x) 1+x
%define foo(x,y) 1+x*y
```

preprocesorul va putea aborda ambele tipuri de invocare a macro-ului, numarand parametri apelati.

Definitiile pot fi anulate prin comanda `%undef`.

De asemenea, in cazul in care un macro single-line nu are parametri si are o valoare numerica, atunci acesta poate fi definit prin comanda `&assign`.

- Macro-uri “multi-line”

O definitie de macro multi-line arata astfel:

```
%macro prologue 1
          push ebp
          mov ebp, esp
          sub esp,%1
%endmacro
```

Astfel a fost definita o functie `prologue` ca un macro. Numarul `1` dupa numele macro-ului defineste numarul de parametri asteptati. Folosirea `%1` in interiorul macrodefinitiei se refera la primul parametru apelat. In cazul folosirii mai multor parametrii, acestia vor fi referiti ca `%2`, `%3` etc.

Macro-urile multi-line pot fi supraincarcate similar cu cele single-line.

De asemenea, preprocesorul NASM mai ofera si alte facilitati pentru aceste macro-uri, printre care: definirea de etichete in interiorul unei macrodefinitii, parametri “greedy”, parametri impliciti, numarator de parametri `%0`, rotire a parametrilor `%rotate`, concatenare a acestora, si folosirea de coduri conditionale ca parametri.

Bibliografie

- Organizarea Structurata a Calculatoarelor – Andrew S. Tanenbaum (editia a IV-a)
- Assembler Design – Prof. S. Seema, MSRIT
- Preprocesorul NASM - documentatie

Procesul de asamblare:

In sectiunile urmatoare vom descrie pe scurt cum functioneaza un asamblor. Desi fiecare masina are un alt limbaj de asamblare , procesul de asamblare este suficient de asemanator pt masini diferite astfel incat sa fie posibila descrierea sa in termen igenerali. [D1]

Vom definii insa inainte cateva concepte: asamblor: program ce traduce un limbaj de asamblare in cod masina; limbaj masina: este un limbaj format numai din instuctiuni codate binar ce sunt folosite diret de calculator; limbajul de asamblare: este limbajul de programare low level in care o mnemonica reprezinta fiecare din instructiunile limbaj masina ale calculatorului [D2]

Asambloare cu 2 treceri

Deoarece un program in limbaj de asamblare consta intr-o serie de instructiuni scrise pe o linie, ar parea natural sa avem un asamblor care citeste cate o instructiune, o introduce intr-un limbaj masina, si in final genereaza intr-un fisier codul in limbaj masina si eventual, in alt fisier listingul corespunzator. Procesul se repeta pana cand intregul program a fost tradus. [D1]

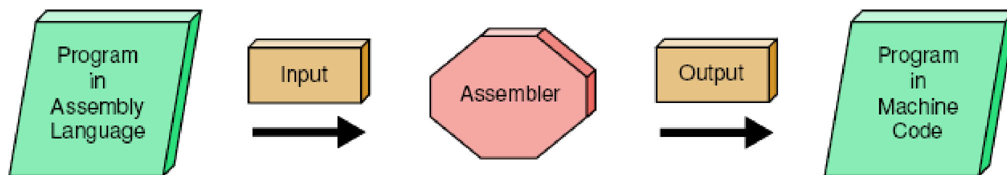

Figure 7.5 Assembly process

[D2]

Din pacate aceasta strategie nu functioneaza.

Sa consideram cazul in care prima instructiune este un salt la L. Asamblorul nu poate sa asambleze aceasta instrunctiune pana cand nu stie adresa instructiunii L. Instructiunea L poate sa fie aproape de sfarsitul programului, facand imposibil pentru asamblor sa gaseasca adresa fara sa citeasca intreg programul. Aceasta dificultate se numeste **problema referintei inainte (forward referince problem)**, deoarece simbolul L a fost utilizat inainte de a fi definit; adica s-a facut o referinta la un simbol care poate sa fie definit ulterior. [D1]

Referintele inainte pot sa fie tratate in 2 moduri. In primul rand, asamblorul poate de fapt sa citeasca programul sursa de 2 ori. Fiecare citire a programului sursa se numeste **trecere**; orice translator care citeste programul sursa se numeste **translator cu 2 treceri**. In prima trecere a asamblorului, definitiile simbolurilor, inclusiv etichetele de instructiuni, sunt corectate si memorate intr-o tabela. Cand incepe a 2-a trecere, valorile tuturor simbolurilor sunt cunoscute; nu mai exista referinte inainte si fiecare instructiune este citita, generata si scrisa. Desi aceasta abordare presupune o trecere suplimentara asupra intrarii, este simpla din punct de vedere conceptual. [D1]

A doua abordare consta dintr-o singura citire a programului in limbaj de asamblare, convertirea sa intr-o forma intermediara, si memorarea formei

intermediare intr-o tabela in memorie. A doua trecere se executa asupra tabelelor si nu asupra programului sursa. Daca exista memorie suficienta (sau memorie virtuala), aceasta abordare economiste timp de I/O. Daca trebuie sa se produca un listing, atunci toata instructiunea sursa inclusiv toate comentariile trebuie sa fie salvate. Daca nu este necesar un listing atunci forma intermediara poate sa fie redusa la ce este esential. [D1]

In ambele cazuri, o alta sarcina a primei treceri este sa salveze toate macro definitiile si sa expandeze apelurile atunci cand sunt intalnite. In acest mod definirea de simblori si expandarea de macro-uri sunt in general combinate in aceeasi trecere. [D1]

Prima trecere

Principala functie a acestei treceri este sa construiasca o tabela numita **tabela de simboluri**, continand valorile tuturor simbolurilor. Un simbol este fie o eticheta sau o valoare careia i s-a atribuit un nume simbolic prin intermediul unei pseudoinstructiuni de tipul:

```
BUFSIZE     EQU   8192
```

In atribuirea unei valori, unui simbol in campul eticheta a unei instructiuni, asamblorul trebuie sa stie ce adresa va avea instructiunea in timpul executiei programului. Pentru a pastra evidenta adreselor din timpul executiei, asamblorul pastreaza in timpul asamblarii o variabila cunoscuta ca **CLI (Instruction Location Counter).** Aceasta variabila este initializata la 0 la inceputul primei treceri si este incrementata cu lungimea instructiunii pt fiecare instructiune procesata. [D1]

In prima trecere majoritatea asambloarelor utilizeaza cel putin 3 tabele: tabela de simboluri, tabela de pseudoinstructiuni si tabela de operatii. Daca este necesar, se construieste si o tabela de literali. Tabela de simboluri are o intrare pt fiecare simbol. Simbolurile sunt definite fie prin utilizarea lor ca etichete sau prin definitii explicite (de ex EQU). Fiecare intrare in tabela de simboluri contine simbolul respectiv (sau un pointer spre el), valoare numerica, si cateodata alte informatii. Aceste informatii suplimentare pot sa includa:

1. lungimea campului de date asociat cu simboluri.
2. biti de realocare.
3. daca simbolul este sau nu accesibil din afara procedurii.

Tabela pt codurile operatiilor contine cel putin cate o intrare pt fiecare nume simbolic (mnemonica) din limbajul de asamblare. Fiecare intrare contine un cod de operatie simbolic, 2 operanzi, valoare numerica a codului operatiei, lungimea instructiunii si un tip numeric care imparte codurile de operatii in grupuri in functie de numarul si tipul de operanzi. [D1]

Unele asambloare permit programatorilor sa scrie instructiuni utilizand adresarea imediata chiar daca in limbajul tinta nu exista astfel de instructiuni. Astfel de instructiuni ”pseudoimediate” sunt tratate in modul urmator: asamblorul aloca memorie pt operandul imediat la sfarsitul programului si genereaza o instructiune care sa il refere. De ex, calculatorul IBM 3090 nu are instructiuni imediate. Totusi programatorul poate sa scrie:

```
L  14,=F’5’
```

pentru a incarca in registrul 14 o constanta 5 reprezentata pe un cuvant intreg. In acest mod programatorul evita sa scrie explicit o pseudoinstructiune pt a aloca un cuvant initializat la 5, dandu-i o eticheta, si apoi utilizand aceasta eticheta in instructiunea L. Constantele pt care asamblorul aloca automat memorie se numesc **literali**. In afara de faptul ca programatorul scrie mai putin, utilizarea literalilor imbunatateste posibilitatea de a citit usor textul programului, deoarece constanta apare in instructiunea sursa. Prima trecere a unui asamblor trebuie sa construiasca o tabela a literalilor utilizati in program. Instructiunile imediate sunt obisnuite in prezent, dar mai demult erau neobisnuite. Este posibil ca utilizarea frecventa a literalilor sa fi facut clar proiectantilor ca adresarea imediata este o idee buna. Daca sunt necesari literali se va construi o tabela de literali in timpul asamblarii, in care se introduce o noua intrare pt fiecare literal intalnit. Dupa prima trecere tabela este sortata si intrarile duplicate sunt sterse.

Pe masura ce citeste programul , prima trecere trebuie sa parcurga fiecare linie si sa identifice codul operatiei (de ex, ADD), sa determine tipul (de fapt configuratia de operanzi), si sa calculeze lungimea instructiunii. Aceasta informatie este, de asemenea, pentru cea de-a 2-a trecere, deci este posibil sa o scriem (intr-un fisier) explicit pt a elimina parcurgerea liniei la urmatoarea trecere. Totusi , rescrierea fisierului de intrare determina aparitia mai multor operatii de I/O. Cu toate acestea alegerea mai multor operatii de I/O si mai multe parcurgeri ale liniilor, se face in functie de viteza relativa a CPU si a discului, de eficienta sistemului de fisiere , si de alti factori. [D1]

Cand se citeste pseudoinstructiunea END, s-a terminat prima trecere a asamblorului. Tabela de simboluri si cele de literali pot si sortate in acest moment daca este necesar. Tabela de literali sortata, poate fi verificata pt intrari duplicate, care vor fi eliminate. [D1]

A doua trecere:

Functia celei de-a doua treceri este de a genera programul obiect si, eventual, de a afisa listingul programului asamblat. In plus, a doua trecere trebuie sa furnizeze anumite informatii necesare editorului de legaturi, in vederea editarii legaturilor pentru procedurile asamblate la diferite momente de timp , intr-un singur fisier executabil.

Operatia realizata de cea de-a doua trecere este mai mult sau mai putin similara de cea realizata de prima trecere: citeste liniile, una cate una, si le prelucreaza, tot una cate una. Deoarece am scris (in fisierul temporar) tipul, codul operatiei si lungimea la inceputul fiecarei linii, toate aceastea sunt citite pt a reduce din parcurgerea si interpretarea liniilor. Operatia principala in generarea codului este facuta de procedurile eval_type1, eval_type2 etc. Fiecare dintre ele prelucreaza un anumit model, cum ar fi un cod de operatie cu operanzi din 2 registre, genereaza codul binar pt instructiune si il returneaza in *code*. Dupa aceea codul este scris intr-un fisier. De fapt, *write_code* depunde codul binar acumulat intr-o zona tampon si scrie fisierul pe disc in blocuri mari de date pt a reduce traficul de disc. [D1]

Prima trecere pentru un asamblor simplu:

```
Public static void pass_one ()
```

```
{
 // aceasta procedura este o schita a primei treceri a unui asamblor simplu
boolean more_input=true                    //indicator determinare a primei treceri
string line, symbol, literal, opcode       //campuri ale instructiunii
int location_counter, length, value, type; //diferite variabile
final int AND_STATEMENT=-2;                //semnaleaza sfarsitul intrarii

location_counter=0;                        //se asambleaza prima instruct la 0
initialize_table();                        //initializari generale

while(more_input) {                        //more_input pus pe de END
        line= read_next_line();            //citeste o linie de la intrare
        length=0;                          //nr de octeti in structiune
        typep=0;                           //tipul(formatul instructiunii)

        if(line_is_not_comment(line)) {
                symbol=check_for_symbol(line);
                if(symbol != NULL)         //daca este, memoreaza simbolul si
                                           //valoarea
                        enter_new_symbol(symbol, location_counter);
                literal=check_for_literal(line);
                if(literal1=NULL)          // daca da, introduce in tabela
                        enter_new_literal(literal);

                //acum se determina tipul codului operatiei; -1 reprezinta cod
                //ilegal de operatie
                opcode=extract_opcode(line); //determina mnemonica codului op
                type=search_opcode_table(opcode);
                if(type<0)
                        type=search_pseudotable(opcode);
                switch(type){              //determina lungimea instructiunii
                        case1: length=get_length_of_type1(line); break;
                        case2: length=get_length_of_type2(line); break;
                        //alte cazuri;
                }
        }

        write_temp_file(type,opcode, length, line); //informatii utile pt a doua
                                                    //trecere
        location_counter=location_counter+length;// actualizare loc_ctr
        if(type==SF_declaratie) {          //s-a terminat intrarea?
                more_input=false;          //daca da, exec act de intretinere
                rewind_temp_for_pass_two();
                sort_literal_table();
                remove_redundant_literals();
                }
```

```
        }
}
```

Instructiunea sursa originala si codul obiect generat pentru ea in hexazecimal pot fi apoi afisate sau depuse intr-o zona tampon pt a fi afisate ulterior.

Dupa ce CLI-ula fost actualizat, urmatoarea instructiune e citita si adusa din memorie.

Cea de-a doua trecere a unui asamblor simplu:

```
Public static void pass_one ()
{
//Aceasta precedura e o schita pt cea de-a doua trecere a unui asamblor simplu
boolean more_input=true                //indicator de terminarea a celei    de-
                                       //a 2-a treceri
string line, opcode;                   //campuri ale instructiunii
intlocation_counte, length,type;       //diferite variabile final
                                       //semnaleaza sfarsitul intrarii
final int MAX_CODE=16;                 //nr max de octeti pe instructiune
byte code[]=new byte[MAX_CODE];        //pastreaza codul generat pt instruct

location_counter=0;                    //se asambleaza prima instructiune de
                                       //la adresa 0

while(more_input) {                    //more_input ia val ‚false’ la END
        type=read_type();              //obtine campul tip al liniei urmatoare
        opcode=read_opcode();          //obtine campul codoperatie al liniei
                                       //urmatoare
        length=read_length();          //obt cp lungime al liniei urmatoare
        line=read_line();              //citeste linia crt din intrare

        if(type!=0) {                  //tipul 0 e pt linii de comentariu
                switch(type) {         //se genereaza codul de iesire
                        case1:eval_type1(opcode,length,line,code);break;
                        case2:eval_type2(opcode,length,line,code);break;
                        //in continuare, aici, alte cazuri
                }
        }
        write_output(code);            //se scrie codul binar
        write_listing(code, line);     //se scrie o linie in listingul programului
        location_counter=location_counter+length
        if(type==END_STATEMENT) {      //am ter cu intrarea?
                More_input=false       //daca da, executa task-urile de
                                       // intretinere
        finish_up();                   //si termina a doua intrare
        }
}
```

}

Pana acum s-a presupus ca programul sursa nu contine erori. Oricine a scris un program in orice limbaj stie cat de realista e aceasta presupunere. Cateva dintre cele mai intalinite erori sunt urmatoarele:

1. un simbol a fost utilizat dar nu a fost definit;
2. un simbol a fost definit de mai multe ori;
3. numele din campul codului operatiei nu este un cod de operatie corect;
4. codul unei operatii nu are numarul necesar de operanzi;
5. codul unei operatii are prea multi operanzi;
6. un numar octal contine una din cifrele 8 sau 9;
7. utilizarea de registre ilegale (de ex, un salt la un registru);
8. lipseste declaratia END.

Programatorii sunt foarte ingeniosi in a produce noi tipuri de erori. Erorile de tip simbol nedefinit sunt produse frecvent de erori de tastare, astfel un asamblor inteligent ar putea sa incerce care dintre simbolurile definite seamana cel mai mult cu cel nedefinit si sa-l utilizeze in locul acestuia. Putine erori pot sa fie facute de catre asamblor. Cel mai bun lucru pe care il poate face asamblorul cu o instructiune eronata este sa afiseze un mesaj de eroare si sa incerce sa continue asamblarea. [D1]

Tabela de simboluri

Tabela de simboluri (TS) este o structura de date destinata stocarii de informatii referitoare la identificatorii care apar intr-un text sursa , pe durata procesului de asamblare. [D3]

Pe durata primei treceri a procesului de asamblare, asamblorul acumuleaza informatii despre simboluri si valorile lor care trebuie memorate in tabela de simboluri pt utilizarea in cea de-a doua trecere. [D1]

Informatiile din TS sunt necesare pentru:

- efectuarea verificarilor legate de respectarea concordantei tipurilor si a regulilor de vizibilitate (verificari semantice);
- generarea de cod [D3]

Informatii continute de TS

In cazul limbajelor sursa care definesc programe structurate pe blocuri (functii si/sau proceduri) care pot sa apara pe mai multe nivele de imbricare, in TS ar trebui sa apara, in principiu, urmatoarele informatii:

- ***NUME*** - referinta la sirul de caractere care formeaza identificatorul. Acest sir constituie de fapt cheia de cautare in TS
- ***CLASA*** - categoria identificatorului, care poate fi: nume de program, nume de constanta (declarat in sectiunea **const**), nume de variabila simpla , nume de functie, nume de procedura , nume de parametru formal transmis prin adresa (declarat cu **var** in lista de parametri), nume de

parametru formal transmis prin valoare, nume de tablou, nume de structura (**record**), nume de camp al unei structuri.

- ***TIP*** - tipul (simplu) declarat pentru variabilele simple, parametrii formali, elementele unui tablou, campurile unei structuri sau rezultatul returnat de o functie. Pentru limbajul dat in Anexa B, acest tip poate fi: intreg, real sau caracter.
- ***VAL*** - se utilizeaza pentru numele de constante (declarate in sectiunea **const**) si contine indicele spre *TabCONST* unde se afla valoarea atribuita constantelor respective.
- ***ADREL*** -adresa relativa fata de inceputul unitatii de program in care este declarat identificatorul. Se refera la variabilele declarate in sectiunea **var** si la parametrii formali.
- ***DEPLREC*** - deplasamentul unui camp de structura fata de inceputul structurii respective. Pentru primul camp dintr-o structura ***DEPLREC*** = 0, iar pentru restul campurilor calculul se face ca si pentru ***ADREL***.
- ***NIVEL*** - reprezinta nivelul de imbricare la care se gaseste un identificator. Pentru domeniul de vizibilitate al programului principal nivelul este 1, pentru blocurile declarate in programul principal nivelul este 2 s.a.m.d. Pentru completarea campului ***NIVEL*** din TS, se poate utiliza o variabila globala care se initializeaza cu 1 si care se actualizeaza astfel:
  - ➔ dupa ce s-a trecut de numele unui subprogram, variabila se incrementeaza cu 1;
  - ➔ dupa ce s-a trecut de "end"-ul unui subprogram, variabila se decrementeaza cu 1.
- ***NR_PAR*** - se completeaza numai pentru nume de functii si proceduri si reprezinta numarul de parametri formali.
- ***DIM_VAR*** - se completeaza pentru numele de functii, proceduri si program principal si reprezinta numarul total de locatii din stiva de lucru necesare pentru memorarea variabilelor locale (NU si a parametrilor formali).
- ***ADR_START*** - se completeaza pentru numele de functii, proceduri si program principal si reprezinta adresa (indicele) in tabela de cod la care incepe corpul de instructiuni al unitatii de program respective.
- ***LISTA_PAR*** - se completeaza pentru numele de functii si proceduri si reprezinta o lista (sau pointer spre inceputul unei liste) avand cate un nod corespunzator fiecarui parametru formal.
- ***IND_MIN, IND_MAX*** - se completeaza pentru numele de tablouri si reprezinta valorile minima , respectiv maxima declarate pentru indice. Pentru simplificare presupunem ca indicii de tablouri pot fi numai de tip intreg.
- ***LISTA_REC*** - se completeaza in cazul numelor de campuri de structura si reprezinta o lista de referinte in TS, spre intrarile corespunzatoare numelor de structuri de care apartin campurile respective. De exemplu, presupunem ca in textul sursa apare o declaratie de forma:

a, *b*, *c* : **record** *x* : tip1; *y* : tip2 **end**;

Pentru campurile *x* si *y* ***LISTA_REC*** va contine referintele spre intrarile din TS corespunzatoare simbolurilor *a*, *b* si *c*. Aceasta lista este necesara pentru a verifica daca , intr-o referire de forma *r.c*, *c* este camp al lui *r.*

- ***INCDOM*** - se completeaza in cazul numelor de subprograme si reprezinta valoarea curenta a indicelui in tabela de cod la momentul intalnirii declaratiei subprogramului respectiv. Aceasta informatie va fi necesara in faza generarii de cod. [D3]

TS trebuie sa contina toti identificatorii din textul sursa VIZIBILI la un moment dat. De aceea, cand se detecteaza sfarsitul unui subprogram, intrarile din TS aferente parametrilor formali si variabilelor locale din subprogramul respectiv vor fi eliberate.
Se observa ca unele informatii din TS au sens doar pentru anumite categorii de simboluri. De aceea, o parte dintre campurile TS pot fi grupate in structuri de tip *union* (*record* cu variante). De exemplu: *ADREL* (care nu are sens pentru campuri de structuri) se poate grupa cu DEPLREC (care are sens doar pentru campuri cu structuri). [D3]

Tehnici de organizare a TS:

Pot fi utilizate mai multe moduri de organizare a tabelei de simboluri. Vom descrie in continuare, pe scurt, unele dintre ele. Toate incearca sa simuleze o **memorie asociativa (associative memory)**, care este, conceptual, o multime de perechi (simbol, valoare). Dandu-se un simbol, memoria asociativa trebuie sa furnizeze valoarea. [D1]

1.Cea mai simpla tehnica de implementare este, implementarea tabelei de simboluri ca un tablou de perechi; primul element al acestuia este (sau refera) simbolul, iar cel de-al doilea este (sau refera) valoarea. Dandu-se un simbol de cautat, rutina asociata tabelei de simboluri cauta liniar in tabela pana gaseste simbolul respectiv. Aceasta metoda e usor de programat dar este lenta deoarece in medie la fiecare cautare va fi parcursa jumatate din tabela. [D1]

2.Un alt mod de organizare a tabelei de simboluri consta din ordonarea tabelei dupa simboluri si utilizarea unui algoritm de **cautare binara (binary search)** pentru a cauta un simbol. Acest algoritm opereaza prin compararea intrarii din mijlocul tabelei cu simbolul cautat. Daca simbolul e alfabetic inainte de intrarea din mijloc simbolul trebuie localizat in prima jumatate a tabelei. Daca simbolul este alfabetic dupa intrarea din mijloc el trebuie sa fie in cea de-a doua jumatate a tabelei. Daca simbolul e egal cu intrarea din mijlocul tabelei, atunci cautarea s-a terminat. Presupunand ca intrarea din mijloc nu este egala cu simbolul cautat, vom stii cel putin in care jumatate a tabelei sa-l cautam. Acum, cautarea binara poate fi aplicata jumatatii identificate, care va furniza fie o potrivire, fie sfertul corect al tabelei. Aplicand recursiv algoritmul, o tabela cu dimensiunea *n* intrari poate fi parcursa in cca $log_2 n$ incercari. Evident, acest mod e mult mai rapid decat cautarea liniara, dar necesita mentinerea ordonata a tabelei. [D1]

3.In cazul organizarii TS ca stiva , atat ordinea logica cat si cea fizica a simbolurilor in tabela coincid cu ordinea de aparitie in textul sursa. Eliberarea spatiului din tabela la intalnirea sfarsitului unui bloc se face simplu: se parcurge

stiva spre baza pana la intalnirea primului simbol pentru care campul *NIVEL* are o valoare mai mica decat simbolul din varf. De asemenea este simplificata si operatia de completare a unor campuri din tabela pentru care informatiile necesare se obtin mai departe de locul in care au aparut simbolurile in cauza.

Dezavantajul acestui mod de organizare l-ar putea constitui timpul de cautare, in situatiile in care in TS se afla foarte multe simboluri, deoarece cautarea trebuie facuta secvential. [D3]

Functii de acces la TS: asupra lui TS se pot efectua 2 operatii de baza:

1. **Cautarea unui identificator**. Cheia de cautare este sirul de caractere care compun identificatorul. Cautarea se va face intotdeauna in sens descrescator al nivelului de imbricare (campul ***NIVEL***) si se va opri la prima aparitie a simbolului cautat sau la epuizarea tabelei;
2. **Inserarea unui identificator**. **Aceasta operatie este efectuata cand in textul sursa se intalnesc declaratii de simboluri (numele programului principal, constante, variabile, functii, proceduri si parametri formali)**. Inainte de inserare se va efectua o cautare a simbolului de inserat, pentru a se detecta eventualele declaratii multiple. Daca un simbol mai este declarat o data in acelasi bloc, se va semnala o eroare de tip "identificator multiplu declarat". [D3]

Exceptie de la aceasta regula vor face numele de campuri de structuri, in sensul ca:

- se permit situatii in care un nume de camp coincide cu un nume de variabila simpla , de tablou sau de functie/procedura declarate in acelasi bloc ca si structura in cauza.
- de asemenea se permite ca doua campuri care apar in structuri diferite sa aiba acelasi nume; in acest caz se vor crea intrari distincte in TS pentru cele 2 campuri.

NU se permite, insa , ca in aceeasi structura sa apara doua campuri cu acelasi nume. [D3]

Rezumat:

Lucrarea urmareste detalierea procesului de asamblare (asambloare in 2 treceri) si a tabelei de simboluri.

In timpul procesului de asamblare poate aparea problema referintei inainte ceea ce duce la erori de asamblare. Acest lucru este evitat prin asamblarea in 2 treceri: asamblorul citeste programul sursa de 2 ori; in prima trecere definitiile sibolurilor sunt corectate si memorate in tabela de simboluri astfel incat, catnt incepe a doua trecere valorile tuturor simbolurilor sunt cunoscute; nu mai exista referinte inainte si fiecare instructiune este citita, generata si scrisa.

Principala functie a primei treceri este sa construiaca tabela de simboluri continand valorile tuturor simbolurilor.

A doua trecere are rolul de a genera programul obiect si, eventual, de a afisa listingul programului asamblat. In plus, a doua trecere trebuie sa furnizeze anumite informatii necesare editorului de legaturi, in vederea editarii legaturilor

pentru procedurile asamblate la diferite momente de timp , intr-un singur fisier executabil.

Tabela de simboluri (TS) este o structura de date destinata stocarii de informatii referitoare la identificatorii care apar intr-un text sursa , pe durata procesului de asamblare. Pe durata primei treceri a procesului de asamblare, asamblorul acumuleaza informatii despre simboluri si valorile lor care trebuie memorate in tabela de simboluri pt utilizarea in cea de-a doua trecere. TS retine informatii cruciale in vederea excecutarii corecte a procesului de asamblare.

TS poate fi organizata in mai multe moduri printer care: implementarea ca un tablou de perechi, ordonarea tabelei dupa simboluri si utilizarea unui algoritm de **cautare binara** sau organizarea TS ca stiva.

De asemeanea TS poate fi accesata pentru cautarea sau inserarea unui identificator.

Bibliografie:

1. Organizarea Structurata a Calculatoarelor – Andrew S. Tanenbaum (editia a IV-a) [D1]

2. Computer Science Illuminated – Nell Dale, John Lewis, ed Jones and Bartlett Publishers [D2]

3. http://labs.cs.utt.ro/labs/lft/html/LFT08.htm [D3]

Editarea legaturilor si incarcarea programelor

Majoritatea programelor sunt formate din mai mult de o procedura. In general, asambloarele translateaza o procedura la un moment dat si scriu iesirea pe disc. Pentru ca programul sa poata fi executat, toate procedurile trebuie gasite si legate in mod corespunzator. Daca nu este disponibila memorie virtuala, programul editat trebuie incarcat explicit in memoria principala. Programele care realizeaza aceste functii sunt denumite editor de legaturi (linker) sau incarcator si editor de legaturi (linking loader) de programe.

Traducerea completa a programului sursa necesita doua etape:

1. Compilarea sau asamblarea procedurilor sursa
2. Editarea legaturilor pentru modulele obiect

Prima etapa este realizata de catre compilator sau asamblor, cea de-a doua de catre editorul de legaturi.

Translatarea de la procedura sursa la modulul obiect reprezinta o modificare de nivel, deoarece limbajul sursa si limbajul obiect au notatii si instructiuni diferite. Procesul de editare de legaturi nu reprezinta totusi o modificare a nivelului deoarece atat intrarile cat si iesirile editorului de legaturi sunt programe pentru aceeasi masina virtuala. Functia editorului de legaturi este de a colecta procedurile translatate separat si de a lega impreuna pentru a fi executate unitar ca program binar executabil. Pentru sistemele MS-DOS, Windows 95/98 si NT modulele obiect au extensia *.obj,* iar programele binare executabile au extensia *.exe.* Pe UNIX modulele obiect au extensia *.o*, iar programele binare executabile nu au extensie.

Compilatoarele si asambloarele translateaza fiecare procedura sursa ca o entiate separata. Altfel, modificare unei declaratii intr-o procedura sursa ar necesita ca toate procedurile sursa sa fie retranslatate. Utilizand tehnica modulelor obiect separate se va retransalta doar procedura modificata, dar va fi necesar sa se reediteze legaturile pentru toate modulele obiect. Editarea legaturilor fiind mult mai rapida decat translatarea, se va economisi timp considerabil, mai ales in cazul programelor cu sute sau mii de module.

Activitatile realizate de editorul de legaturi

La inceputul primei treceri in cadrul procesului de asamblare, numaratorul de locatii de instructiuni este 0, echivalent cu presupunerea ca modulul obiect va fi plasat la adresa (virtuala) 0 la momentul executiei. Pentru a executa programul, editorul de legaturi aduce modulele obiect in memoria principala pentru a forma imaginea programului binar executabil. Se doreste a se construi o imagine exacta a spatiului de adrese virtual al programului executabil in cadrul editorului de legaturi si de a pozitiona toate modulele obiect la locatiile lor corecte. Daca nu exista suficienta memorie(virtuala) pentru a forma imaginea, se poate utiliza un fisier pe disc. De obicei, o mica portiune din memorie incepand de la adresa zero este utilizata pentru vectorii de intrerupere, comunicatia cu sistemul de operare, capturarea referintelor de memorie neinitializate sau pentru alte scopuri, asa ca programul incepe adesea de la o adresa mai mare ca 0.

Problemele care apar cand programul este incarcat in imaginea fisierului binar executabil poarta numele de **probleme de relocare** si se datoreaza faptului ca fiecare modul obiect formeaza separat un spatiu de adresa. Majoritatea versiunilor de Windows si UNIX ofera numai un spatiu liniar de adrese, astfel ca toate modulele trebuie sa fie concatenate intr-un singur spatiu de adrese. O alta problema este cea a **referintelor**

externe datorata faptului ca asamblorul nu are o modalitate de a sti la ce adresa sa insereze instructiunea pentru un alt modul, adresa acestuia nefiind cunoscuta pana in momentul editarii legaturilor.

Editorul de legaturi poate rezolva aceste probleme unind spatiile separate de adresa ale modulelor obiect intr-un singur spatiu liniar de adrese, in urmatorii pasi:

1. Construieste o tabela cu toate modulele obiect si dimensiunile acestora
2. Pe baza tabelei asigneaza o adresa de start pentru fiecare modul obiect
3. Determina toate instructiunile cu referire la memorie si aduna fiecareia o **constanta de realocare (relocation constant)**, egala cu adresa de start a modulului sau.
4. Determina toate instructiunile ce fac referire la alte proceduri si insereaza adresa acelor proceduri in pozitiile respective.

Structura unui modul obiect

Modulele obiect contin adesea sase parti. Prima contine numele modulului, anumite informatii necesare pentru editorul de legaturi, ca lungimea diferitelor parti ale modulului si uneori data asamblarii.

A doua parte a modulului obiect este o lista de simboluri definite in modul, pe care alte module le pot referi, impreuna cu valorile lor. Programatorul in limbaj de asamblare indica care simboluri trebuie sa fie declarate ca puncte de intrare (entry points), prin utilizarea unei pseudoinstructiuni, ca PUBLIC.

A treia parte consta dintr-o lista de simboluri utilizate in modul dar care sunt definite in alte module, impreuna cu o lista a instructiunilor masina care folosesc aceste simboluri. Editorul de legaturi utilizeaza aceasta lista pentru a putea sa insereze adresele corecte si instructiunile ce utilizeaza simboluri externe. O procedura poate apela alte proceduri translatate independent prin declararea numelor procedurilor apelate ca fiind externe. Programatorul in limbaj de asamblare indica ce simboluri sunt declarate ca simboluri externe (external simbols) prin utilizarea unei pseudoinstructiuni ca EXTRN.

A patra parte este codul asamblat si constantele. Aceasta parte este singura care va fi incarcata in memorie pentru a fi executata. Celelalte cinci parti vor fi utilizate de editorul de legaturi si apoi eliminate inainte de inceperea executiei.

Cea de-a cincea parte este dictionarul de relocare. Instructiunile ce contin adrese de memorie trebuie sa aiba adunata o constanta de relocare. Deoarece editorul de legaturi nu are o modalitate de a spune, prin inspectie, care dintre cuvintele de date din partea a patra contin instructiuni masina si care contin constante, informatia despre adresele care vor fi relocate este furnizata de aceasta tabela. Informatia poate lua forma unei tabele de biti, cu 1 bit pentru adresa potential relocabila, sau o lista explicita de adrese ce vor fi relocate.

A sasea parte este o indicatie de sfarsit de modul, uneori o suma de verificare pentru a descoperi erori de citire a modulului si adresa la care incepe executia.

Cele mai multe editoare de legaturi necesita doua treceri. In prima editorul de legaturi citeste toate modulele obiect si construieste o tabela a lungimilor si numelor modulelor si o tabela de simboluri constand din toate punctele de intrare si referintele externe. In a doua trecere modulele obiect sunt citite, relocate si sunt editate legaturile in cate un modul la un moment dat.

Momentul legarii si relocarii dinamice

Intr-un sistem cu multiprogramare, un program poate fi incarcat in memoria principala, executat pentru un timp, scris pe disc si apoi incarcat din nou in memoria principala pentru a fi executat din nou. Intr-un sistem mare, cu multe programe, este dificil sa se asigure ca un program este incarcat din nou in aceleasi locatii, de fiecare data. Informatia de relocare se poate pierde in urma scrierii pe disc. Chiar daca informatia de relocare ar fi inca disponibila, costul de a reloca toate adresele de fiecare data cand programul este readus in memorie ar putea fi mare.

Problema mutarii programelor care au fost deja legate si relocate este strans legata de momentul la care este terminata legarea finala a numerelor simbolice pentru adresele de memorie. Momentul la care este stabilita adresa curenta de memorie principala este denumit momentul atribuirii de adresa (binding time). Exista cel putin sase posibilitati pentru momentul atribuirii de adresa:

1. Cand programul este scris
2. Cand programul este translatat
3. Cand se face editarea legaturilor dar inainte de a fi incarcat
4. Cand programul este incarcat
5. Cand un registru de baza, utilizat pentru adresare, este incarcat
6. Cand se executa instructiunea ce contine adresa

Daca o instructiune ce contine o adresa de memorie este mutata dupa atribuirea adresei, ea va fi incorecta. Daca translatorul furnizeaza ca iesire un cod binar executabil, atribuirea adresei s-a facut la momentul translatiei si programul trebuie sa fie rulat la adresa la care translatorul se astepta sa fie executat. Metoda prezentata leaga nume simbolice de adrese absolute in timpul editarii de legaturi, de aceea mutarea programelor dupa editarea de legaturi esueaza.

Prima problema apare cand se face asocierea intre numele simbolice si adresele virtuale. A doua la momentul cand adresele virtuale sunt asociate cu cele fizice. Numai dupa ce ambele operatii au avut loc atribuirea adresei este completa. Cand editorul de legaturi uneste spatiile separate de adrese ale moduleleor obiect intr-un singur spatiu liniar de adrese, se creeaza de fapt un spatiu virtual de adrese. Acest lucru este adevarat indiferent daa memoria virtuala este utilizata sau nu.

Daca spatiul de adrese este paginat, adresele virtuale ce corespund numelor simbolice au fost deja determinate chiar daca adresele fizice din memoria principala vor depinde de continutul tabelei de pagini la momentul la care ele sunt utilizate. Un program binar executabil este de fapt o asociere a numerelor simblice cu adresele virtuale.

Mutarea programelor in memoria principala, chiar dupa ce ele au fost asociate unui spatiu virtual de adrese, va fi inlesnita de orice mecanism care permite modificarea usoara a punerii in corespondenta a adreselor virtuale. Un astfel de mecanism este paginarea. Dupa ce un program a fost mutat in memoria principala, numai tabela sa de pagini trebuie modificata, nu programul insusi.

Un al doilea mecanism este utilizarea in timpul executiei a unui registru de relocare. Pe masinile ce utilizeaza aceasta tehnica de relocare registrul refera intotdeauna adresa de memorie fizica de start a programului curent. La toate adresele de memorie se va aduna prin hardware continutul registrului de relocare inainte de a fi transmise la memorie. Intregul proces de relocare este transparent pentru programele utilizator. Ele nu stiu nici

macar daca a fost utilizat. Cand un program este mutat, sistemul de operare trebuie sa actualizeze registrul de relocare. Acest mecanism este mai putin general decat paginarea deoarece intregul program trebuie sa fie mutat ca un intreg.

Un al treilea mecanism este posibil pe masinile ce pot referi memoria relativ la contorul program. Ori de cate ori un program este mutat in memoria principala trebuie sa fie actualizat numai numaratorul de instructiuni. Un program ale carui referinte la memorie sunt fie relative la contorul program, fie absolute se spune ca este independent de pozitie. O procedura independenta de pozitie poate fi plasata oriunde in spatiul virtual de adrese, fara a fi nevoie de relocare.

Bibliografie

Organizarea Structurata a Calculatoarelor – Andrew S. Tanenbaum (editia a IV-a)

Exemple de Asambloare si Caractreristicile lor Distinctive

Un asamblor este un porgram care citeste dintr-un fisier text (sursa) instructiuni de asamblare si le converteste in cod masina. Compilatoarele sunt programe ce convertesc similar programele scrise in limbaje de programare de nivel inalt. Un asamblor este insa mult mai simplu decat un compilator. In fiecare limbaj de asamblare unei instructiuni ii corespunde o singura instructiune in cod masina. Spre deosebire de limbajele de asamblare, limbajele de nivel inalt folosesc instructiuni complexe carora le pot corespunde mai multe instructiuni in cod masina.

O mare diferenta intre un limbaj de asamblare si un limbaj de nivel inalt este aceea ca fiecare procesor are propriul set de instructiuni in cod masina si ii corespunde un anumit limbaj de asamblare, spre deosebire de limbajele de programare de nivel inalt care pot functiona pe mai multe familii de procesoare.

Dat fiind faptul ca fiecare familie de procesoare are propriul set de instructiuni s-au dezvoltat mai multe asambloare. In acest capitol ne vom ocupa cu descrierea catorva tipuri de asambloare. Am ales asambloarele: Netwide Assembler (NASM), Microsoft's Assembler (MASM), Borland's Assembler (TASM), SunOS 5.x SPARC Assembler, ASEM-51. Diferentele dintre aceste asambloare sunt de sintaxa si de tipul de procesor pe care ruleaza, unele dintre ele putand rula pe mai multe familii de procesoare.

1. Asamblorul ASEM-51

ASEM-51 este un asamblor in doua treceri pentru familia de microcontrelere Intel MCS-51 si ruleaza pe PC-uri sub MS-DOSM, Windows si Linux.

In DOS in modul real, asamblorul ASEM-51 (executabilul ASEM.EXE) are nevoie pentru a rula doar de 256KB de memorie ram, si sistem de operare MS-DOS 3.0 (sau mai nou).

In modul protejat (ASEMX.EXE) asamblorul necesita un procesor 286 (sau mai bun) si cel putin 512 KB de memorie libera pentru o functionare buna.

Pentru noul mod consola Win32 (ASEMW.EXE), asamblorul functioneaza pe un procesor 386 (sau mai bun) si sistem de operare Windows 9x, NT, 2000 sau XP.

In Linux asamblorul functioneaza pe calculatore cu procesoare incepand cu Intel 80386 (sau compatibil).

Fiind un asamblor in doua treceri, operatia de asamblare se va realiza in doi pasi:

1. La primul pas se genereaza o tabela de simboluri, ce va contine toate numele simbolice din program, exceptand numele simbolice externe (definite in alte module), instructiuni si directive de asamblare. In aceasta etapa asamblorul contorizeaza instructiunile si datele si asociaza numelor simbolice o pozitie relativa (deplasament) fata de inceputul programului, ca si cum programul ar incepe de la adresa 0. In realitate, programul nu se incarca in memoria RAM de la adresa zero, care este zona folosita de sistyemul de operare, ci adresa de la

care se incarca este furnizata de sistemul de operare, in functie de spatiul de memorie disponibil; din acest motiv programul furnizat de asamblor este relocabil.
2. La pasul al doilea se obtine programul obiect, translatand instructiune cu instructiune programul si inlocuind in instructiunile respective numele simbolice cu valorile numerice asociate in tabela de simboluri. Programul executabil se obtine in urma etapei editarii de legaturi, care permite legarea mai multor module relocabile intr-un singur fisier; acest fisier va contine fisierul executabil, rezolvandu-se toate referintele incrucisate care au fost folosite in alte module.

2. MASM

Microsof Assembler (MASM) este un asamblor ce lucreaza sub Dos pe procesaore x86. Ofera compatibilitate atat pe 16 cat si pe 32 de biti. Ultima versiune de MASM care s-a vandut separat a fost MASM 6.11. Dupa aceasta MASM nu s-a mai vandut ca un produs de sine statator, ci a fost inclus in diferite pachete vandute de Microsoft; astfel versiunea 6.15 a aparut cu Visual C++ 6.0 Processor Pack, versiunea 7.0 a fost inclusa in pachetul Visual C++ .NET 2002, versiunea 8.0 a aparut cu Visual C++ 2005 ce contine si o versiune pentru procesoare pe 64 de biti.

Microsoft Assembler suporta o larga varietate de facilitati macro si programare structurata, incluzand constructii de nivel inalt pentru bucle, apeluri de proceduri si alternari. Desi la inceput a fost proiectat sa functioneze pentru MS-DOS, versiunile ulterioare au adus capabilitatea de a programa aplicatii pentru Windows. Astfel de la versiunea 6.1+ MASM este o aplicatie de tip consola Win32.

Cateva caracteristici ale acestui asamblor:

- Suport complet pentru Win32. MASM 6.1x este o aplicatie de tip consola Win32 ce suporta nume lungi pentru fisierele sursa, obiect si de simboluri;
- Suporta ambele tipuri de formate pentru obiecte: Intel OMF (cele vechi), si COFF (Win32);
- Prototipuri pentru functii (headers);
- Posibilitatea declararii de date de tip utilizator (TYPEDEF);
- Directive pentru programarea structurata. Aceasta ajuta la realizarea de programe cu cat mai putine salturi (JMP). Etichetele si instructiunile de salt sunt inca prezente insa;

MASM este un asamblor foarte rapid atunci cand functioneaza sub Windos95 sau NT. Deoarece sistemele Win32 au destula memorie RAM, viteza de asamblare nu are de suferit din cauza compilarii unei cantitati mari de fisiere „include” si/sau a unui numar mare de macro-uri complexe.

3. <u>Netwide Assembler</u>

Asamblorul Netwide Assembler (NASM) este proiectat pentru familiile de procesoare x86 si x86-64 si ofera o buna portabilitate. Suporta o intreaga gama de formate pentru fisierele obiect, inclusiv Linux si *BSD a.out, ELF, COFF, Mach-O, Microsoft 16-bit OBJ, Win32 si Win64. Sintaxa lui este simpla si usor de inteles, asemanandu-se ce cea de la Intel, insa mai putin complexa. Suporta seturile de instructiuni: Pentium, P6, MMX, 3DNow!, SSE, SSE2, SSE3 si x64.

NASM a luat nastere datorita absentei pe piata, a unui asamblor bun si usor de folosit care sa fie gratuit. Mai exista si alte asambloare gratuite, dar acestea fie nu suporta decat procesoare pe 16 biti, fie pot rula doar sub DOS. Nici asambloarele scumpe nu satisfaceau cerintele oricarui programator; spre exemplu: MSAM nu ruleaza decat sub DOS, a86 nu are suport pentru 32 de biti, as86 este specific pentru Minix si Linux doar si documentatia pentru el este prea putina.

Astfel a aparut NASM, un asamblor gratuit care sa raspunda tuturor cerintelor unui programator in limbaj de asamblare.

Ultima versiune aparuta pentru acest asamblor este: NASM 2.0 ce a aparut in Noiembrie 2007.

<u>Comparatie intre NASM si MASM</u>

- O prima deosebire intre aceste doua asambloare este ca NASM face diferenta dintre litera mica si litera mare („case-sensitive”).

- In MASM la declararea unei variabile si retine si tipul acesteia; in NASM insa nu se retine decat adresa la care se gaseste variabila. Ex: in MASM se declara variabila „var” astfel „var dw 0”, astfel la aparitia unei instructiuni „mov var, 2” se cunoaste dimensiunea variabilei, pe cand in NASM trebuie explicitata aceasta in interiorul instructiunii: „mov word [var], 2”. Astfel NASM nu suporta instructiunile: LODS, MOVS, STOS, SCAS, CMPS, INS sau OUTS.

- NASM nu are nici o directiva pentru a suporta diferite modele pentru 16 biti. Programatorul trebuie sa tina cont de forma instructiunilor (call far, call near).

- NASM foloseste nume diferite cand apeleaza registre pentru virgula-mobila: la MASM se folosesc numele ST(0), ST(1) etc. la NASM acestea se numesc st0, st1, etc.

- Modul de declarare a variabilelor neinitializate este, de asemenea, diferit: daca la MASM un programator poate folosi instructiunea: „stack db 32 dup (?)”, la NASM aceasta se scrie „stack resb 32” ce va fi interpretata ca ’rezerba 32 de octeti’. NASM trateaza semnul „?” ca un caracter valid in numele de simboluri.

4. <u>TASM</u>

Turbo Assembler (TASM) este limbajul de asamblare de la firma Borland. Incepand cu vesiunea 4.0 acesta ofera suport pentru programarea pe 32 de biti (are totusi nevoie de un patch pentru a rula in fereastra de DOS in Win95).

Daca inainte TASM era mai bine compatibil cu MASM decat MASM insusi, incepand cu vers. 5.0, TASM nu a mai reusit sa tina pasul cu asamblorul celor de la Microsoft si un numar de facilitati noi prezente in MASM 6.10 nu suportate de TASM. Aceste facilitati aveau in mare parte rolul de a usura programarea in Win32.

5. SunOS 5.x Sparc Assembler

Asamblorul SunOS Sparc ruleaza sub sistemul de operare SunOS 5.x sau sub mediul de operare SolarisTM 2.x si este destinat utilizarii pe calculatoare cu arhitectura Sparc.

In urma asamblarii fisierelor sursa cu SunOS Sparc se creaza fisiere de obiecte de tip ELF (Executable and Linking Format). Aceste fisiere contin cod si date destinate link-arii cu alte fisiere de obiecte, in scopul crearii unui fisier executabil.

Acest limbaj de asamblare, fiind dedicat procesoarelor Sparc, foloseste ca set de instructiuni, instructiunile definite in Manualul despre Arhitectura Sparc.

Bibliografie:

http://plit.de/asem-51/docs.htm
http://en.wikipedia.org/wiki/Assembly_language#Assembler
http://www.asmcommunity.net/projects/nasmx/
http://www.masm32.com/

- **Proiectarea unui asamblor simplu**

Introducere şi obiective

Notă preliminară: noţiunile de bază referitoare la mecanismele de asamblare şi la elementele componente ale unui cod scris în limbaj de asamblare se consideră trecute în revistă şi explicate în cealaltă parte a lucrării de faţă.
Acest capitol enunţă pe scurt la început obiectivele ce se doresc a fi atinse prin implementarea unui asamblor. în cuprinsul lucrării sunt propuse şi analizate soluţii pentru atingerea unora dintre ele.
Un asamblor este o unealtă software *independentă din punct de vedere conceptual* de sistemul hardware pentru care creează cod maşină. Aşa cum s-a descris în capitolele anterioare, spre exemplu, o instrucţiune generică de asamblare este de formă: <etichetă><operaţie><operanzi><comentariu>. [Ref. Bibl. X]. Prin urmare, putem identifica un set de proprietăţi specifice fiecărei platforme care pot fi agregate în configuraţii complexe ale asamblorului. Bineînţeles există şi excepţii de la această regulă, care însă pot fi la rândul lor cumulate într-un set de extensii configurabile de la caz la caz. Această analiză va fi realizată pentru fiecare element în §1.2
O altă caracteristică foarte utilă într-un asamblor este raportarea erorilor sintactice la nivelul liniei de cod eronate.
în cele din urmă se prezintă funcţionalitatea de bază a oricărui asamblor, şi anume traducerea efectivă a instrucţiunilor în cod maşină.

Identificarea construcţiilor din codul sursă

Instrucţiuni

Reamintim că o instrucţiune assembler are în general formă:

[<etichetă>]<operaţie>[<operanzi>][<comentariu>]

Elementele cuprinse între [] sunt opţionale: etichetă, operandul/operanzii şi comentariul. De exemplu instrucţiunea *AAA* (Ascii Adjust for Addition) pentru procesorul Intel x86 nu acceptă nici un operand. Mai jos sunt listate câteva exemple de instrucţiuni care cuprind toate elementele amintitie, pentru procesoarele Intel Pentium 2, Motorola 680x0 şi respectiv SPARC:

```
FORMULA: MOV    EAX,I            ;registrul EAX=I
FORMULA  MOV.L  I,D0             ;registrul D0=I
```

FORMULA: SETHI %HI(I), %R1   !R1=msb din I

Putem extrage din aceste exemple o structură relativ generală pentru recunoaşterea unei instrucţiuni:

[<etichetă><separator_etichetă>]

<instrucţiune>

[<operand_1>]

[<separator_operanzi><operand_2>]

[...]

[<separator_comentariu><comentariu>]

între elementele notate prin **<...>** se pot regăsi oricâte spatii, iar tot şirul de caractere întâlnit de la caracterul de comentariu până la finalul liniei se consideră <comentariu>.

Pentru liniile de asembler prezentate mai sus, trebuie extrase următoarele elemente:

	Intel Pentium 2	***Motorola 680x0***	***SPARC***
etichetă	FORMULA	FORMULA	FORMULA
instrucţiune	MOV	MOV.L	SETHI
nr operanzi	2	2	2
operand 1	EAX	I	%HI(I)
operand 2	I	D0	%R1
comentariu	registrul EAX=I	registrul D0=I	R1=msb din I

De asemenea putem extrage un set de verificări ce sunt efectuate pentru identificarea erorilor la analiza unei astfel de linii:

- unele limbaje impun începerea etichetei de pe prima coloană [Ref bibl X]
- eticheta identificată în linia respectivă trebuie să respecte anumite reguli lexicale (ex: să aibă mai puţin de 256 de caractere [Ref bibl X])
- verificarea prezenţei caracterului de separare a etichetei
- operanzii identificaţi trebuie să respecte anumite reguli lexicale

- **Pseudoinstrucţiuni (directive)**

Să analizăm analog secţiunii anterioare câteva pesudoinstrucţiuni ale aceloraşi limbaje de arhitecturi mentionate mai sus:

I DW 3   ;rezervare 4 octeţi iniţializaţi cu 3

I DC.L 3   ;rezervare 4 octeţi iniţializaţi cu 3

I: .WORD 3 !rezervare 4 octeţi iniţializaţi cu 3

Observăm că fiecare linie începe cu un simbol - în cazul de faţă "l" - urmat eventual de un separator (":" pentru SPARC), de pseudoinstructiunea propriu-zisă, de un "parametru" al pseudoinstrucţiunii şi de un comentariu. Considerăm acum un alt exemplu de pseudoinstrucţiune pentru familia Intel x86:

ALIGN 4

care îndrumă asamblorul să alinieze următoarea instrucţiune sau dată în memorie la o adresă multiplu de 4. De data aceasta se observă o altă structură sintactică a pseudoinstrucţiunii, diferită de cea extrasă în exemplul anterior. Pe de altă parte, fiecare instrucţiune în parte are o semnificaţie diferită pentru asamblor, ceea ce ne conduce la concluzia că nu putem realiza un model generalizat al pseudoinstrucţiunilor, însă ne putem folosi de definiţiile specifice fiecărui limbaj pentru a potrivi liniile de cod cu *anumite* structuri specifice pseudoinstrucţiunilior. Astfel, asamblorul va avea o serie de funcţionalităţi tip pseudoinstrucţiune pe care le suportă, pentru fiecare dintre ele fiind necesar să se specifice per limbaj de asamblare(arhitectură) dacă sunt disponibile sau nu, care anume este formatul sintactic şi erorile ce pot fi detectate într-o asemenea construcţie. Un exemplu de astfel de configuraţie a asamblorului este schiţat mai jos pentru două tipuri de asambloare:

Funcţionalitate	**Intel**			**Z80**		
	Disp.	**Format**	**Erori posibile**	**Disp.**	**Format**	**Erori posibile**
Alocare cuvinte	da	<etichetă> DW <val. iniţială> [;<comentariu>]	- etichetă are un nume ilegal - valoarea iniţială specificată nu este validă	da	<etichetă><:> DEFW <val. iniţială msb> <val. iniţială lsb> [;<comentariu>]	- eticheta are un nume ilegal - valoarile iniţiale specificate nu sunt valide
Comentarii multi linie	da	COMMENT <delim> <corp comentariu> <delim>		nu		

- **Macrouri şi macrouri cu parametri**

Majoritatea limbajelor de asamblare suportă definirea de macrouri, care permit scrierea unui cod mai compact şi mai uşor de urmărit. în general un macro este delimitat de pseudointructiunea de început de macro şi cea de final de macro. Macrourile pot accepta şi parametri, precum în exemplul de mai jos pentru Intel x86:

```
SWAP MACRO P1, P2
    MOV EAX, P
```

```
MOV EBX, Q
MOV Q, EAX
MOV P, EBX
ENDM
```

Din acest exemplu putem extrage un format generalizat pentru macrouri:
<nume_macro> <pseudoinstrucţiune_macro> [<parametru> <delim. parametru>] ...
<corp_macro>
<pseudoinstrucţiune_final_macro>

Dintre erorile ce pot fi raportate în definiţia unui macro putem aminti necesitatea închiderii macroului până la finalul fişierului sursă.

- ***Descriere funcţională***

- **Rezolvarea macrourilor**

După cum s-a explicat n capitolele anterioare, prima parcurgere este responsabilă de construirea tabelei de simboluri şi de înlocuirea definiţiilor de macro. Pentru o uşurare a aspectelor de implementare, fără a se face un mare compromis din punctul de vedere al performanţei, am optat pentru o strategie în care macrourile sunt tratare într-o trecere preliminară. Astfel, mai întâi se determină şi se elimin toate definiţiile de macrouri din cod:

1. se caută în codul sursă apariţia primei pseudoinstrucţiuni de început de macro
2. pe linia respectivă se identifică elementele descrise în §1.2.3 (etichetă, parametri), se fac verificările aferente acestora şi se raportează eventualele erori
3. se elimină linia din cod
4. se caută de la acel punct înainte apariţia primei pseudoinstrucţiuni de final macro; se memorează linie cu linie codul parcurs (conţinutul macroului) şi se elimină din codul original, mai puţin linia care conţine directiva de final de macro care doar se elimin
5. se memorează într-o structură de date temporară numele macroului, numărul şi numele parametrilor şi conţinutul
6. se continuă de la pasul 1

Pentru fiecare macro identificat:

1. se parcurge codul şi se caută o referentiere a numelui macroului
2. când se găseşte o astfel de linie se identifică parametrii macroului
3. se verifică şi se raportează eventualele erori legate de numărul de parametri

4. parametrii se înlocuiesc în conţinutul macroului
5. linia curentă din codul sursă se înlocuieşte cu corpul macroului
6. se continuă de la pasul 1

- **Crearea tabelei de simboluri**

După efectuarea operaţiunilor descrise în §1.3.1 avem un cod în care există numai instrucţiuni şi directive de asamblare (pseudoinstrucţiuni). Parcurgem linie cu linie codul pentru a construi tabela de simboluri şi pentru a interpreta eventualele directive. Asftel, pentru fiecare linie din cod:

1. dacă linia se potriveşte formatului unei instrucţiuni (§1.2.1) atunci se memorează într-o structură de date eticheta, operaţia(mnemonica), operanzii, numărul de locaţii ocupate de instrucţiunea respectivă
2. altfel, dacă linia trebuie să se potrivească unuia dintre modelele disponibile de pseudoinstrucţiuni, definite în §1.2.2; în acest caz se identifică tipul directivei şi se execută funcţionalitatea aferentă (dacă există - spre exemplu directiva COMMENT a asamblorului nu are nici o funcţionalitate asociată, este necesară numai detectarea şi raportarea unei eventuale erori cauzate de lipsa delimitatorului de terminare)
3. altfel se semnalează o eroare - construcţia de cod nu a putut fi recunoscută

în continuare, se parcurg în ordine instrucţiunile, se calculează pentru fiecare valoarea ILC (Instruction Location Counter), iar pentru cele etichetate valoarea acestuia se memorează în structura de date aferenţă.

în capitolul legat de aspecte de implementare se tratează mai extins agregarea structurilor de date menţionate aici în liste înlănţuite care reprezintă de fapt tabela de simboluri a asamblorului. Astfel se rezolvă problema de forward-referencing, întâlnită la translatarea efectivă: dacă într-o instrucţiune unul dintre operanzi este un simbol necunoscut (altceva decât un număr sau registru) atunci acesta este căutat în “tabela de simboluri”, adică în cele două liste construite în acest pas.

- **Translatarea instrucţiunilor în cod maşină**

Se parcurg în ordine intrucţiunile şi directivele identificate. Dacă o instrucţiune conţine o referire la un simbol altul decât registru sau număr, atunci simbolul este căutat în lista de simboluri şi înlocuit corespunzător - cu ICL-ul în cazul unei instrucţiuni, cu valoarea simbolului dacă este vorba spre exemplu de o directivă EQU etc.
în acest moment putem spune că fişierul este asamblat. Pentru a obţine fişierul de ieşire mai este necesară o ultimă parcurgere, în care pentru fiecare instrucţiune se scrie în fişierul de ieşire codul operaţiei şi codul operanzilor săi (fie luaţi din tabelele de configurare, fie calculaţi după cum s-a descris mai devreme).

- *Aspecte de implementare şi componente arhitecturale*

- **Arhitectura generală a asamblorului**

Arhitectura generală a asamblorului împreună cu procesul de asamblare sunt prezentate în figură de mai jos:

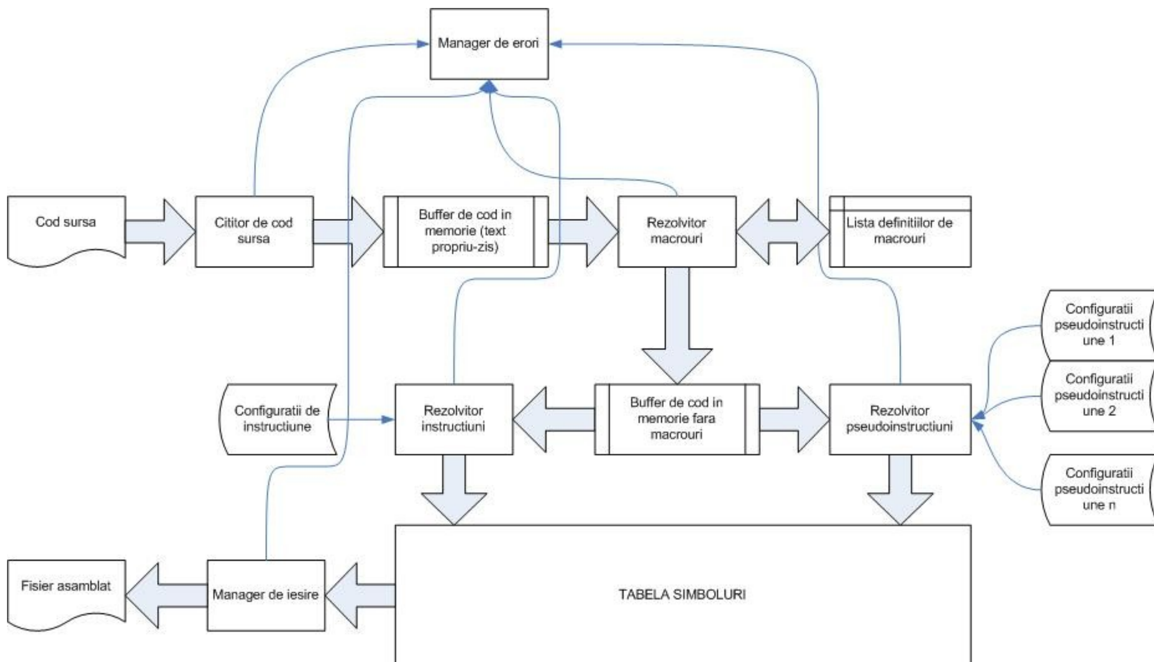

Fig. 1.4.1 Arhitectura generală a asamblorului

Se poate observa că toate componentele arhitecturale au un corespondent funcţional prezentat în §1.3. în plus, apar drept componente externe configuraţiile de instrucţiune şi de pseudinstrucţiuni (acestea ar trebui editate şi modificate de utilizator pentru a adapta asamblorul la limbajul dorit) şi un manager de erori, care este responsabil cu raportarea într-un mod uniform a erorilor întâlnite la orice nivel al procesării fişierului de intrare. Spre exemplu, dacă se întâlneşte o eroare la nivelul rezolvitorului de instrucţiuni, managerul de erori este responsabil să semnaleze eroarea la linia corectă (întrucât la nivelul respectiv deja se lucrează pe un buffer în care numărul liniilor a fost deja modificat de rezolvitorul de macrouri).

- **Cititorul de cod sursă**

Un aspect important din este modul în care se tratează codul sursă: linie cu linie prin citire din fişier, sau citirea întregului cod de la bun început într-un buffer de memorie, şi citirea ulterioară din bufferul respectiv. Fiecare variantă are avantaje şi dezavantaje: prima varianta este limitată din punctul de vedere al vitezei (accesul la disc este lent în raport cu accesul la memorie) iar a doua poate

ajunge la un consum foarte mare de memorie pentru fişiere de intrare mărfi. O altă variantă ar fi una combinată (tehnica “lazy” - prelucrare pe blocuri) în care se execută citri de blocuri de cod de pe disc în memorie în funcţie de referentierile care sunt făcute în cod, şi reprezintă un compromis între celelalte două.

Având în vedere însă că în general programatorii sunt amatori ai modularizarii codului pe blocuri componente funcţionale - deci fişierele sursă nu vor avea dimensiuni foarte mari - şi cantităţile mari de memorie (de ordinul GB) de care dispun maşinile acutale, se va obţine probabil un câştig important de viteză dacă se optează pentru tehnica citirii întregului fişier sursă într-un buffer de memorie, şi lucrul ulterior numai cu acest buffer.

- **Rezolvitorul de macrouri**

Un avantaj al separării rezolvării macrourilor este acela că după realizarea §1.3.1 memoria alocată temporar pentru acest pas poate fi eliberată, pentru pasul următor fiind utilizat numai bufferul ce conţine codul propriu-zis cu macrouri expandate. Practic lista definiţiilor de macrouri (vezi fig. 1.4.1).

Ar mai fi de menţionat că prima parcurgere este foarte rapidă, încât se caută o sigură cheie per linie (directiva de început de macro - de ex MACRO pentru Intel x86). Consumul real de resurse de calcul apare la înlocuirea parametrilor de macro şi a “apelurilor” de macro din cod - când se face şi o verificare a corectitudinii numărului de parametri - când se foloseşte căutare de patternuri (regular expression matching).

- **Tabela de simboluri**

Construirea tabelei de simboluri este pe departe cel mai consumatoar dintre procesele implicate în această arhitectură. Pentru fiecare linie de cod se realizează potrivirea de modele luate din configuraţia sistemului (pattern matching) cu liniile de intrare, iar această operaţie este mai complicată şi mai consumatoare de resurse decât o simplă căutare.

Pentru realizarea tabelei de simboluri este necesară o structură de date care să conţină eticheta, operaţia (mnemonica), operanzii, numărul de locaţii de memorie ocupate de instrucţiune. Pentru optimizarea folosirii resurselor, mnemonica poate fi un pointer într-un tabel care conţine întregul set de instrucţiuni al limbajului şi pentru fiecare instrctiune codul maşină aferent si numarul de locatii ocupate. In cazul pseudoinstructiunilor tratarea este diferita, intrucat fiecare pseudoinstructiune are o semnificatie aparte, specificata in configuratia corespunzatoare. De exemplu pseudoinstrucţiunea EQU a MASM va genera o structură de date specifică, în care este memorată eticheta şi valoarea acesteia. Toate aceste structuri de date generate în §1.2.1 si §1.2.2 sunt memorate într-o listă înlăntuită pentru a putea fi uşor parcurse şi accesate ulterior.

- ***Referinţe bibliografice***

[Ref bibl Y] http://www.nongnu.org/z80asm/directives - Documentaţia unui asamblor pentru Z80 dezvoltat ca aplicaţie GNU sub licenţa GPL

http://www.regular-expressions.info – cea mai importanta resursa online pentru informatii despre regular expressions