ASC – Lab synthesis

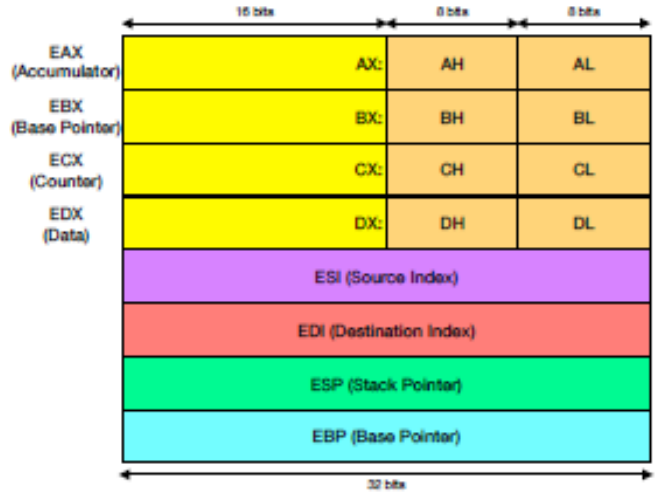
I.Limbaj

Assembly x86 este un limbaj de programare de nivel scazut (low-level programming language) care permite controlul direct asupra unitatii centrale de calcul (CPU) a unui calculator cu arhitectura de calcul x86. In multe limbaje de programare de nivel inalt compilatorul creeaza prima data cod assembly ca un pas intermediar. Codul sursa assembly este o bucata de text, deci un procesor nu are cum sa interpreteze si sa execute acest text. Assembly este transformat apoi in cod masina (machine code): instructiuni masina binare codate clar (opcode) care pot fi apoi executate de procesor pascu pas. Vrem sa va atragem atentia ca Assembly x86 are doua sintaxe distincte: Intel (folosit pe sisteme Windows) si AT&T (folosit predominant pe sisteme Unix). Diferentele intre cele doua nu sunt semnificative (este vorba doar de o sintaxa putin diferita) dar le puteti consulta online[8]. In acest laborator folosim doar sintaxa AT&T. Daca intelegeti una dintre sintaxe, cealalta e foarte usor de citit si inteles.

Componentele principale ale limbajului Assembly x86 sunt:

- Registrii. Ca in orice limbaj de programare, avem nevoie de variable cu care sa lucram. Aceste variable stocheaza date (in cazul nostru, valori pe 32 biti). Putem seta si modifica valorile registrilor si putem realiza operatii cu si intre acesti registri (de exemplu: putem aduna valorile din doi registri si il putem pune intr-un al treilea, sau in unul dintre registrii care deja stocheaza un operand). Unii registri stocheaza o locatie in memorie (de exemplu: trebuie sa stim undese afla urmatoarea instructiune din program care trebuie executata, deci avem un Instruction Pointer - e defapt un “Instruction Register”);
- Instructiuni. Procesorul suporta o serie de operatii intre registrii si locatii din memorie. In primul rand avem nevoie de abilitatea de a muta din/in registri date in/din memorie. In registri, putem face operatii matematice, logice, I/O, etc.
- Flaguri. Dupa ce facem o operatie, avem o serie de flag-uri (indicatori) care se activeaza in functie de rezultat (daca rezultatul operatilor este negativ sau zero, sau daca un overflow s-a produs);
- Adresarea memoriei. In procesor avem doar cativa registri. Daca avem nevoie sa lucram cu un volum mare de date (de exemplu, continutul unui fisier) atunci avem nevoie sa ne putem referi la o adresa din memorie (*Random Access Memory - RAM*). Accesul la registri este rapid, dar citirea/scrierea datelor din/in RAM este o operatie mult mai lenta ca timp.
- Stiva programului. Orice program care se afla in executie are o locatie speciala de memorie care se numeste stiva (*the program stack*). Aceasta este o structura de date de tip *Last In First Out (LIFO)*. Vom vedea mai tarziu exemple unde aceasta structura este folositoare (apelul de functii in Assembly);
- Intreruperi. Programul pe care voi il executati (indiferent de limbajul de programare) nu are acces direct la toate resursele hardware. In cele mai multe cazuri, sistemul de operare controleaza accesul la memorie si periferice. De aceea, avem nevoie sa cerem sistemului de operare sa execute niste operatii pentru programul nostru. Aceasta delegare a muncii catre sistemul de operare se realizeaza prin intreruperi.

II.Registrii



Registrul	Nume	Descriere	Restaurat dupa apel
AX	accumulator	este utilizat in operatii aritmetice	nu
BX	base	utilizat ca pointer la date	da
CX	counter	este utilizat in instructiunile repetitive	nu
DX	data	este utilizat in operatii aritmetice si I/O	nu
SP	stack pointer	pointer la varful stivei	da
BP	stack base pointer	pointer la baza stivei	da
SI	source index	pointer la sursa in operatii stream	da
DI	destination index	pointer la destinatie in operatii stream	da

Restaurat dupa apel = DA ➔ push si pop in procedura

III.Adrese

Pentru a reprezenta adresele de memorie, putem utiliza una dintre urmatoarele variante:
• valoarea unui registru pe 32 de biti: (%eax);
• suma dintre constanta numerica si valoarea unui registru: 4(%eax) (inteles ca %eax + 4);
• suma dintre doi registri: (%eax, %edx) (inteles ca %eax + %edx);
• suma dintre doi registri si o constanta numerica: 4(%eax, %edx) (inteles ca %eax + %edx +4);
• suma a doi registri, dintre care unul inmultit cu 2, 4 sau 8, la care se poate aduna o constanta: 16(%eax, %edx, 4) (inteles ca %eax + 4 * %edx + 16);

IV. Tipurile principale de date

1. **byte** - dupa cum ii spune si numele, ocupa 1 byte, adica 8 biti in memorie;
2. **single** - ocupa 4 bytes (32 de biti) si este utilizat pentru stocarea numerelor fractionare;
3. **word** - ocupa 2 bytes (16 biti) si este utilizat pentru stocarea intregilor;
4. **long** - ocupa 4 bytes (32 de biti) si este utilizat pentru a stoca intregi pe 32 de biti sau numere fractionare pe 64 de biti. **Double word** (dword) ocupa tot 32 de biti;
5. **quad** - ocupa 8 bytes (64 de biti);
6. **ascii** - este utilizat pentru declararea sirurilor de caractere care nu sunt finalizate cu termina- torul de sir - nerecomandat;
7. **asciz** - este utilizat pentru declararea sirurilor de caractere care sunt finalizate cu terminatorul de sir;
8. **space** - defineste un spatiu in memorie, a carui dimensiune o specificam, de exemplu **.space 4**, insemnand ca se lasa 4 bytes = 32 de biti. Este util atunci cand declaram variabile pe care le calculam in cadrul programului si pentru care nu vrem initial o valoare default.

Instructiune	Efect
add op1, op2	op2 := op2 + op1
sub op1, op2	op2 := op2 - op1
mul op	(edx, eax) := eax × op
imul op	(edx, eax) := eax × op
div op	(edx, eax) := (edx, eax) / op
idiv op	(edx, eax) := (edx, eax) / op

Observatie: pentru `div`, scrierea `(edx, eax)` inseamna ca se imparte $2^{32} * edx + eax$ la `op`. Daca nu vrem sa impartim un numar foarte mare, trebuie sa punem in `edx` valoarea 0 initial.

Explicatii inmultire si impartire.

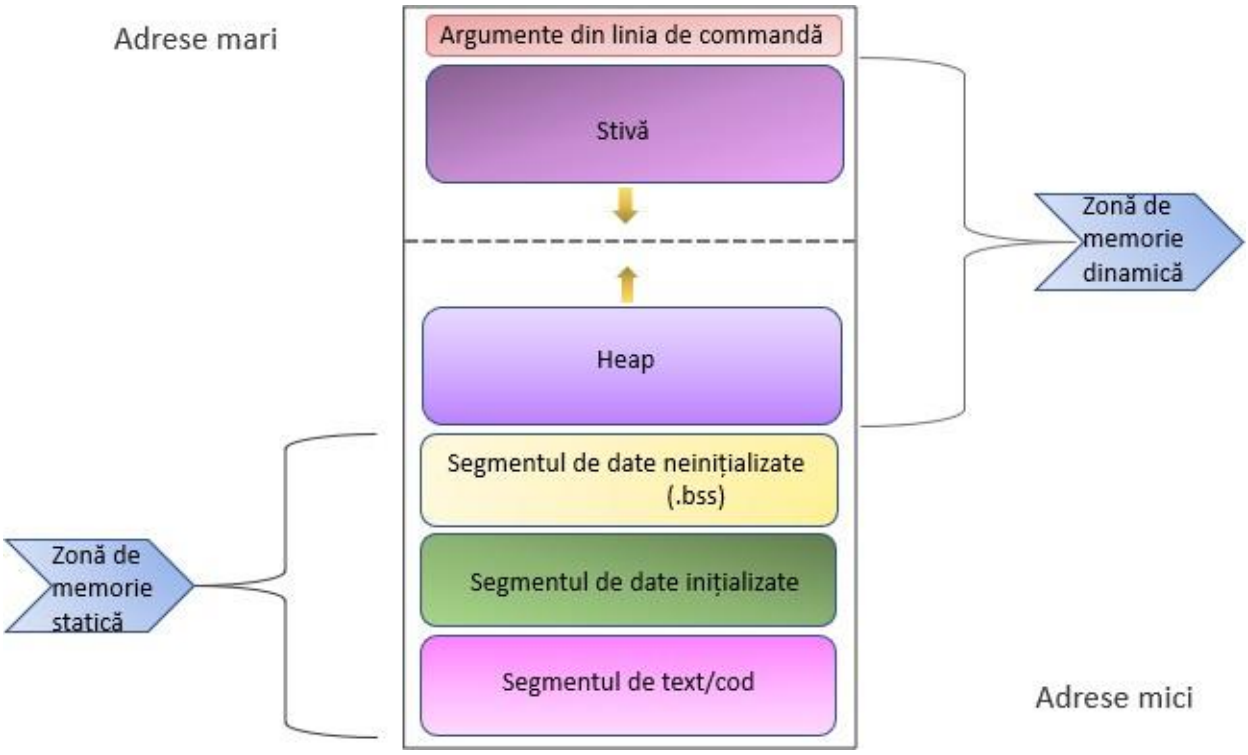
- atat inmultirea, cat si impartirea au operanzi impliciti, acestia fiind EDX si EAX. Pentru a inmulti un numar cu o anumita valoare (sau a imparti la un numar o anumita valoare), trebuie sa avem deja informatia completata in registrul EAX;
- trebuie sa diferentiem intre MUL si IMUL, intre DIV si IDIV, instructiunea precedata de I fiind pentru prelucrarea numerelor **cu semn**;
- pentru inmultire, rezultatul este dat de $2^{32} \times edx + eax$;
- pentru impartire, eax stocheaza catul, iar edx stocheaza restul;
- in cazul inmultirilor si al impartirilor, operandul *op* **NU** poate fi o constanta numerica! In rest, *op* poate fi registru sau locatie de memorie.

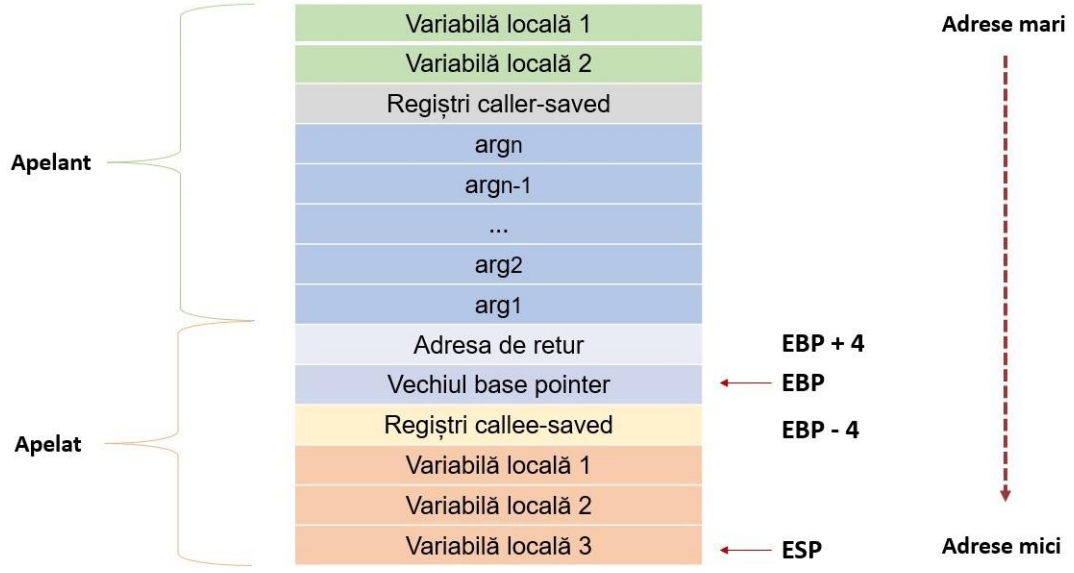
Instructiune	Efect
not op	op := ~op
and op1, op2	op2 := op2 & op1
or op1, op2	op2 := op2 | op1
xor op1, op2	op2 := op2 ^ op1

Operatiile logice de deplasare sunt **shr**, **shl**, **sar** si **sal**, toate avand structura `operatie destinatie, numar` unde *numar* este cati biti sunt deplasati. In numele instructiunii, s este *shift*, distinctia l/r este pentru left/right (stanga/dreapta) iar distinctia h/a este pentru logic/arithmetic (deplasare pe biti sau aritmetica). In cazul deplasarilor aritmetice, bitul cel mai semnificativ isi pastreaza valoarea (se pastreaza semnul numarului).

Instructiune	Efect
shr numar, op	op := op $\gg$ numar
shl numar, op	op := op $\ll$ numar
sar numar, op	op2 := op $\gg$ numar (cu pastrare semn op)
sal numar, op	op2 := op $\ll$ numar (cu pastrare semn op)

Operator	Descriere
jc	jump daca este carry setat
jnc	jump daca nu este carry setat
jo	jump daca este overflow setat
jno	jump daca nu este overlow setat
jz	jump daca este zero setat
jnz	jump daca nu este zero setat
js	jump daca este sign setat
jns	jump daca nu este sign setat
Operatori pentru numere fara semn	
jb	jump if below (op1 < op2)
jbe	jump if below or equal (op1 <= op2)
ja	jump if above (op1 > op2)
jae	jump if above or equal (op1 >= op2)
Operatori pentru numere cu semn	
jl	jump if less than (op1 < op2)
jle	jump if less than or equal (op1 <= op2)
jg	jump if greater than (op1 > op2)
jge	jump if greater than or equal (op1 >= op2)
Operatori de egaliate	
je	jump if equal (op1 == op2)
jne	jump if not equal (op1 ! = op2)


Adrese mari
Argumente din linia de commandă
Stivă
Heap
Segmentul de date neinițializate (.bss)
Segmentul de date inițializate
Segmentul de text/cod
Zonă de memorie dinamică
Zonă de memorie statică
Adrese mici


Variabilă locală 1
Variabilă locală 2
Registri caller-saved
argn
argn-1
...
arg2
arg1
Adresa de retur
Vechiul base pointer
Registri callee-saved
Variabilă locală 1
Variabilă locală 2
Variabilă locală 3
Apelant
Apelat
EBP + 4
EBP
EBP - 4
ESP
Adrese mari
Adrese mici