TUTORIAT ASC 2

MARIA PREDA

1. Adunarea si scaderea in sistemul binar

Asemanator cu sistemul zecimal, unde, atunci cand depasim cifra 9, punem 0 pe acea pozitie si adaugam 1 la cifra de pe pozitia imediat la stanga, in sistemul binar, atunci cand depasim cifra maxima, 1, punem 0 pe pozitia curenta si "tinem 1 in minte", pe care il adaugam la cifra din stanga.

Exemplu 1.1. *Trebuie sa adunam numerele* 10101010 *si* 1010101*, pe care le consideram fara semn:*

$$\begin{array}{r} 10101010 \\ +\ 1010101 \\ \hline 11111111 \end{array}$$

In acest caz, nu am avut nicio pozitie pe care sa avem valori 1 *in ambele numere.*

Exemplu 1.2. *Trebuie sa adunam numerele* 0101110011110011 *si* 0111000011110000*:*

$$\begin{array}{r} 0101110011110011 \\ +\ 0111000011110000 \\ \hline 1100110111100011 \end{array}$$

Scaderea se face in mod similar. Daca trebuie, la o anumita pozitie, sa scadem 1 din 0, atunci punem 1 si scadem cu 1 mai mult decat ar fi trebuit pe pozitia de la stanga celei curente ("ne imprumutam" cu 1 la cifra imediat mai semnificativa).

Exemplu 1.3. *Trebuie sa scadem numerele* 10101010 *si* 1010101*, pe care le consideram fara semn:*

$$\begin{array}{r} 10101010 \\ -\ 1010101 \\ \hline 1010101 \end{array}$$

2. Operatii logice pe biti

Cu toate ca nu mai aveti semestrul acesta cursul de "Logica Matematica si Computationala", probabil ati mai intalnit in liceu operatiile: not ($\neg$), and ($\wedge$) si or($\vee$). Atunci le foloseati pentru "Adevar" si "Fals", pe care uneori le notati si 1 (adevarat), 0 (fals) Acestea respectau urmatoarele tabele:

Intrare (A)	NOT A
0	1
1	0

A	B	A $\wedge$ B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A $\vee$ B
0	0	0
0	1	1
1	0	1
1	1	1

Acum, vom adauga o noua operatie: "sau disjunctiv", cunoscuta si ca xor. Ideea acestei operatii este ca avem 1 (Adevar), doar daca una singura dintre intrari este 1 (Adevar), ceea ce inseamna ca urmeaza urmatorul tabel:

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

In cazul nostru, vrem sa putem utiliza aceste operatii pentru numere scrise in baza 2, adica pentru siruri de biti. Astfel, daca dorim sa aplicam operatia intre doua numere, vom aplica operatia pe perechi de cate doi biti, acestia trebuind sa fie pe aceeasi pozitie, de la dreapta la stanga.

Exemplu 2.1. *Aplicati operatia not pentru fiecare dintre cele doua numere, iar operatiile and, or, xor intre ele, numerele fiind:* 1001101 *si* 1010110.

$$\neg 1001101 \Rightarrow 110010$$

Avem cu una mai putine cifre, deoarece primul 1 *din stanga devine* 0*, deci nu mai este necesara scrierea sa.*

$$\neg 1010110 \Rightarrow 101001$$

$$\begin{array}{r} 1001101 \\ AND\ 1010110 \\ \hline 1000100 \end{array}$$

$$\begin{array}{r} 1001101 \\ OR\ 1010110 \\ \hline 1011111 \end{array}$$

$$\begin{array}{r} 1001101 \\ XOR\ 1010110 \\ \hline 0011011 \end{array}$$

Observaţie 2.2. *In cazul in care numerele nu au acelasi numar de cifre, vom completa cu* 0 *la stanga numarului cu mai putine, pana cand ajung de aceeasi lungime.*

In Assembly, operatiile isi pastreaza numele. Pentru operatia logica "not", avand aritate 1 (adica se aplica unui singur numar), vom avea un singur operand pe care il punem. Sintaxa: *not operand*. Celelalte trei au aritatea 2, adica se aplica intre doua numere (la fel ca adunarea, impartirea etc). Vom avea sintaxa: *operatie operand*1, *operand*2.

Exemplu 2.3. *xor* $2, %eax

3. Operatii de shift

Operatiile de inmultire si impartire sunt destul de lente, asadar, in cazul in care este posibil, dorim sa evitam sa le facem. In acest sens, introducem ideea de shift pe biti, care, asa cum ii spune si numele, reprezinta o deplasare a tuturor bitilor spre stanga sau spre dreapta.

Shift cu n biti catre stanga va fi, de fapt, o inmultire a numarului initial cu 2^n, in timp ce shift cu n biti catre dreapta va fi o impartire (doar catul, nu si restul) a numarului initial la 2^n.

Exemplu 3.1. *Vrem sa shiftam numarul* 25 *cu* 3 *biti catre stanga. Primul pas este sa transformam numarul in baza* 2*, apoi sa deplasam toti bitii cu* 3 *biti la stanga, ceea ce inseamna adaugarea a* 3 *biti* 0 *la stanga celor existenti.*

$$25_{(10)} \rightarrow 11001 \rightarrow 11001000 = 200_{(10)}$$

Intuitie*: Imaginati-va ca avem un sir de cutii goale. Pe ultimele pozitii, in ultimele* 5 *cutii, punem in cutii* 0 *sau* 1*, pentru a avea sirul* 11001 *(le avem la finalul dreapta al sirului de cutii). Atunci cand dam shift cu* 3 *cutii spre stanga, inseamna ca vom avea ultimele* 3 *cutii din dreapta goale. Dar nu pot fi goale daca la stanga lor mai avem cutii pline, acesta nu ar mai fi un numar, asa ca punem* 0 *si obtinem sirul* 11001000*. Este un principiu asemanator cu cel de pe banda dintr-o masina clasica Turing.*

De ce este echivalentul inmultirii? (Optional)

Pornind de la exemplul de deasupra, atunci cand adaugam 3 biti 0 la dreapta, inseamna ca fiecare dintre bitii nenuli dinainte va fi inmultit cu o putere a lui 2 cu exponent cu 3 mai mare decat era la inceput, ceea ce inseamna ca fiecare termen din suma va fi de 2^3 ori mai mare, adica este ca si cum am inmulti intreg numarul initial cu 8.

Exemplu 3.2. *Vrem sa shiftam numarul* 37 *cu* 2 *biti catre dreapta. Primul pas este sa transformam numarul in baza* 2*, apoi sa deplasam toti bitii cu* 2 *biti la dreapta. Deoarece spre dreapta nu mai avem pozitii libere, inseamna sa taiem* 2 *biti de la dreapta numarului.*

$$37_{(10)} \rightarrow 100101 \rightarrow 1001 = 9_{(10)}$$

In assembly, distingem doua tipuri de shift, cel cu semn si cel fara semn, pentru a se adapta numerelor negative reprezentate in complement fata de doi. De exemplu, daca avem un numar negativ pe care il shiftam la dreapta, bitii de la stanga de tot ar deveni 0, iar asta ne-ar face sa pierdem informatia ca numarul este negativ.

Avem:

(1) shr numar, numar_biti_shift → shift fara semn, spre dreapta (right)

(2) shl numar, numar_biti_shift → shift fara semn, spre stanga (left)

(3) sar numar, numar_biti_shift → shift cu semn, spre dreapta (right)

(4) sal numar, numar_biti_shift → shift cu semn, spre stanga (left)

4. Afisarea de mesaje - Assembly

Pentru a realiza afisarea unui mesaj, vom avea nevoie de un apel de sistem. Astfel, vom apela functia Write, care are codul 4. De asemenea, aceasta trebuie sa aiba ca argumente: locul in care sa afiseze mesajul, in cazul nostru parametrul va fi 1, adica in consola, mesajul de afisat si lungimea mesajului (in bytes).

Observaţie 4.1. *Atunci cand folosim tipul de date asciz, trebuie sa adaugam* 1 *la lungimea sirului. Trebuie adaugat pentru a include si caracterul, pe care noi nu il vedem, de terminare a sirului. In cazul tipului de date ascii, acest lucru nu este necesar.*

Asadar, vom avea sintaxa:

```
mov $4, %eax
mov $1, %ebx
mov $nume_variabila, %ecx
mov $constanta(lungime sir), %edx
int $0x80
```

Observaţie 4.2. *Atunci cand vrem sa folosim o variabila de tip asciz sau ascii, trebuie sa o prefixam cu "$", aceasta semnifica preluarea adresei din memorie pentru variabila respectiva. Nu vom pasa sirul de caractere ca valoare, ci ca adresa, pe care o va folosi apoi registrul cand are nevoie de "valoarea" sirului de caractere.*

Exemplu 4.3. *Avem str: .asciz "ASC". Afisarea corecta:*

```
mov $4, %eax
mov $1, %ebx
mov $str, %ecx
mov $4, %edx
int $0x80
```

Exemplu 4.4. *Avem str: .ascii "ASC". Afisarea corecta:*

```
mov $4, %eax
mov $1, %ebx
mov $str, %ecx
mov $3, %edx
int $0x80
```

Observaţie 4.5. *Daca numarul de caractere pe care il punem in registrul %edx este mai mare decat numarul de caractere din variabila al carei continut vrem sa il afisam, atunci se va continua in memorie. Memoria fiind continua, in cazul in care noi am mai declarat alte variabile de acelasi tip imediat sub, atunci se vor lua caracterele din ele de care mai este nevoie. In caz ca sunt de alt tip, de exemplu numere, atunci aceste valori vor fi interpretate ca fiind coduri ASCII. Atentie: se vor lua byte cu byte, pentru ca atat ocupa un caracter ASCII.*

Exemplu 4.6. *Consideram urmatoarele declaratii in sectiunea .data:*

```
str1: .ascii "ASC"
str2: .ascii "FMI"
```

Si urmatoarea afisare:

```
mov $4, %eax
mov $1, %ebx
mov $str1, %ecx
mov $5, %edx
int $0x80
```

str1 are doar 3 *caractere, iar noi incercam sa afisam* 5*, asadar, se va trece si la sirul declarat dupa in memorie, adica str2.*
Se va afisa: ASCFM.

Inainte de a da un exemplu pentru cazul in care str2 nu ar fi tot ascii, ci de alt tip, trebuie sa vorbim despre reprezentarea octetilor in memorie. Arhitectura x86 foloseste sistemul little-endian, ceea ce inseamna ca octetii sunt stocati in ordine inversa in memorie. De exemplu, pentru numarul $0X31323334$, am avea octetii (in hexazecimal): 31 32 33 34. De fapt, in memorie o sa ii avem 34 33 32 31.

Exemplu 4.7. *Consideram urmatoarele declaratii in sectiunea .data:*

```
str1: .ascii "ASC"
str2: .long 0X31323334
```

Si urmatoarea afisare:

```
mov $4, %eax
mov $1, %ebx
mov $str1, %ecx
mov $5, %edx
int $0x80
```

str1 are doar 3 *caractere, iar noi incercam sa afisam* 5*, asadar, se va trece si la sirul declarat dupa in memorie, adica str2. Va interpreta octetii luati din str2 ca fiind coduri ASCII si va afisa: ASC43, deoarece:*

$$0X31 \rightarrow 49_{(10)} \rightarrow 1 \ (in\ ASCII)$$
$$0X32 \rightarrow 50_{(10)} \rightarrow 2 \ (in\ ASCII)$$
$$0X33 \rightarrow 51_{(10)} \rightarrow 3 \ (in\ ASCII)$$
$$0X34 \rightarrow 52_{(10)} \rightarrow 4 \ (in\ ASCII)$$

Asadar, se va trece la primi doi octeti din str2, care, asa cum am mentionat mai devreme sunt reprezentati ca fiind:

34 33 32 31

Deci se iau octetii 34 *si* 33*, motiv pentru care, dupa str1, se mai afiseaza* 4 *si* 3*.*

5. Inmultiri si impartiri in Assembly

Vom incepe prin a discuta despre operatia mul, care se foloseste pentru inmultire si are sintaxa urmatoare:

mul operand

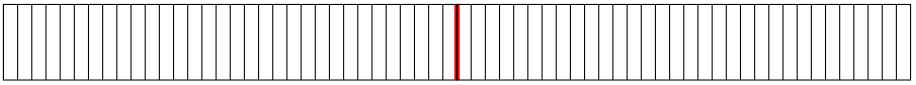
Operandul pe care il avem dupa mul este unul dintre cele doua numere pe care le inmultim. Celalalt numar este valoarea stocata in registrul %eax. Deci, pentru a face o inmultire, trebuie intai sa punem una dintre valori in %eax.

Observaţie 5.1. *Operandul folosit nu poate sa fie o constanta numerica. Totusi, daca folosim un sufix pentru a preciza tipul de date, putem folosi si constante numerice.*

Exemplu 5.2. *Exemplu de inmultire:*

```
mov $4, %eax
mov $5, %ebx
mul %ebx
```

In continuare, rezultatul va fi stocat in registrii %edx si %eax concatenati in felul urmator:



%edx %eax

Ne putem imagina ca avem un registru pe 64 de biti in loc de 32. Astfel, daca rezultatul impartirii este mai mic decat $2^{32}-1$, atunci in %edx vom avea 0, iar in %eax rezultatul. Altfel, rezultatul va fi $\%edx \cdot 2^{32} + \%eax$. De ce inmultim valoarea din %edx cu 2^{32}? Pentru ca este ca si cum bitii lui %edx ar fi de la bit-ul al 33-lea la al 64-lea.

Exemplu 5.3. *Continuarea exemplului 5.2:*
Rezultatul inmultirii va fi 20*, care are reprezentarea* 10100 *in binar.*

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0

Primii 32 *de biti de la stanga corespund lui %edx, resul lui %eax.*

Observaţie 5.4. *Operatorul mul este pentru numere fara semn. Pentru a face o inmultire tinand cont de semnul numerelor, vom folosi operandul imul, care functioneaza similar.*

Pentru impartire, se foloseste operandul div, care are urmatoarea sintaxa:

div operand

Observaţie 5.5. *Operandul folosit nu poate sa fie o constanta numerica. Totusi, daca folosim un sufix pentru a preciza tipul de date, putem folosi si constante numerice.*

Acest operand este, de fapt, impartitorul din cadrul operatiei. Deimpartitul este format prin concatenarea, la fel ca la rezultatul inmultirii, a registrilor %edx si %eax, deci este un numar pe 64 de biti. In cazul in care

avem de impartit un numar mic, trebuie sa punem valoarea 0 in registrul %edx si valoarea pe care vrem sa o impartim in registrul %eax. Pentru valori mari, peste $2^{32} - 1$, va trebui sa privim deimpartitul ca fiind $\%edx \cdot 2^{32} + \%eax$.

Exemplu 5.6. *Exemplu pentru impartire:*

```
mov $21, %eax
mov $0, %edx
mov $5, %ebx
div %ebx
```

Dupa impartire, catul impartirii va fi stocat in %eax, iar restul in %edx.

Observaţie 5.7. *La fel ca in cazul inmultirii, daca lucram cu numere cu semn, avem nevoie de idiv.*

6. Salturi neconditionate

Salturile neconditionate in limbajul de asamblare x86 sunt instructiuni care permit programului sa sara la o anumita adresa de memorie, fara a verifica o conditie prealabila.

Un salt neconditionat este o instructiune care face ca procesorul sa continue executia de la o adresa specificata, in cazul nostru, de obicei, adresa unei etichete din program. Este utilizat frecvent in structuri repetitive.

In x86, instructiunea principala pentru salturile neconditionate este `jmp`. Exemple:

- **Salt la o eticheta:**

  ```
  jmp label
  ```

- **Salt la o adresa (putin probabil sa folositi):**

  ```
  jmp *%eax
  ```

Exemplu 6.1.

```
.data
.text
.global main

main:
    movl $3, %ecx
    jmp end          ; sare la sfarsit

    movl $4, %ecx

end:
    mov $1, %eax
    mov $0, %ebx
    int $0x80        ; apel de sistem
```

In acest exemplu, daca rulam cu gdb, punem un break point la eticheta end si folosim comanda i r ecx, vom observa ca valoarea lui este 3.

7. Salturi conditionate

Salturile conditionate sunt instructiuni care permit programului sa sara la o anumita adresa de memorie (de obicei adresa unei etichete) in functie de rezultatul unei comparatii anterioare.

In x86, instructiunile de salt conditionat sunt formate de obicei dintr-o comparatie (utilizand instructiunea `cmp`) urmata de un salt. Instructiunile de salt conditionat sunt:

operand	Descriere	Semn (Da/Nu)	Flaguri setate	Condiție Flag
jc	jump dacă este carry setat	Nu	CF (Carry Flag)	CF = 1
jnc	jump dacă nu este carry setat	Nu	CF	CF = 0
jo	jump dacă este overflow setat	Nu	OF (Overflow Flag)	OF = 1
jno	jump dacă nu este overflow setat	Nu	OF	OF = 0
jz	jump dacă este zero setat	Nu	ZF (Zero Flag)	ZF = 1
jnz	jump dacă nu este zero setat	Nu	ZF	ZF = 0
js	jump dacă este sign setat	Da	SF (Sign Flag)	SF = 1
jns	jump dacă nu este sign setat	Da	SF	SF = 0
jb	jump if below (unsigned $op2 < op1$)	Nu	CF	CF = 1
jbe	jump if below or equal (unsigned $op2 \leq op1$)	Nu	CF, ZF	CF = 1 or ZF = 1
ja	jump if above (unsigned $op2 > op1$)	Nu	CF, ZF	CF = 0 and ZF = 0
jae	jump if above or equal (unsigned $op2 \geq op1$)	Nu	CF	CF = 0
jl	jump if less than (signed $op2 < op1$)	Da	SF, OF	SF ≠ OF
jle	jump if less than or equal (signed $op2 \leq op1$)	Da	SF, OF, ZF	SF ≠ OF or ZF = 1
jg	jump if greater than (signed $op2 > op1$)	Da	SF, OF, ZF	SF = OF and ZF = 0
jge	jump if greater than or equal (signed $op2 \geq op1$)	Da	SF, OF	SF = OF
je	jump if equal ($op1 = op2$)	Nu	ZF	ZF = 1
jne	jump if not equal ($op1 \neq op2$)	Nu	ZF	ZF = 0

TABELA 1. Operandi de salt condiționat, flagurile setate și condițiile lor

7.1. **Instructiunea `cmp`.** Instructiunea `cmp` compara doua valori si seteaza anumite flaguri in registrul de stare (EFLAGS) pe baza rezultatului comparatiei. Sintaxa generala este:

```
cmp operand1, operand2
```

Dupa executarea acestei instructiuni, procesorul va evalua flagurile pentru a determina rezultatul comparatiei:

(1) ZF (Zero Flag): este setat daca operand1 este egal cu operand2.

(2) SF (Sign Flag): este setat daca rezultatul comparatiei este negativ.

(3) CF (Carry Flag): este setat daca operand1 este mai mic decat operand2.

Exemplu 7.1. *Un exemplu simplu care ilustreaza utilizarea salturilor conditionate si a instructiunii `cmp`:*

```
.data
.text
.global main

main:
    movl $0, %ecx
    movl $3, %eax        ; incarca 3 in EAX
    movl $5, %ebx        ; incarca 5 in EBX
    cmp %ebx, %eax       ; compara EAX cu EBX
    jge greater_or_equal  ; sare la 'greater_or_equal' daca EAX >= EBX

    # Cod pentru cazul in care EAX < EBX
    movl $1, %ecx        ; setam ECX la 1
    jmp end              ; sare la sfarsit

greater_or_equal:
    # Cod pentru cazul in care EAX >= EBX
    movl $2, %ecx        ; setam ECX la 2

end:
    movl $1, %eax
    xor %ebx, %ebx
    int $0x80
```

In acest exemplu, la finalul programului, %ecx va avea inca valoarea 1.

Cu ajutorul salturilor conditionate, putem obtine o functionalitate similara cu cea a unui if.

8. Exercitii

(1) Faceti urmatoarele calcule in sistem binar:

(a) $1001010 + 101101 =$

(b) $111111111010 + 1010101 =$

(c) $101010101 + 1001000 =$

(d) $110101010 - 1000101 =$

(e) $101010101 - 1001000 =$

(f) $1010101 \wedge 1001010 =$

(g) $100010101 \wedge 100010101 =$

(h) $1111 \wedge 101001 =$

(i) $101010 \vee 100010 =$

(j) $\neg 10101001 =$

(k) $\neg 1111 =$

(l) $101010 \vee 10101 =$

(m) 101010011001 *xor* $100010100110 =$

(n) 10010 *xor* $10110 =$

(o) 11111110101001001010 *xor* $11111110101001001010 =$

(2) Avem urmatorul program:

```
.data
    x1: .ascii "maria"
    x2: .ascii "maria1"
    x3: .ascii "maria2"
.text
.global main

main:

    mov $4, %eax
    mov $1, %ebx
    mov $x1, %ecx
    mov $14, %edx
    int $0x80

end:
```

```
    movl $1, %eax
    xor %ebx, %ebx
    int $0x80
```

Dupa executarea programului, ce se va afisa?

Soluţie:
Programul va afisa: mariamaria1mar. De ce?

Trebuie sa afisam 14 bytes, fiecare caracter ascii ocupa exact un byte. Sirul x1 are doar 5 bytes, deci se va continua cu ceea ce avem in memorie. Aceasta fiind liniara, inseamna ca sirurile x2 si x3 sunt la rand dupa x1, deci vor fi luati bytes din ele.

(3) Avem urmatorul program:

```
.data
    x1: .ascii "maria"
    x2: .long  0x61676362
    x3: .ascii "maria2"
.text
.global main

main:

    mov $4, %eax
    mov $1, %ebx
    mov $x1, %ecx
    mov $14, %edx
    int $0x80

end:
    movl $1, %eax
    xor %ebx, %ebx
    int $0x80
```

Dupa executarea programului, ce se va afisa?

Soluţie:
Programul va afisa: mariabcgamaria. De ce?

Trebuie sa afisam 14 bytes, fiecare caracter ascii ocupa exact un byte. Sirul x1 are doar 5 bytes, deci se va continua cu ceea ce avem in memorie. Aceasta fiind liniara, inseamna ca sirurile x2 si x3 sunt la rand dupa x1, deci vor fi luati bytes din ele. Pentru ca, atunci cand lucram cu numere, octetii sunt reprezentati invers in memorie, dupa ce sunt luati octetii, transformati in coduri ascii, vor aparea invers.

(4) Avem urmatoarea secventa de cod:

```
    mov $0x2, %eax
    mov $0x00000001, %edx
    mov $2, %ebx
    div %ebx

et_exit:
```

Daca executam programul cu gdb si scriem urmatoarele linii:

```
b et_exit
run
i r eax
i r edx
```

Ce valori vor fi afisate?

Soluţie:
Vom avea registrii %edx si %eax concatenati, deci, in total, vom avea valoarea: $2^{32} + 2$ (deimpartitul). valoarea la care impartim este 2, deci vom avea catul (in %eax) $2^{31} + 1$, iar restul (in %edx) 0. Acestea sunt valorile pe care le vom avea in cei doi registri atunci cand ajungem la eticheta et_exit.

Observatie: Asa cum am precizat si la tutoriatul trecut, atunci cand folosim "i r (nume_registru)" in gdb vom obtine valoarea registrului din acel punct al rularii.

(5) Fie urmatorul program:

```
.data

.text

.global main

main:
    mov $0xff67fff, %ecx
    mov $0x5263dfa, %eax

    cmp %ecx, %eax
    jge et_exit

et_1:
    mov $4, %ecx

et_exit:
    mov $1, %eax
    xor %ebx, %ebx
    int $0x80
```

Ce valoare va fi stocata in registrul %ecx atunci cand programul ajunge la eticheta et_exit.

Soluţie:
Valoarea stocata in registrul %ecx va fi $0xff67fff$.

"jge" este folosit pentru numere cu semn. Cum f este, in baza 2 reprezentat ca 1111, inseamna ca prima cifra a numarului din registrul %ecx este 1, deci numarul este negativ, in timp ce 5 este 0101, ceea ce inseamna ca numarul din registrul %eax este pozitiv, deci, automat, mai mare decat cel din %ecx.

$jge \%ecx, \%eax$ se interpreteaza ca fiind $\%eax \geq \%ecx$, ceea ce este adevarat, deci se face direct saltul la eticheta et_exit, asa ca valoarea din registrul %ecx ramane cea initiala.