Assembler proqramlaşdırma dili

x86 Linux (64 bit)

Əhməd Sadıxov

Buraxılış 1.0

24.04.2012

Önsöz

Prosessoru (**CPU**) kompüterin "ürəyi" və "beyni" hesab edə bilərik.Kompüterin yaddaşından və giriş/çıxış portlarından məlumatın oxunması, məlumatlar üzərində müxtəlif hesab, müqaisə, sürüşmə,məntiq v.s. əməliyyatların aparılması, emal olunmuş məlumatların təkrar yaddaşa (portlara) yazılması, hər-cür kəsilmələrin qəbul olunması və cavab verilməsi, habelə fiziki yaddaşın müxtəlif məlumat və instruksiya sahələrinə edilən müraciətlərə nəzarətin təmin olunması v.s. işlər məhs **CPU** tərəfindən həyata keçirilir.

CPU -nun dilinə proqramlaşdırma dilləri arasında ən yaxın dil Assembler dilidir. Yüksək səviyyəli dildə proqram tərtib edərkən **CPU** -nun əhəmiyyəti demək olar ki hiss olunmur. Buna səbəb isə kompilyator proqramlarıdır. Kompilyator bizim yüksək səviyyəli dildə yazdığımız proqramları assembler dilinə, daha sonra isə maşın dilinə , yəni obyekt koda çevirir.

Assembler dilində isə proqram tərtib edərkən yazdığımız kod cüzi dəyişikliyə uğrayaraq (maşın dilinə çevrilərək) birbaşa **CPU** tərəfindən icra olunur.

Bu kitab Assembler dilindən bəhs edir. Paralel olaraq **CPU**-nun arxitekturası, iş prinsipi, fiziki yaddaşın strukturu, **CPU** -nun yaddaşa müraciət metodları v.s. barədə nəzəri məlumatlar verilir və assembler dilində müvafiq proqram kodları ilə praktik nümunələr gətirilir.

İçindəkilər

$1 Prosessor - CPU

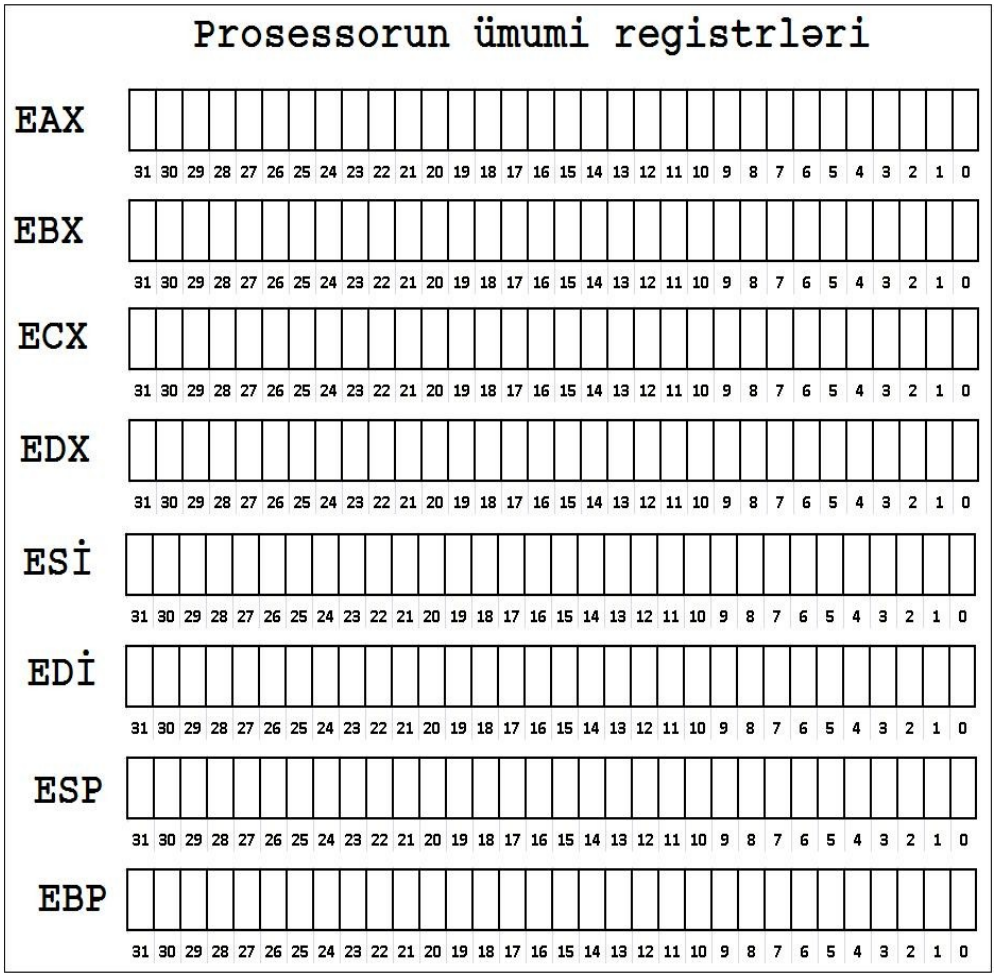
CPU(prosessor) ilə sadə tanışlıq

CPU -nu kiçik bir mikrosxemə yerləşdirilimiş böyük bir zavod kimi təsəvvür eləmək olar. Onun iş prinsipini, strukturunu izah eləmək üçün bir neçə kitab tələb olunur. **CPU** barədə ətraflı məlumat əldə etmək istəyənlərə üçün **Intel 80386 Programmer's Reference Manual**. kitablarını məsləhət görürəm.
CPU unit(vahid) adlandırılan bir neçə hissədən ibarətdir. Bunlardan ən əsas hissələr **Execution** (instruksiyaları icra edən), **İnstruction Decode** (instruksiyaları çevirən), **Segment** , **Paging** (yaddaşı idarə edən) və Proqramlaşdırmada ən çox istifadə olunan **Registers** (reqistrlər) hissəsidir. Reqistrlərdən başqa yerdə qalan bütün hissələr sırf sistem proqramlaşdırmaya aiddir. Yeganə reqistrlərdən istifadəçi proqramlaşdırmada istifadə olunur . Bundan əlavə istifadəçi proqramlarına prosessorun əməliyyatlar sisteminin müəyyən elədiyi kəsilmələr çağırmaq imkanı da verilir.
Burada biz assembler dilində istifadəçi proqramlaşdırmanı örgənəcəyik. Buna görə hələlik sadəcə reqistrləri bilməyimiz kifayətdir.

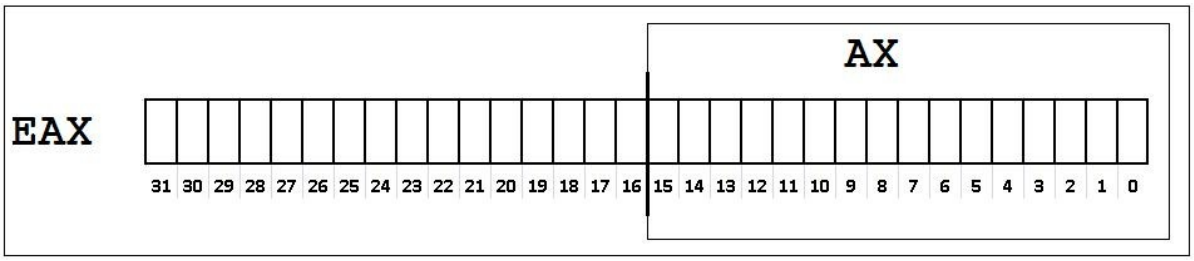
Registerlər

Reqistrlər prosessorun xarici dünya ilə əlaqə saxlaması üçündür. **CPU** ilə əlaqə saxlamaq üçün istifadə olunan ən əsas iki vasitədən biri məhs **CPU** -nun reqistrləridir, digər vasitə isə siqnallardır(kəsilmələr).
Qeyd: Nə qədər üzücü olsa da qeyd eləməliyəm ki, Windows əməliyyatlar sistemində hansısa (naməlum) məqsədlər üçün istifadə olunan Reqistr ifadəsi **CPU** -nun reqistrləri ilə qarışıqlıq yaradır. **CPU** -nun reqistrləri ilə Windowsda istifadə olunan reqistr faylları arasında qətiyyən heç bir əlaqə yoxdur, təmamilə ayrı məhfumlardır. Sadəcə olaraq anlamıram niyə Microsoftun proqramçıları bu qədər kritik əhəmiyyəti olan reqistrlər terminindən ayrı məqsəd üçün istifadə edirlər.
Gəlin registrlərlə daha yaxından tanış olaq
Registr çox kiçik ölçüyə malik olan, prosessora aid yaddaş sahəsidir. Prosessor hər-hansı proqramı icra etmək üçün fiziki yaddaşdan bu proqramın məlumat və instruksiyalarını kiçik hissələrlə əvvəlcə özünün reqistrlərinə köçürür, daha sonra emal etmək üçün digər hissələrə ötürür. Bu fikirdən yanaşsaq görərik ki, reqistrlərdən prosessor məlumatları müvəqqəti olaraq saxlamaq üçün istifadə edir. Registrlərin ölçüsü prosessorun arxitekturasından asılı olaraq 2, 4, 8 bayt və daha artıq ola bilər. Başqa sözlə 16, 32, 64 bit v.s. Prosessorun reqistrlərinin sayı arxitekturadan asılı olaraq müxtəlif ola bilər. **İntel i386** arxitekturalı prosessorların təqribən 17-yə yaxın reqistri olur. Prosessor bu reqistrləri müxtəlif məqsədlər üçün istifadə edir. İstifadə sahələrinə görə reqistrlər aşağıdakı qruplara bölünür: ümumi reqistrlər, yaddaş reqistrləri, idarə reqistrləri, sazlama reqistrləri , bayraqlar v.s.
Biz bu dərslikdə əsasən prosessorun ümumi reqistrlərini örgənəcəyik.
Bundan sonra prosessor dedikdə İntel 80386 arxitekturalı prosessorlar nəzərdə tutulur.

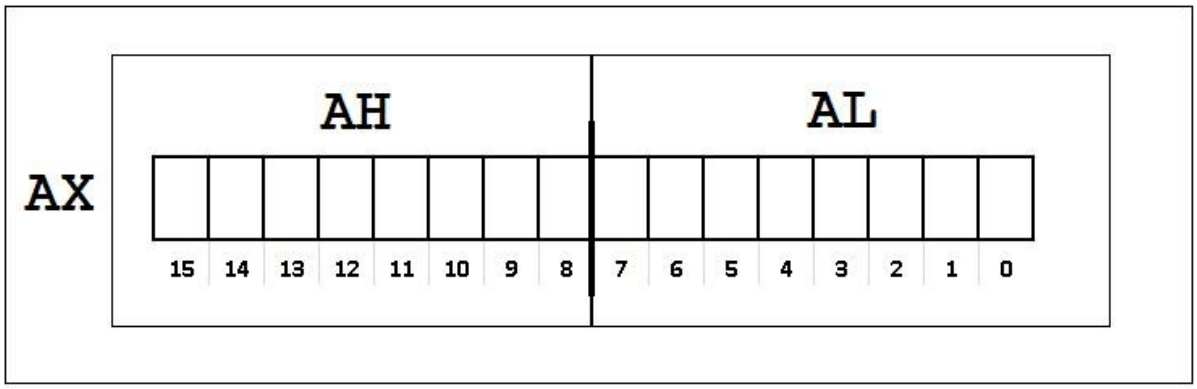
Prosessorun ümumi reqistrləri aşağıdakılardır: **EAX, EBX, ECX, EDX, EBP, ESP, ESİ və EDİ**. Bu reqistrlərin hər birinin ölçüsü 32 bitdir (4 bayt)



Bu reqistrlərinin hər birinin ilk 16 bitinə ayrıca uyğun olaraq **AX, BX, CX, DX, BP, SP, Si** və **Di** adları ilə müraciət etmək olar. 16 bitlik məlumatlarla işləyən zaman bu imkan çox əlverişlidir.



Bundan əlavə **AX, BX, CX, DX** reqistrlərinin özləri də 8 bit - 8 bit olmaq üzərə 2 hissəyə bölünürlər. Bu hissələr uyğun olaraq **AH, BH, CH, DH** (15 - 8 bit) və AL, BL, CL DL (7-0 bit) -dir.



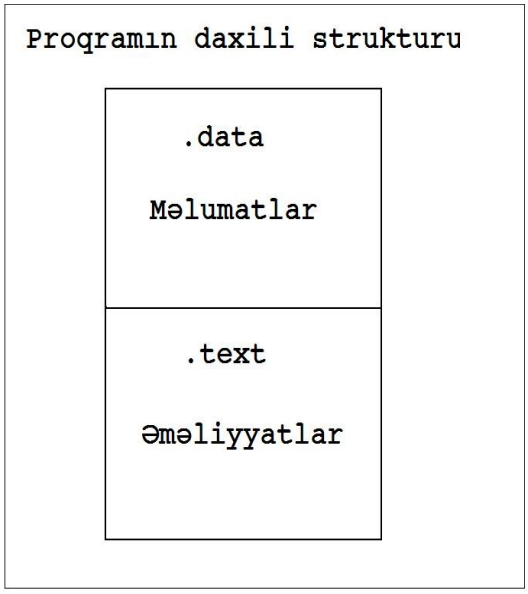
Proqramın daxili strukturu

Müasir icraolunabilən faylların (proqramların) daxili strukturu müxtəlif hissələrdən ibarətdir. Bu hissələrin hər birinin təyinatı və iş prinsipinin örgənilməsi sistem proqramlaşdırma mövzusudur. Bu barədə ətraflı məlumat almaq istəyənlər aşağıdakı sənəddən faydalana bilərlər:

Tool Interface Standards (TIS) Portable Formats Specification, Version 1.1 Executable and Linkable Format (ELF)

Bu sənəddə **elf** tipli linklənəbilən və icraolunabilən faylların (obyekt faylların və proqramların) daxili strukturu tam detalları ilə izah olunur.

Biz isə bir qədər sadələşdirilmiş şəkildə proqramın strukturu ilə tanış olacayıq. Strukturunun mürəkkəbliyindən asılı olmayaraq hər bir proqram özündə iki əsas hissəni saxlamalıdır : məlumatlar və instruksiyalar. Məlumatlar proqramın icrası zamanı elan olunan və istifadə olunan məlumatlardır, İnstruksiyalar isə proqramın bu məlumatlar üzərində yerinə yetirdiyi əməliyyatlardır. Bunları nəzərə alsaq icraolunabilən proqramın sadə strukturunu aşağıdakı kimi verə bilərik:



Proqramın məlumat hissəsi **.data** , instruksiyalar hissəsi isə **.text** kimi işarə olunur. Qeyd edim ki, yüksək səviyyəli dillərdə biz, məlumatları və instruksiyaları proqram mətnində qarışıq istifadə edirik və proqramın mətnini məlumatlar və əməliyyatlar kimi müxtəlif hissələrə ayırmırıq. Bu işlər proqramın kompilyasiyası zamanı kompilyator tərəfindən həyata keçirilir.
Assembler dilində isə görəcəyimiz kimi məlumatlar və instruksiyaları proqramın müvafiq hissələrində yerləşdirməliyik.

İlk proqram

Beləliklə biz prosessorun və proqramın strukturu ilə müəyyən dərəcədə tanış olduq, artıq assembler dilində ilk proqram nümunəsini daxil edə bilərik.

```
#prog1.s

.text
.globl _start

_start:

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

Proqramın izahı

Proqramın ilk sətri **#prog1.s** sətridir. Assembler proqramlarında # ilə başlayan sətirlər şərhləri bildirir.
Növbəti sətir **.text** sətridir. Bu ifadədə proqramın instruksiyalar hissənin başlanğıcını elan edirik.
Proqramımızda növbəti sətir .**globl** _**start** sətridir. Ümumiyətlə . ilə başlayan ifadələr yığma proqramı üçün nəzərdə tutulub(proqramın assembler mətn kodunu obyekt koda çevirən proqrama yığma proqramı deyilir).
.globl ifadəsi yığma proqramına _**start** nişanının qlobal nişan olduğunu bildirir. Bu barədə hələlik dərinə getməyək.
Proqramın növbəti sətri _**start:** sətridir. Assembler dilində qoşanöqtə (:) ilə bitən ifadələr nişan adlandırılır və hansısa məlumat və ya kod hissəsinin başlanğıcına

işarə edirlər.
Nişanları **.text** və **.data** hissələrində istədiyimiz kimi təyin edə bilərik. Bəzi nişanlar isə, o cümlədən **_start** nişanı əvvəlcədən təyin olunmuş xüsusi əhəmiyyətli nişanlardır.
_start nişanı proqramın icraya başladığı yeri bildirir.
Növbəti sətir **movl $1, %eax** instruksiyasıdır. Bu instruksiya birbaşa prosessor tərəfindən icra olunur və bu zaman prosessorun **%eax** reqistrinə 1 qiyməti yazılır. Assembler dilində (AT&T sintaksisi) reqistr adlarını dəyişən və digər adlardan fərqləndirmək üçün reqistr adlarının əvvəlinə faiz(**%**) işarəsi artırılır. Ədədləri isə yaddaş ünvanlarından fərqləndirmək üçün **$** işarəsindən istifadə edirlər. Bu barədə ətraflı irəlidə yaddaşa müraciət metodlarını örgəndikdə tanış olacayıq.
Proqramın sonuncu instruksiyası isə **int $0x80** instruksiyasıdır.
int instruksiyası kəsilmə instruksiyasıdır. İstifadəçi proqramları əməliyyatlar sistemindən hər hansı xidmətin yerinə yetirilməsini sifariş vermək üçün kəsilmədən istifadə edirlər. Bu xidmətlərə daimi yaddaşa, şəbəkəyə və kompüterin digər resurslarına müraciət daxildir. Baxdığımız halda biz sistemdən proqramı söndürməyi xahiş edirik.
Bütün bu deyilənlərdən belə aydın olur ki, baxdığımız proqram icraya başlayır,prosessorun **%eax** reqistrinə 1 qiyməti yazır və icrasın başa çatdırır.

Assembler dilində proqramların icrası
Yuxarıda verilmiş proqramı icra edək. Bunun üçün proqram kodunu **proq1.s** adlı mətn faylında yadda saxlayırıq. **proq1.s** -dən icra oluna bilən proqram almaq üçün onu yığmalıyıq. Terminaldan **prog1.s** faylı yerləşən qovluğa daxil oluruq və aşağıdakı əmrləri daxil edirik:
as prog1.s -o prog1.o
ld prog1.o -o prog1
Nəticədə işci qovluqda **proq1** adlı yeni proqram faylı yaranacaq. Bu proqramı icra eləmək üçün **./prog1** əmrini daxil edirik.
Aşağıdakı kimi:
```
[user@unix progs]$
[user@unix progs]$ as prog1.s -o prog1.o
[user@unix progs]$ ld prog1.o -o prog1
[user@unix progs]$ ./prog1
[user@unix progs]$
```

Qeyd: Kitabda daxil etdiyimiz bütün proqramlar bu üsulla yığılıb yerinə yetirilir. Proqramı icra etdikdə, gördüyümüz kimi heç bir nəticə almırıq, proqramı yükləyən kimi o dərhal başa çatır. Ancaq gördüyümüz işin doğruluğunu yoxlamaq üçün nəticəni test edə bilmək mütləq vacibdir. Ən azından ekranda hansısa məlumat çap edə bilsək nəyisə yoxlaya bilərik. Amma təəssüf ki, bunu etmək üçün əlimizdə yüksək səviyyəli dillərdə olduğu kimi **printf, std::cout, Console.writeln** v.s. kimi funksiyalar yoxdur. Buna görə assemblerdə çap etməni örgənənə kimi bir müddət primitiv üsullarla nəticələri yoxlamalı olacağıq.
Misal üçün əgər proqramın sonunda **%ebx** reqistrinə hansısa məlumat yazsaq, proqramı icra etdikdən sonra bu məlumatı **echo $?** əmri ilə örgənə bilərik.
Gəlin assembler dilində ikinci proqramımıza nəzər salaq.

```
#proq2.s
```

```
.text
.globl _start

_start:

movl $4, %ebx

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

Bu proqramı yığıb, icra etsək və dərhal sonra **echo $?** əmrini daxil etsək onda ekranda 4 qiyməti çap olunar.

Nəticə:

```
[user@unix progs]$
[user@unix progs]$ as proq2.s -o proq2.o
[user@unix progs]$ ld proq2.o -o proq2
[user@unix progs]$ ./proq2
[user@unix progs]$ echo $?
4
[user@unix progs]$
[user@unix progs]$
```

Bu başlıqda biz prosessorun məlumat reqistrləri, proqramın əsas struktur vahidləri, assembler dilində proqram yazmaq, yığmaq və test etmək qaydaları ilə tanış olduq. Növbəti başlıqda biz **CPU** -nun bəzi ümumi instruksiyaları ilə tanış olacayıq.

Tapşırıq:

1. Hər dəfə **%ebx** reqistrinə 0-la 255 arasında olan müxtəlif qiymətlər verməklə proq2.s proqramını bir neçə dəfə təkrar yerinə yetirib nəticəni yoxlayın.

$2 Sadə instruksiyalar

mov instruksiyası

Prosessorun məlumat üzərində yerinə yetirdiyi ən əsas işlərdən biri məlumatı bir yerdən başqa yerə köçürməkdir. Bunun üçün proqramlarda **mov** instruksiyasından istifadə olunur.

Mov instruksiyası iki arqument qəbul edir, **mov arq1, arq2**. Arq1 köçürülən məlumatı, arq2 isə məlumatın köçürüldüyü yeri bildirir.

Arqument olaraq mov instruksiyasına hər hansı ədəd, reqistr və ya yaddaş ünvanı verə bilərik. Misal üçün aşağıdakı instruksiya ilə prosessorun **%ecx** reqistrində olan məlumat **%eax** reqistrinə köçürülür:

movl %ecx, %eax

Mov instruksiyasının imkanları çox genişdir,belə ki, bu instruksiya vastəsilə məlumatı reqistrlərdən fiziki yaddaşa, fiziki yaddaşdan reqistrlərə, bir reqistrdən digərinə v.s. köçürmək olur. Bunlar ilə irəlidə kompüterin fiziki yaddaşının strukturunu örgəndikdən sonra məşğul olacayıq. Hələlik isə ancaq reqistrlərlə işləyək.

Başqa bir nümunəyə baxaq:

movl $5, %ebx

Bu instruksiya isə prosessorun **%ebx** reqistrinə 5 qiyməti yazır.

add instruksiyası

Add instruksiyası iki məlumatı cəmləmək üçün istifadə olunur.

add arq1, arq2

Bu zaman **arq1** ilə **arq2** -nin cəmi hesablanır və nəticə **arq2** -nin üzərinə yazılır. Add instruksiyasına aid nümunələrə baxaq:

```
add %eax, %ebx
add $56, %ecx
add dey1, %edx
```

Birinci instruksiya **%eax** reqistrində olan məlumatı **%ebx** -dəki ilə cəmləyir və nəticəni **%ebx** -ə yazır, ikinci instruksiya **%ecx** reqistrinin qiymətin 56 vahid artırır, üçüncü instruksiya isə fiziki yaddaşda yerləşən **dey1** dəyişəninin qiymətini **%edx** -in qiməti ilə cəmləyir və nəticəni **%edx** -ə yerləşdirir.

Assembler dilində sadə cəmləmə proqramına baxaq:

```
#proq3.s

.text
.globl _start

_start:

movl $23, %ecx
movl $5, %exb

add  %ecx, %ebx

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

Proqramı icra edək:

```
[user@unix progs]$
[user@unix progs]$ as proq3.s -o proq3.o
[user@unix progs]$ ld proq3.o -o proq3
[user@unix progs]$ ./proq3
[user@unix progs]$ echo $?
28
[user@unix progs]$
```

Proqramın izahı:

Proqramın ilk sətirlərinin izahını bilirik. **movl $23, %ecx** instruksiyası **%ecx** -ə 23 qiymətini yazır. **movl $5, %exb** instruksiyası isə **%ebx** -ə 5 qiymətini yazır. add **%ecx**, **%ebx** instruksiyası **%ecx** -dəki məlumatı **%ebx** -in üzərinə əlavə edib, nəticəni **%ebx** -də saxlayır.

sub instruksiyası

sub - çıxma instruksiyasıdır. Bu instruksiya **add** instruksiyası ilə eyni qəbildən olan instruksiyadır, sadəcə **sub** birinci arqumentin qiymətini ikincidən çıxıb nəticəni ikinciyə yerləşdirir, misal üçün:

subl $5, %eax

instruksiyası **eax** reqistrinin qiymətini 5 vahid azaldar.

mull instruksiyası

mull - vurma instruksiyasıdır. O da **add** instruksiyasına analojidir, belə ki, iki arqument qəbul edir, **mull arq1, arq2**. Birinci ilə ikinci arqumentin hasilini hesablayıb ikinciyə yerləşdirir.

div instruksiyası

div - bölmə instruksiyasıdır. Sintaksisi divl arq şəklindədir. Bu zaman div instruksiyası **%edx:%eax** -də yerləşən qiyməti arq -a bölür. Arqument olaraq reqistr və ya nişan göstərilə bilər. Qismət **%eax**, qalıq isə **%edx** instruksiyasına yerləşdirilir. Misal üçün 523 ədədini 8 -ə bölək.

```
movl $0, %edx
movl $523, %eax
movl $8, %ecx
divl %ecx
```

Bölmə həyata keçirilmişdir, **divl** instruksiyasından sonra **%eax** -də qismət, **%edx** -də isə qalıq yerləşir.

$3 Fiziki Yaddaşın Strukturu

Fiziki Yaddaş hər-birinin ölçüsü eyni olan və ardıcıl yerləşdirilmiş kiçik yaddaş hissələrindən – **BAYTLARDAN** ibarət bir sahədir.
1 bayt ən kiçik yaddaş hesab olunur və bu ölçüdə yaddaşda 0-dan 255 -ə (255 də daxil olmaqla) kimi ədəd yerləşdirmək olar(natural ədədlər).
Onu da nəzərinizə çatdırım ki, fiziki yaddaşda ədəddən başqa heçnə yerləşdirə bilmərik. Yəni bizim musiqi, şəkil, mətn, proqram faylları v.s. kimi adət etdiyimiz şeylər kompüterin yaddaşında ədədlər ardıcıllığından başqa bir şey deyillər.
Biz qeyd elədik ki, yaddaş ardıcıl düzülmüş və hər-birinin ölçüsü eyni olan kiçik yaddaş hissələrindən – baytlardan ibarətdir. Kompüter yaddaşı ilə bağlı digər mühüm məsələ **ÜNVAN** məsələsidir. Fiziki yaddaşı təşkil edən hər-bir baytın öz ünvanı olur. Fiziki yaddaşı təşkil edən baytlar ardıcıl düzülür,və 0-dan başlayaraq nömrələnir.
Yəni yaddaşın ilk baytının nömrəsi 0, növbəti baytın nömrəsi 1, v.s.
TƏRİF: Fiziki yaddaşda verilmiş baytın sıra nömrəsi onun ÜNVANI adlanır.
Yeri gəlmişkən qeyd edim ki, Fiziki yaddaş bizim bildiyimiz RAM qurğusudur.
Tapşırıq:
1. Həcmi 512 MB, 1GB, 2 GB olan fiziki yaddaşın sonuncu baytının ünvanını hesablayın.
2. Yuxarıdakı məsələdə aldığınız ünvanları 16-lıq (hex) say sistemi ilə ifadə edin.
3. Sonuncu baytının ünvanı **0xffffffff** olan fiziki yaddaşın həcmini hesabalayın.
Fiziki yaddaşdan əlavə kompüterin daimi yaddaş qurğuları da mövcuddur. Bunlara **Sərt disk**, **CD/DVD**, **USB** disk v.s. misal gətirmək olar. Daimi yaddaş məlumatın uzun müddət saxlanması üçün istifadə olunur. Proqramı icra edərkən əməliyyatlar sistemi proqramın obyekt faylını daimi yaddaşdan fiziki yaddaşa köçürür. Fiziki yaddaşda isə uzun müddət saxlanılan məlumatlar yox, icra olunan proqramlar yüklənir. Fiziki yaddaş məhs bu məqsədlə hazırlanıb və onunla **CPU** arasında məlumat mübadiləsi çox sürətlidir. Daimi yaddaşa isə **CPU** birbaşa müraciət etmir. Bundan sonra yaddaş dedikdə yalnız fiziki yaddaş qurğusu başa düşülür.

Prosessorun Məlumata Müraciətetmə Üsulları

Prosessorun əldə etdiyi məlumat birbaşa instruksiyanın daxilində, fiziki yaddaşda və ya reqistrdə ola bilər. Prosessor məlumata 2 üsulla müraciət edə bilər: birbaşa və fiziki yaddaşdan. Birbaşa müraciət zamanı məlumat ya instruksiyanın daxilində, ya da reqistrdə yerləşir.
Müasir prosessorlar fiziki yaddaşda yerləşmiş məlumata **mov** – köçürmə instruksiyasından istifadə etməklə müxtəlif yollarla müraciət edə bilər. Biz aşağıda bu qaydalar və onların assembler dilində istifadəsi ilə tanış olacayıq. Məlumatın köçürülməsi üçün prosessorun istifadə elədiyi işci instruksiyası mov instruksiyasıdır. mov instruksiyasının sintaksisi ilə biz artıq tanışıq:
mov arq1, arq2
Bu zaman **arq1** məlumatı **arq2** -yə köçürülür. İndi cavab axtırdığımız sual **mov** instruksiyasına **arq1** və **arq2** olaraq hansı arqumentləri verə bilərik sualıdır. Burada bir neçə müxtəlif məqam var ki, ilk dəfəyə başa düşməyə çətinlik yaradır. Məqsəd məlumatı köçürməkdir. **arq1** köçürülən məlumatı, **arq2** isə bu məlumatın köçürüldüyü yeri bildirir. mov instruksiyası 4 müxtəlif yolla məlumatı köçürə bilər: konkret verilən məlumat, konkret fiziki ünvanı verilən məlumat, reqistrdə yerəşən məlumat, fiziki ünvanı reqistrdə yerləşən məlumat.
Mənsəb isə, yəni məlumatın köçürüldüyü yer 2 -dir: fiziki yaddaş və ya **CPU** -nun

reqistri. Birinci arqument olaraq hər-hansı ədəd, nişan və ya reqistr verilə bilər. Misal üçün aşağıdakı kimi.

```
movl 45, %eax
movl %ecx, %edx
movl dey, %ebx
movl 67, melumat
```

Assembler dilində nişanlar proqramda hər-hansı məlumat və ya kod ünvanını bildirmək üçün istifadə olunur. Bundan əlavə arqumentlər bəzən mötərizə arasında da göstərilə bilər. Məsələn:

```
movl (%ecx), %edx
movl (67), melumat
```

Əlavə olaraq arqumentlərin əvvəlinə **$** işarəsi də artıtıla bilər. Misal üçün:

```
movl $45, %eax
movl $dey, %ebx
```

Bütün bunların hamısının əlbəttdə öz mənası var, gəlin onlarla tanış olaq.

1. Arqument olaraq ədəd verilir.

Əgər arqument olaraq ədəd verilirsə bu Fiziki Ünvan bildirir. Misal üçün movl 45, **%ebx** instruksiyasında birinci arqument olaraq 45 ədədi göstərilmişdir. Bu Fiziki yaddaşın 45 nömrəli baytını bildirir. Bu instruksiya icra olunanda yaddaşın 45-ci baytından etibarən növbəti 4 bayt (45-48 -ci baytlar) **%ebx** registrinə köçürülür.

2. Arqument olaraq ədəd verilir və qarşısında $ işarəsi qoyulur.

Əgər arqumentin qarşısında $ işarəsi yerləşirsə, bu zaman artıq fiziki ünvan yox, sadəcə ədəd başa düşülməlidir. Misal üçün **movl $45, %eax** instruksiyası **%eax** reqistrinə 45 ədədini yerləşdirəcək. Bunun Fiziki yaddaşın 45-ci baytında yerləşən məlumatla heç bir əlaqəsi yoxdur.

3. Arqument olaraq nişan göstərilir.

Nişanlar qeyd elədik ki, hər hansı kod və ya məlumatın fikizi yaddaşda yerləşdiyi yeri bildirir. Bu barədə ətraflı dəyişənləri keçdikdə tanış olacayıq. Hələlik isə sadəcə nəzərə alaq ki, nişanlar hər-hansı məlumat və ya kod hissəsinin fiziki ünvanının adla əvəzolunmuş formasıdır. Əgər mov instruksiyasına arqument olaraq nişan verilibsə bu zaman həmin nişanın fiziki yaddaşda istinad elədiyi məlumat nəzərdə tutulur. Misal üçün movl dey, **%ecx** instruksiyası dey - ünvanında yerləşən 4 bayt məlumatı **%ecx** reqistrinə köçürür.

4. Arqument olaraq nişan verilir və qarşısına $ işarəsi qoyulur.

Əgər arqument olaraq nişan verilirsə və qarşısına **$** qoyulursa bu zaman həmin nişanın bildirdiyi fiziki ünvan başa düşülür. Misal üçün **movl $dey, %edx** instruksiyası dey nişanının istinad elədiyi fiziki ünvanı (baytın nömrəsi) **%edx** reqistrinə yazır.

5. Arqument olaraq reqistr verilir.

Əgər arqument olaraq reqistr verilirsə bu zaman həmin reqistrdə yerləşmiş məlumat nəzərdə tutulur. Misal üçün **movl %edx, dey** instruksiyası **%edx** reqistrində olan məlumatı dey dəyişəninə yazır.

6. Arqument olaraq reqistr verilir və arqument mötərəzə işarəsi içinə yerləşdirilir.

Əgər arqument mötərizə işarəsi içindədirsə bu zaman fiziki ünvan başa düşülür. Misal üçün **movl (%edx), %eax** instruksiyası artıq **%edx** reqistrində olan məlumat deyil, fiziki ünvanı **%edx** -dəki ədədə bərabər olan məlumatı **%eax** reqistrinə köçürür. Bu zaman **%edx** -də olan ədəd köçürüləcək məlumatın fizki ünvanını bildirir.

7. Arqument olaraq ədəd, nişan, reqistr və mötərizədən istifadə olunur.

Misal üçün **movl dey(%ecx, %edx, 4), %ebx** . Bu zaman **dey(%ecx, %edx, 4)** ünvanında yerləşən məlumat **%ebx** reqistrinə köçürülür. Bu halda mənbə ünvanın hesablanması üçün aşağıdakı qaydadan istifadə edirlər:

Yekun ünvanın hesablanması qaydası

Başlanğıc_Ünvan (%sürüşmə_reqistri, %index_reiqstri, əmsal) şəklində verilmiş ünavanı yekun ünvana çüvirmək üçün aşağıdakı düsturdan istifadə olunur:

Yekun_ünvan = Başlanğıc_ünvan + %sürüşmə_reqistri + %index_reqistri * əmsal;

Burada **Başlanğıc_ünvan** və **əmsal** olaraq nişan və ya ədəd, **%sürüşmə_reqistri** və **%index_reqistri** olaraq isə reqistrlərdən istifadə olunur.

misal üçün:

movl dey(%eax, %ecx, 4), %edx

Bu zaman əvvəl yuxarıdakı düsturla müraciət olunan yaddaş ünvanı hesablanır:

dey + %eax + %ecx*4;

Sonra fiziki yaddaşda bu ünvanda yerləşən məlumat **%edx**-ə köçürülür. Bu sintaksisdə bütün parametrlərin olması vacib deyil. Əgər hər hansı parametr verilməyibsə onun qiyməti 0 qəbul olunur.

misal üçün: **dey(%eax)** nümunəsində **%index** və əmsal həddlərinin, **dey(,%ecx, 5)** nümunəsində **%sürüşmə** həddinin, **(%eax)** nümunəsində **başlanğıc_ünvan**, **%index** və **əmsal** həddlərinin qiymətləri 0-ra bərabər qəbul olunur.

$4 Dəyişənlər.

Cərgələrin elan olunması .

Əvvəlki Dəyişənlər proqramın **.data** hissəsində elan olunur. Dəyişənlərdən hər - hansı məlumat saxlamaq üçün istiad olunur. Assembler dilində dəyişən elan elan etmək üçün aşağıdakı sintaksisdən istifadə olunur.

```
nişan:
.tip  ilkin_qiymət
```

nişan dəyişənin adını bildirir. tip isə dəyişənin yaddaşda neçə bayt yer tutduğunu göstərir. Qeyd edim ki, assembler dilində yüksək səviyyəli dillərdə olduğu kimi tam tipi, həqiqi tipi, v.s. tiplər xarakteristik deyil. Tip deyərkən əsasən yaddaşda tutulan yerin ölçüsü başa düşülür. Ən geniş istifadə olunan tiplər aşağıdakılardır: **byte**, **int**, **long** və **ascii**. **byte** və **ascii** tiplər bir bayt, **int** və **long** isə uyğun olaraq 2 və 4 bayt qədər yer tutur. **ascii** tipindən Simvol tipli məlumatları yerləşdirmək üçün istifadə olunur.

Misal üçün **dey1** və **dey2** adlı iki dəyişən elan etmək istəsək, aşağıdakı kimi yazmalıyıq

```
dey1:
.long

dey2:
.long
```

Bu zaman yaddaşda **dey1** və **dey2** adlı hər biri 4 bayt yer tutan iki dəyişən elan etmiş oluruq.

Əgər elan zamanı dəyişənlərə ilkin qiymət mənimsətmək istəsək onda bu qiyməti tipdən sonra qeyd etməliyik, aşağıdakı kimi

```
dey1:
.long 34
```

Bu zaman artıq **dey1** dəyişənin ilkin qiyməti 34 olar.

Cərgələrin elan olunması

Yüksək səviyyəli dillərdə olduğu kimi, assembler dilində də cərgələrdən (massivlər) istifadə etmək mümkündür. Bunun üçün adi dəyişənlər elan etdiyimiz qaydadan istifadə olunur, sadəcə olaraq bu zaman cərgənin elementləri qeyd edilərək vergüllə ayrılır. Misal üçün **byte** tipindən olan 5 elementli f cərgəsi elan etmək istəsək onda aşağıdakı kimi yazmaq lazımdır

```
f:
.byte 1,3,45,6,7
```

Bu zaman biz cərgənin elementlərinə ilkin qiymətlər mənimsətmiş olduq.

Sətirlərin elan olunması

Assmebler dilində sətir elan etmək istəsək onda yenə də adi dəyişən elan etmə yolundan istifadə edəcəyik və tip olaraq **ascii** seçəcəyik. İlkin qiymət olaraq isə cütdırnaq arasında elan etmək istədiyimiz sətri yerləşdirəcəyik, misal üçün aşağıdakı kimi

```
set:
.ascii "Salam dunya"
```

Bu qayda ilə biz Salam dunya sətrini elan etmiş oluruq.

Dəyişənlərin qiymətlərinə müraciət.
Gəlin assembler dilində dəyişənlərə aid bir neçə proqram nümunəsinə baxaq. **proq3.s** -də biz iki ədədin cəmini hesabladıq. Ədədlərin qiymətlərini reqistrlərdə yerləşdirdik. İndi artıq biz fiziki yaddaşa müraciət edə bilərik. Gəlin eyni proqramı dəyişənlərdən istifadə etməklə tərtib edək.
İki ədədin cəmi proqramı, proq4.s

```
#pro14.s

.data

eded1:
.long 45

eded2:
.long 34

.text
.globl _start
.type _start,@function

_start:

movl eded1, %eax
movl eded2, %ebx

addl %eax, %ebx

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

Proqramı kompilyasiya və icra edək:

```
[user@unix progs_as]$
[user@unix progs_as]$ as proq4.s -o proq4.o
[user@unix progs_as]$ ld proq4.o -o proq4
[user@unix progs_as]$ ./proq4
[user@unix progs_as]$ echo $?
79
[user@unix progs_as]$
[user@unix progs_as]$
```

jmp instruksiyası
jmp instruksiyası (hoppanmaq) proqramın icra istiqamətini bir yerdən başqa yerə yönləndirmək üçün istifadə olunur. Əsasən **cmp** instruksiyası ilə birlikdə istifadə olunur. Yəni müqaisənin nəticəsindən asılı olaraq proqramın icrasını bu və ya digər yerdən davam etdirmək.
jmp instruksiyasının sintaksisi aşağıdakı kimidir:
jmp mövqe
Bu zaman icraolunma artıq "**mövqe**" adlı yerə sıçrayır.

Sadə proqrama baxaq.

```
    # proq6.s

     .text
     .globl _start
     .type _start,@function

_start:

     movl $1, %ecx
     movl $5, %edx
     addl %ecx, %edx

     movl $6, %ebx

     addl %edx, %ebx

son:
     movl $1, %eax
     int $0x80
```

Bu proqramda biz **_start** -dan əlavə son nişanını tərtib etmişik. Nişanları yuxarıda qeyd elədiyimiz kimi proqramın məlumat və kod hissəsinin istənilən yerində elan edə bilərik. Proqram **_start** -dan başlayaraq aşağı doğru yerləşmiş instruksiyaları ardıcıl icra edir. Proqramı icra eləsək nəticə olaraq 12 alarıq. İndi isə proqramda **jmp** instruksiyasından istifadə edək.

```
    # proq6.s

     .text
     .globl _start
     .type _start,@function

_start:

     movl $1, %ecx
     movl $5, %edx
     addl %ecx, %edx

     movl $6, %ebx
     jmp son

     addl %edx, %ebx

son:
     movl $1, %eax
     int $0x80
```

Əgər bu proqramı icra eləsək onda cavab olaraq 6 qiymətini alarıq. Səbəb isə odur ki, **movl $6, %ebx** instruksiyasından sonra **jmp son** instruksiyası icra olunur. Nəticədə **addl %edx, %ebx** instruksiyası icra olunmadan **son** nişanına keçid baş verir.

cmp instruksiyası

Məlumatların qiymətlərini müqaisə etmək üçün **CPU cmp** instruksiyasından istifadə edir. **cmp** instruksiyasının sintaksisi aşağıdakı kimidir:

cmp arq1, arq2

arq1 ilə **arq2** -in qiyməti yoxlanılır və nəticə **flags** reqistrində qeydə alınır. Müqaisənin nəticəsindən asılı olaraq bu və ya digər əməliyyatı icra etmək üçün **cmp** instruksiyasından sonra şərti keçid instruksiyalarından istifadə etməliyik.

Şərti keçid instruksiyaları aşağıdakılardır:

jg, jge, jl, jle, je, jne.

(jump great, jump great equal, jump less, jump less equal, jump equal, jump not equal)

Bu keçid instruksiyaları iki kəmiyyətin müqaisəsinin bütün mümkün nəticələrini uyğun olaraq aşağıdakı kimi nəzərə alır: keç əgər ikinci arqument birincidən böyükdürsə, böyük bərabərdirsə, kiçikdirsə, kiçik bərabərdirsə, bərabərdirsə, fərqlidirsə.

proqram nümunəsi: İki ədədin böyüyünü tapan proqram tərtib edək.

```
#prog7.s


.data

eded1:
.long 110

eded2:
.long 15

.text

.globl _start
.type _start, @function


_start:

#eded1 -i eax -e yaz

movl eded1, %eax
#eded2 -ni ebx -e yaz

movl eded2, %ebx

#ededleri muqaise et, eger ikinci birinciden boyukdurse

# son -a kec, cunki artiq boyuk eded ebx -dedir

cmpl %eax, %ebx
jg son

#birinci boyukdur, onu ebx -e kocur

movl %eax, %ebx

son:
```

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

Proqramın izahı:

Biz əvvəlcə **eded1** və **eded2** -nin qiymətlərini uyğun olaraq **%eax** və **%ebx** reqistrlərinə köçürdük. Daha sonra **cmp** vastəsilə bu qiymətləri müqaisə etdik. Əgər **%ebx %eax** -dən böyük olarsa bu zaman **jg** instruksiyası **son** -a keçəcək, əks halda **(%eax > %ebx)** olarsa proqramın icrası növbəti instruksiyadan davam edir. **movl %eax, %ebx**. Böyük qiymət **%ebx** -ə yazılır. Proqram sona çatır.

Tapşırıq:

1. eded1 və **eded2** -yə (0-255 arası) müxtəlif qiymətlər verməklə proqramı test edib necə işlədiyin yoxlayın.

$5 Dövrlər.

Proqramda bəzən hansısa əməliyyatları müxtəlif şərtdən asılı olaraq bir neçə dəfə təkrarlamaq lazım gəlir. Yüksək səviyyəli dillərdə bunun üçün xüsusi dövr operatorlarından istifadə olunur(**for**, **while** ...). Assembler dilində isə hər-hansı kod parçasın təkrarlamaq üçün cmp və jmp instruksiyalarından istifadə edirlər. Əslində yüksək səviyyəli dillərdə istifadə olunan dövr operatorları da aşağı səviyyədə cmp və jmp instruksiyaları ilə realizə olunur.
Sadə dövrdən istifadə edən proqram nümunəsi ilə tanış olaq. Verilmiş ədədlər ardıcıllığı içərisində ən böyüyünü tapan proqram.
Tutaq ki, bizə 7 ədəd verilmişdir. Bu ədədlər içərisində ən böyüyünü tapan proqram tərtib edək. Bu zaman biz cərgədən (massiv) istifadə edəcəyik. Cərgənin elan olunması ilə yuxarıda tanış olmuşuq.

```
        #prog8.s

        .data

        eded_ard:
        .long 241, 15, 242, 123, 50, 100, 240

        say:
        .long 7

        .text

        .globl _start
        .type _start,@function


        _start:

        movl $0, %ebx
        movl $0, %edx

dovr:
        cmpl say, %edx
        je son

        movl eded_ard(,%edx,4), %eax
        cmpl %eax, %ebx
        jg boyuk
        movl %eax, %ebx
boyuk:
        incl %edx
        jmp dovr

        son:

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

Programın izahı:
Proqramın məlumat hissəsində (**.data**) biz aşağıdakı məlumatları yerləşdiririk.
Əvvəl biz 7 ədəddən ibarət ardıcıllıq elan edirik.

```
eded_ard:
.long 220, 15, 3, 123, 50, 100, 240
```

Daha sonra isə say nişanı.

```
say:
.long 7
```

Say dəyişənində biz ədələrin sayını yerləşdiririk. Bu bizə dövrün bitməsi şərtini yoxlamaq üçün lazımdır.
Daha sonra proqramın instruksiyalar hissəsini elan edirik.
.text
%edx reqistrində biz nəzərdən keçirdiyimiz ədədlərin sayını saxlayırıq. Ona görə ilk başlanğıcda bu reqistrə 0 qiyməti yerləşdiririk. Hələlik heç bir ədədin qiymətini yoxlamamışıq.
movl $0, %edx
%ebx -də isə ədədlər ardıcıllığından nəzərdən keçirdiyimiz ədədlər içərisindən ən böyüyünü yerləşdiririk. Dövr hər dəfə təkrarlandıqca cərgənin növbəti elementinin qiyməti **%eax** reqistrinə köçürülür və onun qiyməti **%ebx** ilə müqaisə olunur. Əgər böyükdürsə həmin qiymət **%ebx** -ə yazılır. Beləliklə **%ebx** -də həmişə baxılan ədədlər içərisində ən böyüyü yerləşir. Başlanğıcda isə **%ebx** -də 0 qiyməti yerləşdirməliyik.
Növbəti sətirdə biz **dovr** nişanını elan edirik.
dovr:
Bu nişan dövrün başlanğıcı hesab olunur. Daha sonra biz say dəyişəni ilə **%edx** -də olan qiyməti müqaisə edirik(**cmpl say, %edx**). Əgər onlar bərabərdirsə deməli bütün ədədlər yoxlanılıb dövrdən çıxırıq(**jmp son**).

```
cmpl say, %edx
je son
```

%edx -də biz baxdığımız ədədlərin sayını saxlayırıq və başlanğıcda ona 0 mənimsətmişik. Dövr hər dəfə təkrarlandıqda biz **%edx** -in qiymətin 1 vahid artırırıq. Növbəti instruksiya hər dəfə dövr təkrar olduqda **eded_ard** cərgəsinin növbəti elementini **%eax** registrinə köçürür.

```
movl eded_ard(,%edx,4), %eax
```

Burada **FİZİKİ ÜNAVININ** hesablanma düsturunu yada salsaq mənbə ünvan aşağıdakı kimi hesablanır:

```
eded_ard + 0 + %edx*4
```

eded_ard məlumatın yaddaşdakı ünavanıdır. Mötərizədən sonrakı birinci həd buraxıldığından onun qiyməti 0 götürülür. Cəmin üzərinə **%edx** -lə 4 - ün hasili əlavə olunur. Beləliklə köçürülməli olan məlumatın yekun ünvanı hesablanır. Dövrün başlanğıcında **%edx** -in qiyməti 0 olduğundan düsturun nəticəsi elə **eded_ard** olacaq. Bu isə cərgənin ilk elementinin ünvanıdır. Beləliklə dövrün başlanğıcında bu instruksiya icra olunduqda cərgənin ilk elementi yəni 220 **%eax** -ə yazılır. Dövr hər dəfə təkrarlandıqda qeyd etdiyimiz kimi **%edx** -in qiyməti 1 vahid artır. Bu isə yuxarıdakı düstura görə ünvanın qiymətini 4 vahid artırır və cərgənin növbəti

elementinin ünvanını almış oluruq. Cərgənin elementlərinin tipini long elan etdiyimizdən onun hər bir elementi yaddaşda 4 bayt yer tutur və cərgənin elementləri yaddaşda ardıcıl yerləşir. Buna görə **k**-cı elementin ünvanını almaq üçün ilk elementin ünvanının üzərinə **(k-1)*4** əlavə etməliyik.
Növbəti instruksiyalar aşağıdakı kimidir:

```
        cmpl %eax, %ebx
        jg boyuk
        movl %eax, %ebx
```

Burada **%ebx** nəzərdən keçirdiyimiz ədərlər içərisində ən böyüyünü, **%eax** isə cərgənin növbəti elementinin qiymətini özündə saxlayır. Əgər **%ebx %eax** -dən böyükdürsə onda **%eax** -də olan qiymət bizim üçün maraqlı deyil və biz **jb boyuk** instruksiyası ilə boyuk nişanına keçid edirik. Harada ki, yeni dovrə keçid işləri üçün hazırlıq işləri görülür və yeni dövrə keçid edilir. Lakin əks halda, yəni **%eax %ebx** -dən böyük olarsa deməli cərgənin hal - hazırda baxılan qiyməti indiyə kimi baxdığımız qiymətlərdən böyükdür və maksimum olaraq onu götürməliyik. Bu halda artıq jg boyuk (**jump great**) instruksiyası icra olunmur və növbəti instruksiya, **movl %eax, %ebx** instruksiyası icra olunur və **%eax** -in qiymətin **%ebx** -ə yazır. Nəticədə **%ebx** -də baxılan ədədlərin içərisindən ən böyüyü yerləşir.
Daha sonra proqram kodu aşağıdakı kimidir:

```
boyuk:
        incl %edx
        jmp dovr
```

Buarada artıq qeyd elədiyimiz kimi, boyuk nişanı elan olunur. **incl %edx** instruksiyası **%edx** -in qiymətini 1 vahid artırır. Daha sonra jmp dovr instruksiyası vastəsilə dövrün başlanğıcına (**dovr** nişanı) keçid edilir.
Hər dəfə dövr təkrarlandıqda **%edx** -in qiyməti 1 vahid artdığından **%edx** -in qiyməti **say** qiymətinə bərabər olduqda **son** nişanına keçid edilir və proqram sona çatır.

incl və decl instruksiyaları

incl , decl instruksiyaları müvafiq olaraq arqumentlərinin qiymətlərini bir vahid artırır, azaldır. Bu instruksiyaların əvəzinə **addl $1, arq** və ya **subl %1, arq** -dan da istifad edə bilərik, lakin bunun üçün hazır instruksiya mövcud olduğundan ondan istifadə edirik. Həm də bu zaman eyni nəticənin alınmasına daha az **CPU** vaxtı sərf olunur.

Salam Dünya proqramı

Artıq biz assembler dilində bir qədər mürəkkəb proqramlar tərtib edə bilərik. İndiyə kimi biz nəticəni yoxlamaq üçün ancaq **%ebx** reqistrinin ilk baytından istifadə edə bilirdik. İndi isə bizə lazım olan məlumatın ekranda çap etməni örgənək. Elə proqram tərtib edək ki, ekranda **%ecx** reqistrinin qiymətini çap etsin.
Qeyd eliyim ki, ilk başlanğıcda bu kifayət qədər çətin məsələ olduğundan biz hələki daha asan məsələ üzərində çalışaq. Ekranda verilmiş simvolu, məsələn **'a'** simvolunu çap edən proqram tərtib edək. Əvvəlcə proqramı daxil edək, daha sonra izahını verərik.

```
#proq9.s
# Ekranda kicik 'a' simvolu cap eden proqram

.data

simvol:
.ascii "a"

.text

.globl _start
.type _start, @function

 _start:

 movl $4, %eax
 movl $1, %ebx
 movl simvol, %ecx
 movl $1, %edx
 int  $0x80

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

Əgər biz bu proqramı icra eləsək onda aşağıdakı nəticəni alarıq:

```
[user@unix progs_as]$
[user@unix progs_as]$ as prog9.s -o prog9.o
[user@unix progs_as]$ ld prog9.o -o prog9
[user@unix progs_as]$ ./prog9
a[user@unix progs_as]$
[user@unix progs_as]$
```

Proqramın nəticəsinin daha aydın seçilməsi üçün biz **'a'** simvolundan sonra yeni sətir - **'\n'** simvolunu da çap etməliyik. Müvafiq proqram aşağıdakı kimi olar:

```
.data

simvol:
.ascii "a\n"

.text

.globl _start
.type _start, @function

 _start:

 movl $4, %eax
 movl $1, %ebx
 movl $simvol, %ecx
 movl $2, %edx
 int  $0x80

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

Nəticə:

```
[user@unix progs_as]$
[user@unix progs_as]$ as prog10.s -o prog10.o
[user@unix progs_as]$ ld prog10.o -o prog10
[user@unix progs_as]$ ./prog10
a
[user@unix progs_as]$
```

proq9.s proqamının izahı(proq10.s analojidir):

Proqramın əvvəlində biz **.data** hissəsində ascii tipindən olan simvol nişanı elan edirik və bu yerə **"a"** məlumatını, ekranda çap etmək istədiyimiz sətri yerləşdiririk. Bizim məqsədimiz verilmiş sətri ekranda çap etməkdir. Bunun üçün **%eax**, **%ebx**, **%ecx** və **%edx** reqistrlərinə lazım olan qiymətləri yerləşdirməli, daha sonra isə int **$0x80** instruksiyasını icra etməliyik. **%eax** və **%ebx** reqistrlərinə müvafiq olaraq 4 və 1 qiymətləri yerləşdirilməlidir. **%ecx** reqistrinə çap olunmalı sətrin yaddaşdakı ünvanı, **%edx** -ə isə həmin ünvandan başlayaraq çap olunmalı simvolların sayı yerləşdirilmədir.

Proqramın **.text** hissəsində ilk iki instruksiyası aşağıdakı kimidir:

```
movl $4, %eax
movl $1, %ebx
```

Bu instruksiyalar **%eax** və **%ebx** registrlərinə müvafiq olarq 4 və 1 qimyətlərini yazırlar. Daha sonra **movl $simvol, %ecx** instruksiyası ilə simvol nişanın ünvanını **%ecx** -ə yerləşdiririk. **movl $1, %edx** isə **%edx** -ə 1 qiymətini yazır. Yəni cəmi 1 simvol çap et. Bütün bunlar hamısı hazırlıq işləri idi, sadəcə prosessorun reqistrlərinə lazımı məlumatlar yerləşdirirdi və bu zaman ekranda heçnə çap olunmur. Real iş isə **int $0x80** instruksiyanı yerinə yetirən zaman baş verir. Bu zaman ekranda istədiyimiz sətir çap olunar. int istuksiyası kəsilmə instruksiyasıdır.

Ekranda tələb olunan simvolu çap etdikdən sonra proqram başa çatır, bunun üçün

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

instruksiyalarını icra edirik.

Başqa bir nümunə, ekranda **"Salam dünya"** sətrini çap edən proqram ətrtib edək. Artıq bu proqramı özünüz sərbəst tərtib edə bilməlisiniz. Proqram aşağıdakı kimi olar:

```
.data

simvol:
.ascii "Salam dunya\n"


.text

.globl _start
.type _start, @function

 _start:

 movl $4, %eax
 movl $1, %ebx
```

```
        movl $simvol, %ecx
        movl $12, %edx
        int  $0x80

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

Nəticə:

```
[user@unix progs_as]$
[user@unix progs_as]$ as prog11.s -o prog11.o
[user@unix progs_as]$ ld prog11.o -o prog11
[user@unix progs_as]$ ./prog11
Salam dunya
[user@unix progs_as]$
```

Ekranda simvol və ya sətir çap etməni örgəndik. İndi bundan daha mürəkkəb olan məsələ ilə, verilmiş ədədin ekranda çap olunması ilə məşğul olaq. Əvvəl proqramı daxil edək, sonra izahı verərik.

```
         #prog12.s
         #melumatlar hissesi

         .data

         #cap etmek istediyimiz eded
         dey:
         .long 907841

         onluq_simvollar:
         .ascii "0123456789"

         say:
         .long 0

         yeni_setir:
         .ascii "\n"

       # kod  hissesi
       .text

       .globl _start
       .type _start, @function

_start:

dovr:
         movl $10, %edi
         movl $0, %edx
         movl dey, %eax
         div  %edi

         pushq %rdx
         movl  %eax, dey

         incl say

         movl $0, %esi
```

```
        cmpl dey, %esi
        je dovr_son

        jmp dovr

dovr_son:

        movl $0, %esi

cap_et:
        cmpl %esi, say
        je   yeni_str_cap

        popq %rdi

        movl $4, %eax
        movl $1, %ebx
        movl $onluq_simvollar, %ecx
        addl %edi, %ecx
        movl $1, %edx
        int  $0x80

        incl %esi
        jmp  cap_et

yeni_str_cap:
        #yeni setir cap et
        movl $4, %eax
        movl $1, %ebx
        movl $yeni_setir, %ecx
        movl $1, %edx
        int  $0x80

son:
        movl $1, %eax
        int  $0x80
```

Proqramı icra edək:

```
[user@unix progs_as]$
[user@unix progs_as]$ as prog12.s -o prog12.o
[user@unix progs_as]$ ld prog12.o -o prog12
[user@unix progs_as]$ ./prog12
907841
[user@unix progs_as]$
```

Proqramın izahı:

Proqramın məlumatlar hissəsində (.**data**) long tipindən dey nişanı elan edirik və buraya çap etmək istədiyimiz ədədi yerləşdiririk - 907841 ədədini. Bu ədədi ekranda çap etmək üçün aşağıdakı qaydadan istifadə edirik. Əvvəlcə verilmiş ədədin rəqəmlərini sonuncudan birinciyə kimi bir-bir tapıb stekə yerləşdiririk. Stekin qısa izahı irəlidə verilir,daha ətraflı izah isə növbəti mövzularda verilir.
Daha sonra stekdən bu ədədləri bir-bir çıxarıb onların uyğun gəldikləri simvolu ekranda çap edirik. Burada əsas məsələlərdən biri verilmiş ədədin təşkil olduğu rəqəmləri tapmaqdır. Bunun üçün müxtəlif üsullar olsa da, biz 10-a bölmə və qalığı hesablama qaydasından istifadə edirik. Misal üçün tutaq ki 497 ədədinin

rəqəmlərini tapmaq istəyirik. Bunun üçün yuxarıdakı qaydanı tətbiq eləsək əvvəlcə 497 -ni 10-a bölək. Qismət 49, qalıq isə 7 alarıq. 7 rəqəmlərdən biridir. Daha sonra eyni qaydanı qismətə tətbiq edirik. 49 -u 10 -a bölsək, qismət 4, qalıq isə 9 alarıq. Beləliklə 2 -ci rəqəmi də tapdıq. Bu prosesi qismətdə 0 alana kimi davam etdirsək verilmiş ədədin bütün rəqəmlərini alarıq. Lakin bu üsulun çatışmayan bir cəhəti ondadır ki, bu üsulla ədədin rəqəmləri əks ardıcıllıqla, əvvəldən axıra alınır. misal üçün yuxarıdakı münunədə 497 ədədi üçün biz 7 9 və 4 kimi nəticə alacayıq. Bu şəkildə biz onu çap edə bilmərik. Bu problemi stekdən istifadə etməklə asanlıqla həll etmək mümkündür. **Stek** yaddaşda müəyyən bir sahədir. Bu sahəyə məlumat yerləşdirmək və stekdə olan məlumatı götürmək üçün pushl və popl instruksiyalarından istifadə olunur. Məlumatlar yerləşmə ardıcıllığının əksi istiqamətində stekdən çıxarılır.

Beləliklə bütün rəqəmləri stekə yerləşdirdikdən sonra onları bir-bir stekdən çıxarıb, uyğun gəldikləri **ascii** somvolun ekranda çap etməliyik. Bunun üçün .data hissəsində **onluq_simvollar** nişanı elan edirik və burada 0 -dan 9 -a kimi ascii simvolları yerləşdiririk. Simvol çap etmə ilə yuxarıda tanış olmuşuq. **onluq_simvollar** nişanı bu **simvollar** sətrinin ilk elementinin, **"0"** simvolunun ünvanını özündə saxlayır. Hər bir **ascii** simvolu yaddaşda bir bayt yer tutduğundan verilmiş rəqəmə uyğun simvolun ünvanın almaq üçün **onluq_simvollar** nişanının üzərinə həmin rəqəmi əlavə etməliyik.

Çalışmalar:

1. Verilmiş reqistrin qiymətini ekranda çap edən proqram tərtib edin.

2. Verilmiş reqistrin qiymətini ekranda 16-lıq(hex) say sistemində çap edən proqram tərtib edin.

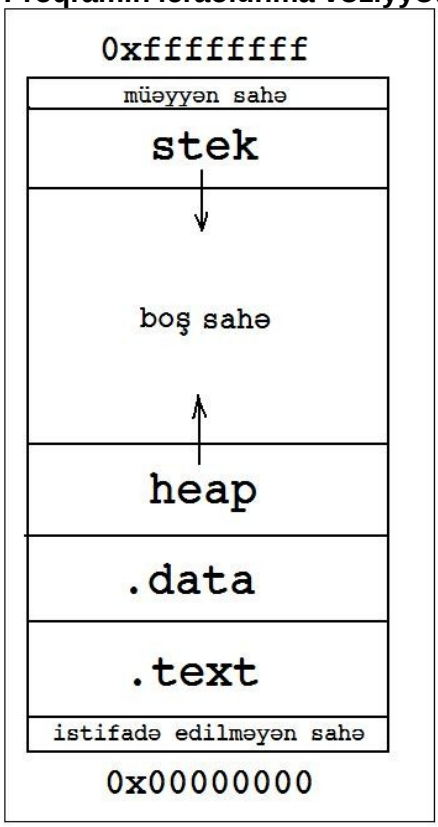
3. Proqramda hər hansı dəyişən elan edin, onun ünvanın ekranda çap edən proqram tərtib edin.

4. _start nişanının ünvanını çap edən proqram tərtib edin.

$6 Stek.

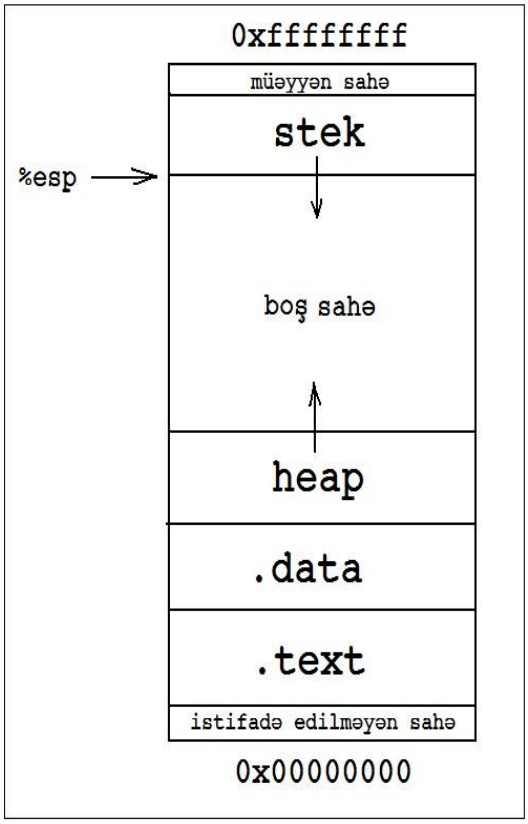
Hər bir yeni proqram işə salındıqda əməliyyatlar sistemi bu proqram üçün yaddaşda müəyyən yer ayırır. Bu yer yalnız həmin proqrama məxsus olub, onun yaddaş sahəsi adlanır. Proqramın yaddaş sahəsi, başqa sözlə icra olunan proqramın yaddaşdakı surəti aşağıdakı struktura malikdir:

Proqramın icraolunma vəziyyətindəki strukturu:



Burada .**text** və **.data** hissələri proqramın uyğun olaraq instruksiyalar və məlumatlar hissələridir(ikili formada). Bu hissələrin ölçüsü proqramın icrası boyu dəyişmir. **Heap** və **stek** sahələrinə proqramın icrası boyu müxtəlif məlumatlar yerləşdirilir. **Heap** sahəsi dinamik dəyişənlər, **stek** isə funksiyalara ötürülən parametrləri yerləşdirmək və digər məqsədlər üçün istifadə olunur. Şəkildən göründüyü kimi stek yuxarıdan aşağı, **heap** isə aşağıdan yuxarıya doğru artır. Əgər bu yaddaş sahələrinə davamlı məlumat yerləşdirsək onlar bir-birlərinin sərhədlərinə toxunar və nəticədə yaddaş xətası baş verər.

Əgər şəkilə diqqət yetirsək görərik ki, **stek** sahəsi başlanğıc sərhəddi yaddaş fəzasının ən yuxarı hissəsində yerləşir. Stekə məlumatlar yerləşdirdikcə stekin sərhəddi yaddaş fəzası boyunca "aşağıya" doğru hərəkət edir. **Stek** sərhəddinin hal-hazırdakı ünvanını **%esp** reqistri vastəsilə təyin edilir.



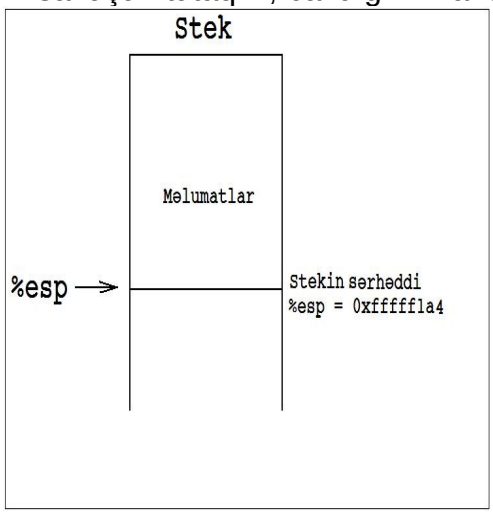
Stekə məlumat yerləşdirmək üçün **push** instruksiyasından istifadə olunur. **push** instruksiyasının sintaksisi aşağıdakı kimidir:

push melumat

Bu zaman məlumat stekə yerləşdirilmiş olur.

Stekə məlumat yerləşdirdikdə və ya götürdükdə **%esp** -in də qiyməti müvafiq olaraq dəyişir və beləliklə də onun həmişə stekin üst hissəsinə istinad etməsi şərti təmin olunmuş olur. Bu işlər prosessor tərəfindən avtomatik yerinə yetirilir.

Misal üçün tutaq ki, baxdığımız anda stekin vəziyyəti aşağıdakı kimidir:

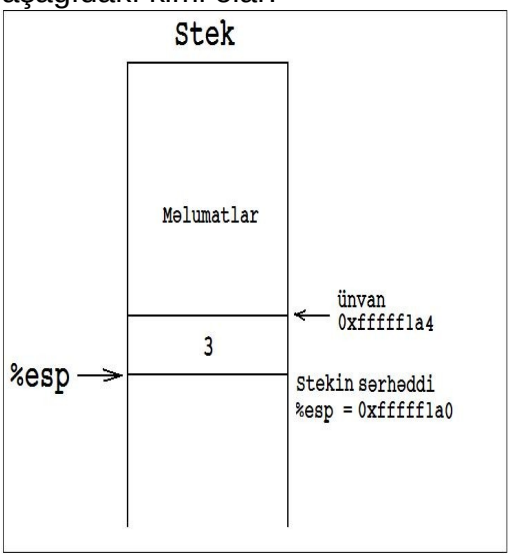


Stekə 3 qiymətini yerləşdirmək üçün aşağıdakı instruksiyanı daxil edirik:

pushl $3

(Qeyd 32 bitlik maşınlarda pushl, 64 bitliklərdə isə **pushq** instruksiyasından istifadə

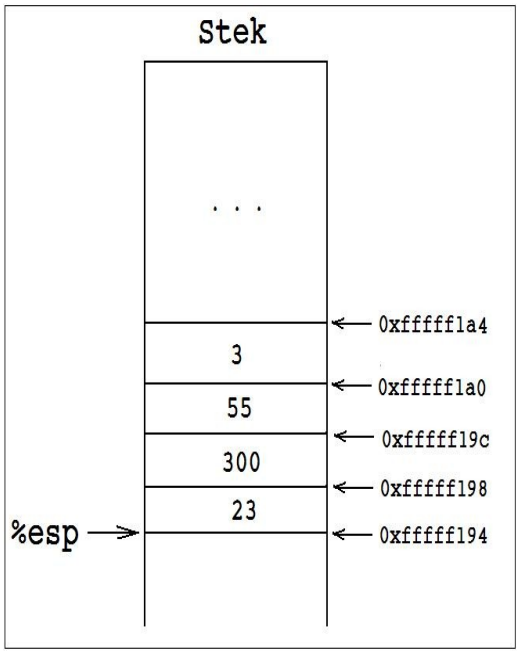
olunur.)
Bu zaman stek göstəricisinin (**%esp**) qiyməti 4 bayt azalar və stekin vəziyyəti aşağıdakı kimi olar:



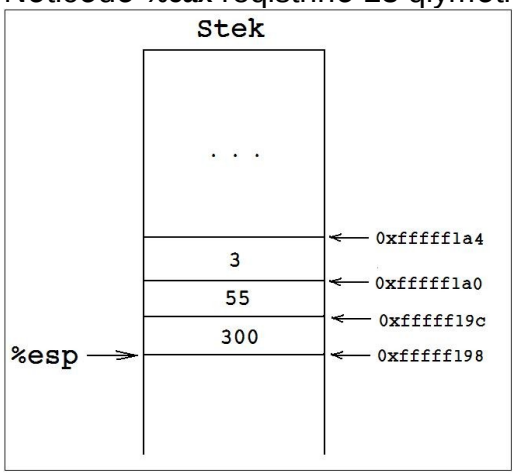
Şəkildən göründüyü kimi, stek göstəricisinin baxdığımız anda qiyməti **0xffff1a4** idi. Stekə 3 qiymətini yerləşdirdikdən sonra stek yaddaşının həcmi artdı və stek göstəricisinin yeni qiyməti **oxffff1a0** oldu, əvvəlki qiymətindən 4 bayt az. Bunu göstərməkdə məqsəd stekə məlumat yerləşdirən zaman onun sahəsinin artamsının və stek göstəricisinin isə azalmasının izahıdır. Bu hissədə anlaşılmazlıq ola bilər, amma stekə məlumat yerləşdirən zaman onun göstəricisinin qiymətinin azalması , başqa sözlə stekin aşağıya doğru artması ideyası sistem arxitektorlarına məxsusdur.
Daha bir neçə məlumat stekə əlavə edək:

```
pushl $55
pushl $300
pushl $23
```

Stekin cari vəziyyəti aşağıdakı kimi olar:



Ünvanlar 16-lıq say sistemində göstərilir. Qeyd edim ki, nümunələr 32 bitlik maşın üçün verilir. Bu zaman stekə məlumat yerləşdirdikdə və ya onda olan məlumatı götürdükdə **stek** göstəricisinin qiyməti uyğun olaraq 4 bayt azalır(artır), 64 bitlik maşınlarda isə 8 bayt.
Stekdən məlumat götürmək üçün isə pop instruksiyasından istifadə edirlər. **pop** instruksiyasının sintaksisi aşağıdakı kimidir:
pop arq
Bu zaman **pop** instruksiyası stekin ən aşağı hissəsində, yəni **stek** göstəricisinin istinad etdiyi yerdə olan 4 bayt (8) məlumatı götürüb arq -a yerləşdirir. **Stek** göstəricisini 4 bayt yuxarı sürüşdürür.
Misal üçün yuxarıdakı nümunədə stekə ən son yerləşdirdiyimiz qiyməti **%eax** reqistrinə köçürmək istəsək, onda aşağıdakı kimi yazmalıyıq:
popl %eax
Nəticədə **%eax** reqistrinə 23 qiyməti yazılar, stekin vəziyyəti isə aşağıdakı kimi olar:



İndi isə proqram nümunələri ilə tanış olaq. Əvvəlki mövzuda biz hər hansı dəyişənin qiymətini ekranda çap edən proqram tərtib etdik. Hər-hansı məlumatın ekranda çap olunması istifadəçi üçün çox mühümdür. Lakin eyni dərəcədə

istifadəçinin də proqrama hər-hansı məlumat daxil edə bilməsinə ehtiyac var. Gəlin indi bu məslənin assembler dilində necə realizə olunması isə məşğul olaq. Əvvəlcə yenə ilk olar sadə hal, hər-hansı sətrin istifadəçi tərəfindən daxil olunması ilə məşğul olcayıq. Daha sonra isə ədədlərin hansı -yolla daxil olunması örgənəcəyik.
Elə proqram tərtib edək ki, istifadəçidən hər-hansı sətir qəbul etsin və ekranda həmin sətirdə olan **'a'** simvollarının sayını çap etsin. Əgər istifadəçinin daxil etdiyi sətirdə **'a'** somvolu yoxdursa onda bu barədə məlumat çap eləsin.

```
#prog14.s
        .data

setir1:
        .ascii "100 simvoldan cox olmayan her hansi setir daxil edin\n"

#setirde olan simvollarin sayi
say1:
        .long 53

setir2:
        .ascii "a simvollarinin sayi -> "

#setirde olan simvollarin sayi
say2:
        .long 24

yeni_setir:
        .ascii "\n"

#neticeni setre ceviremek ucun melumatlar
dey:
        .long 0

onluq_simvollar:
        .ascii "0123456789"

say:
        .long 0

bufer:
        .rept 100
        .ascii "\0"
        .endr

      .text

      .globl _start
      .type _start, @function

_start:
        #setri cap edek
        movl $4, %eax
        movl $1, %ebx
        movl $setir1, %ecx
        movl say1, %edx
        int  $0x80
```

```
        #istifadeciden her-hansi setir qebul edirik(max 20 simvol)
        movl $3, %eax
        movl $0, %ebx
        movl $bufer, %ecx
        movl $100, %edx
        int  $0x80

        #dovre hazirliq
        movl $0, %edi
        movl $0, %esi
        #%ebx ve %eax -de olan melumatlari silirik
        #bunun ucun movl $0, %ebx -de yaza bilerdik
        #amma xorl daha suretle ishleyir

        xorl %ebx, %ebx
        xorl %eax, %eax
        movb $'a', %bl

dovr:
        movb bufer(,%edi,1), %al
        incl %edi
        cmpl $100, %edi
        je dovr1_son
        cmpb %al, %bl
        jne dovr
        incl %esi
        jmp dovr

dovr1_son:
        movl %esi, dey
#setir2 cap_et

        movl $4, %eax
        movl $1, %ebx
        movl $setir2, %ecx
        movl say2, %edx
        int  $0x80

#neticeni ekranda cap etmek ucun setire cevirek
dovr1:
        movl $10, %edi
        movl $0, %edx
        movl dey, %eax
        div  %edi

        pushq %rdx
        movl  %eax, dey

        incl say

        movl $0, %esi
        cmpl dey, %esi
        je dovr_son

        jmp dovr1

dovr_son:
```

```
        movl $0, %esi

cap_et:
        cmpl %esi, say
        je   yeni_str_cap

        popq %rdi

        movl $4, %eax
        movl $1, %ebx
        movl $onluq_simvollar, %ecx
        addl %edi, %ecx
        movl $1, %edx
        int  $0x80

        incl %esi
        jmp  cap_et

yeni_str_cap:
        #yeni setir cap et
        movl $4, %eax
        movl $1, %ebx
        movl $yeni_setir, %ecx
        movl $1, %edx
        int  $0x80
son:
        movl $1, %eax
        int  $0x80
```

Nəticəni yoxlayaq:

```
[user@unix progs_as]$
[user@unix progs_as]$ as prog14.s -o prog14.o
[user@unix progs_as]$ ld prog14.o -o prog14
[user@unix progs_as]$ ./prog14
100 simvoldan cox olmayan her hansi setir daxil edin
kszdfjhlaskja aaa  kdfjnka  ads
a simvollarinin sayi -> 7
[user@unix progs_as]$
[user@unix progs_as]$ ./prog14
100 simvoldan cox olmayan her hansi setir daxil edin
aaa
a simvollarinin sayi -> 3
[user@unix progs_as]$
```

Stekdən qeyd etdiyimiz kimi funksiyalara çağırışların realizə olunmasında geniş istifadə olunur. Növbəti mövzuda biz bu məsələ ilə məşğul olacayıq.

$7 Funksiyalar.

Funksiya çağırmaq üçün **call** instruksiyasından, funksiyadan geri qayıtmaq üçün isə ret instruksiyasından istifadə olunur. Funksiya çağırmazdan əvvəl ona ötürüləcək parametrlər və ya onların ünvanları stekə yerləşdirilməlidir.
Gəlin assembler dilində funksiyadan istifadəyə aid proqramlarla tanış olaq.
Funksiyadan istifadə etməklə iki ədədin cəmini hesablayan proqram tərtib edək.

```
 # prog15.s

       .data

       eded1:
       .long 23

       eded2:
      .long 45

       netice:
       .long

      .text
      .globl _start
      .type _start, @function

_start:
       # neticenin unvanin ve ededleri steke yerleshdiririk
        pushq $netice
        pushq eded1
        pushq eded2

        #cemle funksiyasin cagiririq
        call cemle

        #neticeni %ebx -e yaziriq
        movl netice, %ebx

son:
       movl $1, %eax
       int $0x80

# cemle funksiyasinin proqram kodu
         .type cemle, @function
cemle:
         pushq %rbp
         movq %rsp, %rbp

         #ikinci ededi %rax -e kocurek
         movq  16(%rbp), %rax

         #birinci ededi %rbx-e kocurek
         movq 24(%rbp), %rbx

         #bu iki ededi cemleyek
         addq %rax, %rbx

         #netice deyisheninin unvanin %rcx -e yazaq
         movq 32(%rbp), %rcx
```

```
        #aldigimiz cemi netice deyishenine yazaq
        movl %ebx, (%ecx)

        # %rbp ve %esp reqistrlerinin qiymetlerini berpa eded
        movq %rbp, %rsp
        popq %rbp

        #funksiyadan qayidaq
        ret
```

Proqramı icra edək:

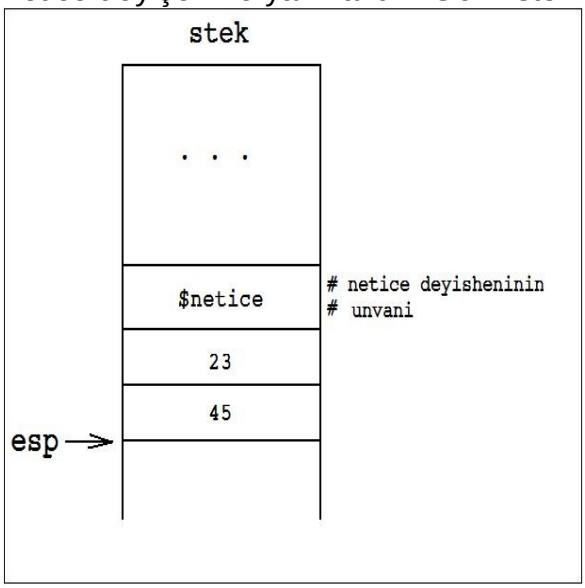
```
[user@unix progs_as]$
[user@unix progs_as]$ as prog15.s -o prog15.o
[user@unix progs_as]$ ld prog15.o -o prog15
[user@unix progs_as]$ ./prog15
[user@unix progs_as]$ echo $?
68
[user@unix progs_as]$
```

İzahı:

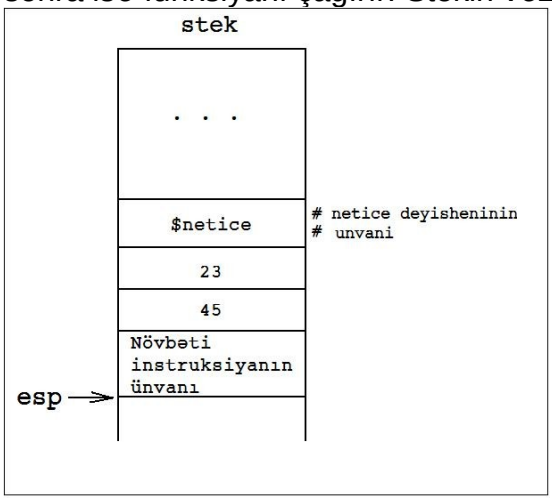
Proqramın məlumatlar hissəsində 3 dəyişən elan edirik, **eded1**, **eded2** və **netice**. Daha sonra aşağıdakı instruksiyalar vastəsilə bu məlumatları funksiyaya ötürmək üçün onları stekə yerləşdiririk.

pushq $netice
pushq eded1
pushq eded2

Əvvəlcə stekə netice dəyişəninin ünvanını, daha sonra **eded1** və **eded2** dəyişənlərinin qiymətlərini ötürürük. Funksiya **eded1** və **eded2** -nin cəmini hesablayıb netice dəyişəninə yazmalıdır. Gəlin stekin hazırki vəziyyətinə nəzər salaq.



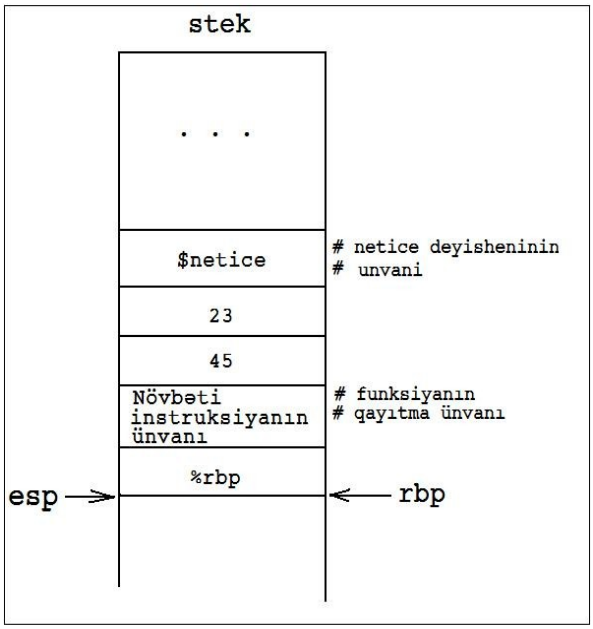
Daha sonra aşağıadakı instruksiya icra olunur:
call cemle
Bu instruksiya yerinə yetirilən zaman icraolunma **cemle** funksiyasına ötürüləcək. Qeyd edək ki, proqramın hər-hansı yerindən icraolunmanı başqa bir yerə yönləndirmək üçün biz **jmp** instruksiyasından istifadə edirdik. Bəs onda **cemle** funksiyasının proqram kodu üzərinə sürüşmək üçün niyə **jmp** instruksiyasından istifadə etmədik? Misal üçün aşağıdakı kimi: **jmp cemle** Bu halda da, icra olunma cemle: nişanına sürüşməli idi. Bunun izahını irəlidə funksiyanın çağırılması və ondan geri qayıtma bölməsində verəcəyik. Hələlik isə onu qeyd edim ki, call instruksiyası əvvəlcə **"NÖVBƏTİ İNSTRUKSİYANIN ÜNVANIN"** stekə yerləşdirir, daha sonra isə funksiyanı çağırır. Stekin vəziyyətinə nəzər salaq.



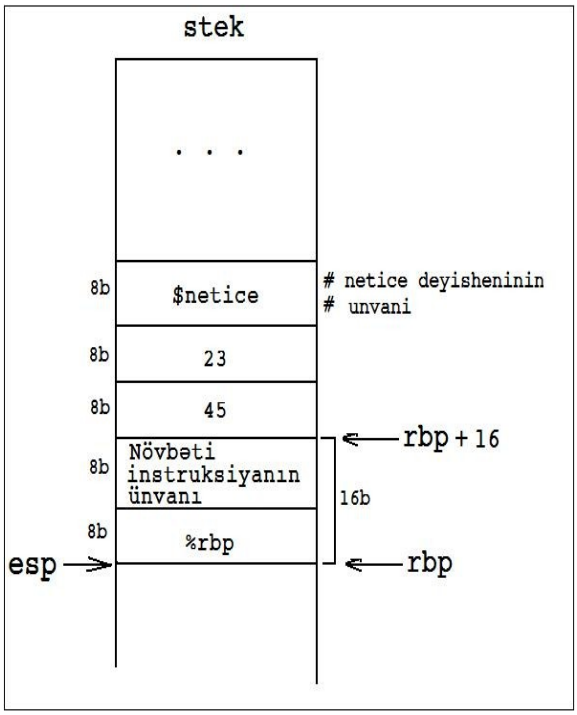
call cemle instruksiyası icraolunmanı **cemle** funksiyasına ötürür və **cemle** nişanına yerləşdirilmiş instruksiyalar icra olunmağa başlayır.
İlk olaraq aşağıdakı instruksiyalar yerinə yetirilir.

```
pushq %rbp
movq %rsp, %rbp
```

pushq %rbp instruksiyası **%rbp** reqistrinin qiymətini stekə yerləşdirir, **movq %rsp, %rbp** instruksiyası isə stek göstəricisinin qiymətini **%rbp** -reqistrinə köçürür. Bütün bunlar funksiyanın proqram kodunun icrası zamanı ona ötürülən parametrlərə müraciətedəbilmək və funksiyanın çağırılma yerinə qayıda bilmə üçündür. Stekin vəziyyətinə nəzər salaq.



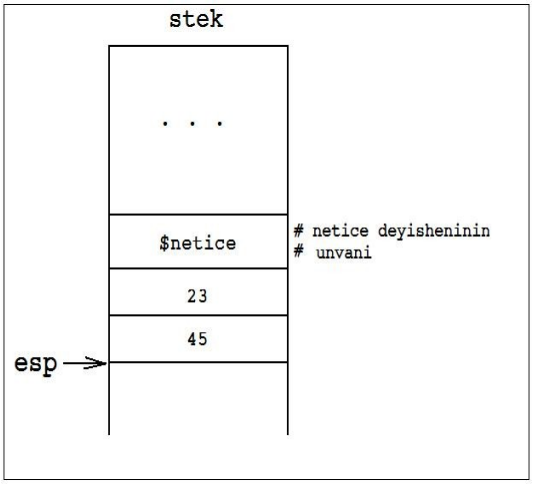
Bu işlər hazırlıq işləridir və bir qayda olaraq bütün funksiyalar məhs bu instruksiyaların icrası ilə başlayır. Daha sonra isə artıq tələb olunan işlər görülə bilər. Bizdən iki ədədi cəmləmək tələb olunur. Ədədlər stekə yerləşdirilib. Növbəti instruksiya ilə biz ikinci ədədi stekdən **%rax** reqistrinə köçürürük. Bunun üçün **movq 16(%rbp)**, **%rax** instruksiyasından istifadə edirik. Bildiyimiz kimi bu instruksiya **%rbp + 16** ünvanında yerləşən məlumatı **%rax** reqistrinə köçürür. Stekə push instruksiyası ilə məlumat yerləşdirəndə stek göstəricisi 8 bayt aşağı sürüşür (azalır). Yəni stekə yerləşdirilən hər bir elementə 8 bayt həcmində yer ayrılır. Aşağıdakı kimi:



Ən son stekə **%rbp** reiqstri, ondan əvvəl isə call instruksiyası tərəfindən növbəti icraolunma ünvanı, başqa sözlə funksiyanın işini bitirdikdən sonra qayıtmalı olduğu ünvan yerləşdirilib. Analoji olaraq
movq 24(%rbp), %rbx instruksiyası birinci ədədin qiymətini **%rbx** reqistrinə yazır. Daha sonra bu iki qiyməti cəmləyirik və nəticəni **%rbx** -də saxlayırıq.
addq %rax, %rbx
Cəm hesablanıb onu netice dəyişəninə yazmalıyıq. netice dəyişenin ünvanını stekdən **%rcx** reqistrinə köçürmək üçün aşağıdakı instruksiyanı icra edirik:
movq 32(%rbp), %rcx
Cəmi ünvanı **%ecx** -də yerləşən sahəyə, netice nişanına köçürürük:
movl **%ebx, (%ecx)**
ret instruksiyası ilə funksiyadan qayıtmadan öncə **%rsp** və **%rbp** -in funksiya çağırılan andakı qiymətlərini bərpa edirik.

```
movq %rbp, %rsp
popq %rbp
```

Funksiyadan qayıtmaq üçün **ret** instruksiyasını icra edirik. **ret** instruksiyası Stekin üst hissəsindəki məlumatı götürür və həmi ünvana keçid edir. **ret** instruksiyasından sonra stekin vəziyyəti aşağıdakı kimi olar:

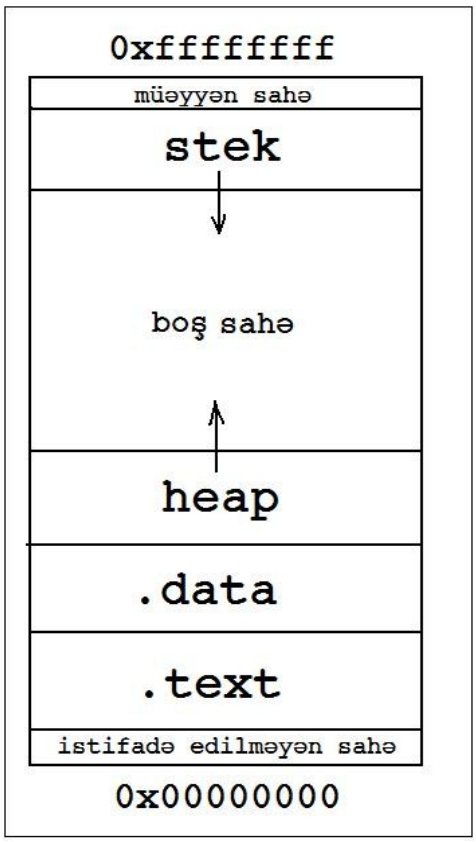


Funksiyanın çağırılması

Yuxarıda qeyd elədik ki, hər-hansı funksiyanı çağırmaq üçün **call** instruksiyasından istifadə edirik.

Geri qayıtmaq üçün isə **ret** instruksiyasından. Qeyd elədik ki, **call** istruksiya icra olunan zaman əvvəlcə növbəti icraolunacaq instruksiyanın ünvanın stekə yerləşdirir, **ret** isə stekin üst hissəsində yerləşən məlumatı götürüb həmin ünvana keçid edir.

Biz ilk mövzularda icraolunan proqramın strukturu barədə söz açmışdıq. Qeyd elədik ki, proqram yaddaşa yüklənən zaman təxminən aşağıdakı struktura malik olur:



Proqramın icraolunan kod hissəsi **.text** seqmentində yerləşir. Əməliyyatlar sistemi **.text** segmentini proqramın icraolunabilən faylından oxuyub, fiziki yaddaşa yükləyir və prosessorun **%rip** registrin proqramın başlanğıc ünvanına (**_start** nişanı) kökləyir. **%rip** registri prosessora növbəti icraolunacaq instruksiyanın ünvanın bildirir. Bu reqistrin qiymətini adi mov instruksiyası vastəsilə dəyişə bilmərik. Hər instruksiya yaddaşda (**.text** seqmenti) 8 bayt yer tutur (X86_64).
Hər instruksiya icra olunduqca **%rip** reqistrinin qiyməti avtomatik olaraq 8 bayt artır, yəni o növbəti icraolunmalı instruksiyanın üzərinə sürüşür. **%rip** reqistrinin qiymətini başqa yolla call və jmp instruksiyaları dəyişə bilir. **jmp** instruksiyası **%rip** reqistrinin qiymətini birbaşa verilmiş yeni ünvana kökləyir və prosessor icraolunmanı həmin yeni ünvandan davam edir. **jmp** istruksiyası əlavə heç bir iş görmür. **call** instruksiyası isə **jmp** -dan fərqli olaraq çağırılan zaman call -dan sonra gələn instruksiyanın yaddaşdakı ünvanın **(%rip + 8)** stekə yerləşdirir. Daha sonra **%rip** reqistrinin qiymətini verilmiş ünvana kökləyir. Həmin ünvanda istənilən proqram kodu yerləşə bilər. O koddan geri qayıdan instruksiya **call** -dan sonrakı yerdən bərpa olunmalıdır. Həmin ünvan isə **call** tərəfindən stekə yerləşdirilib. **ret** stekdən həmin ünvanı əldə edir və **%rip** reqistrinin qiymətin bu ünvana kökləyir.
Funksiyalardan istiadəyə aid digər proqram nümunələri.
Gəlin funksiyalardan istifadə etməklə aşağıdakı kimi bir proqram tərtib edək hansı ki istifadəçidən 2 eded qəbul edir, onların cəmini hesablayaraq çap edir.

```
#prog16.s
```

```
.data
         setir1:
        .ascii "Birinci ededi daxil edin\n"

          say1:
        .long 25

         setir2:
        .ascii "Ikinci ededi daxil edin"

         say2:
        .long 24

         setir3:
        .ascii "Sizin daxil etdiyiniz ededlerin cemi  = "

         say3:
        .long 40

         yeni_str:
        .ascii "\n”

         str:
        .ascii "            "

         eded1:
        .long 0

          eded2:
        .long 0

         netice:
        .long 12345

         onluq_simvollar:
        .ascii "0123456789"

          bufer:
        .rept 100
        .ascii "\0"

        .endr

.text

.globl _start
.type _start, @function

_start:

        # birinci ededin daxil edilmesini iste
        pushq say1
        pushq $setir1
        call  cap_et

        #binirci ededi setir sheklinde qebul et
```

```
        pushq $100
        pushq $bufer
        call  daxil_et

        #setri edede cevir
        pushq $bufer
        pushq $eded1
        call  setri_edede_cevir

        # ikinci ededin daxil edilmesini iste
        pushq say2
        pushq $setir2
        call  cap_et

        #ikinci ededi setir sheklinde qebul et
        pushq $100
        pushq $bufer
        call  daxil_et

        #setri edede cevir
        pushq $bufer
        pushq $eded2
        call  setri_edede_cevir

        #eded1 ile eded2 -ni cemle
        movl eded1, %eax
        addl eded2, %eax

        #cemi neticeye yaz
        movl %eax, netice

        #neticeni setre cevir
        pushq netice
        pushq $str
        call ededi_setre_cevir

        #netice barede melumati cap et
        pushq say3
        pushq $setir3
        call  cap_et

        # setir formasinda olan ededi cap et
        pushq $10
        pushq $str
        call  cap_et

        # yeni setir simvolun cap et
        pushq $1
        pushq $yeni_str
        call  cap_et

son:    movl $1, %eax
        int $0x80

#=========================================#
#                FUNKSIYALAR              #
#=========================================#
```

```
# cap_et funksiyasi
.type cap_et, @function

cap_et:
        pushq %rbp
        movq %rsp, %rbp

        movl $4, %eax
        movl $1, %ebx
        movq 16(%rbp),%rcx
        movq 24(%rbp),%rdx
        int  $0x80

        popq %rbp
        ret

# daxil_et funksiyasi
.type cap_et, @function

daxil_et:
        pushq %rbp
        movq %rsp, %rbp

        movl $3, %eax
        movl $0, %ebx
        movq 16(%rbp),%rcx
        movq 24(%rbp),%rdx
        int  $0x80

        popq %rbp
        ret

#setiri edede cevir
.type setri_edede_cevir, @function

setri_edede_cevir:

        pushq %rbp
        movq %rsp, %rbp

# buferdeki setri edede cevir
        subq $8, %rsp

        movl $0, %esi
        movl $0, -8(%rbp)
        movl $1, %edi

s_e_c_dovr:
        movb $'\n', %bh
        movl 24(%rbp), %ecx
        addl %esi, %ecx
        movb (%ecx), %ah
        cmpb %bh, %ah
        je   s_e_c_dovr_son
```

```
        sub  $48, %ah
        movl -8(%rbp), %ebx
        imull $10, %ebx
        movl %ebx, -8(%rbp)
        addb %ah, -8(%rbp)
        incl %esi
        jmp  s_e_c_dovr

s_e_c_dovr_son:
        movl 16(%rbp),%ecx
        movl -8(%rbp),%ebx
        movl %ebx, (%ecx)

        addq $8, %rsp
        popq %rbp
        ret

#ededi setre cevir
.type ededi_setre_cevir, @function

ededi_setre_cevir:

        pushq %rbp
        movq %rsp, %rbp

        subq $8, %rsp
        movl $0, -8(%rbp)
e_s_c_dovr:
        movl $10, %edi
        movl $0, %edx
        movl 24(%rbp), %eax
        div  %edi

        movl %eax, 24(%rbp)

        xorq %rbx, %rbx
        movb onluq_simvollar(%edx), %bh
        pushq %rbx

        incl -8(%rbp)

        movl $0, %ebx
        movl 24(%rbp), %eax
        cmpl %eax, %ebx
        je   e_s_c_dovr_son

        jmp e_s_c_dovr

e_s_c_dovr_son:

        movl -8(%rbp), %ecx
        movl 16(%rbp), %eax
        xorl %ebx, %ebx
stekden_setre:
        cmpl %ebx, %ecx
        je e_s_c_son
```

```
        popq %rdx
        movb %dh, (%eax)
        incl %eax
        incl %ebx
        jmp  stekden_setre

e_s_c_son:
        addq $8, %rsp
        popq %rbp
        ret
```

Kompilyasiya və icra etsək:

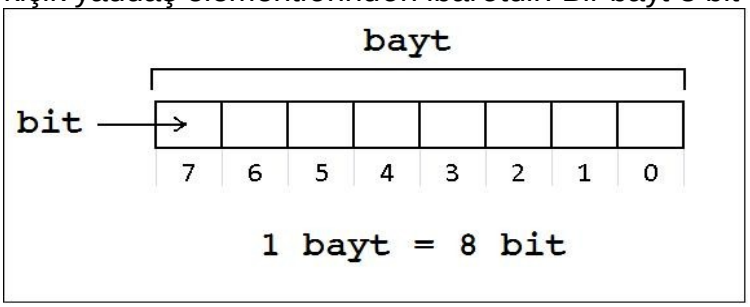
```
[user@unix progs_as]$
[user@unix progs_as]$ as prog16.s -o prog16.o
[user@unix progs_as]$ ld prog16.o -o prog16
[user@unix progs_as]$ ./prog16
Birinci ededi daxil edin
345
Ikinci ededi daxil edin
5678
Sizin daxil etdiyiniz ededlerin cemi  = 6023
[user@unix progs_as]$
```

$8 Say sistemləri.

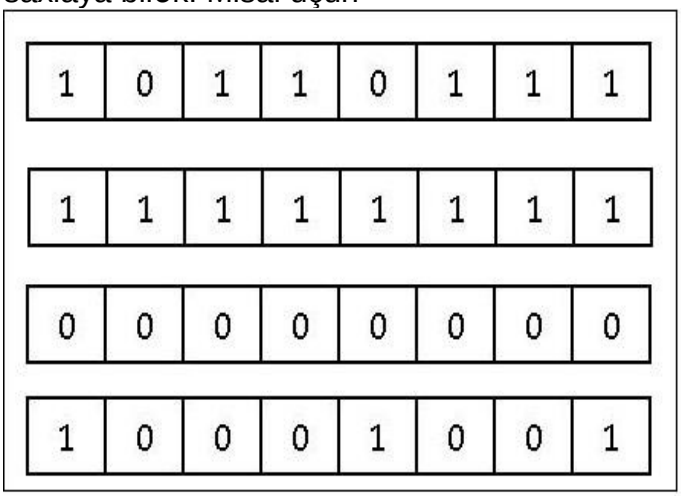
Biz indiyə kimi proqramlarımızda yalnız onluq ədədlər və ascii simvollardan istifadə etdik. Proqramlaşdırmada, xüsusilə sistem proqramlaşdırmada yaddaş ünvanları ilə işləyərkən 16-lıq(hexal) say sistemləri ilə işləmək daha məsləhətdir. Bəzən isə bit əməliyyatları (irəlidə) yerinə yetirərkən ədələrin ikili say sistemindəki halı ilə işləmək lazım gəlir. Bir proqramçı kimi, say sistemlərinin mahiyyətini bilmək, bir say sistemindən digərinə keçməyi bacarmaq mühümdür.

Gəlin bu məsələlər ilə məşğul olaq.

Məlumat ölçüsü vahidi olaraq bayt qəbul olunur. Bayt özü də bit adlandırılan daha kiçik yaddaş elementlərindən ibarətdir. Bir bayt 8 bit -dən ibarətdir.



Bir bit özündə 2 müxtəlif qiymət: 0 və ya 1 qiymətlərini saxlaya bilər(eyni vaxtda yalnız birini). Bu isə imkan verir ki bir baytda 256 bitlərin müxtəlif düzülüşünü saxlaya bilək. Misal üçün



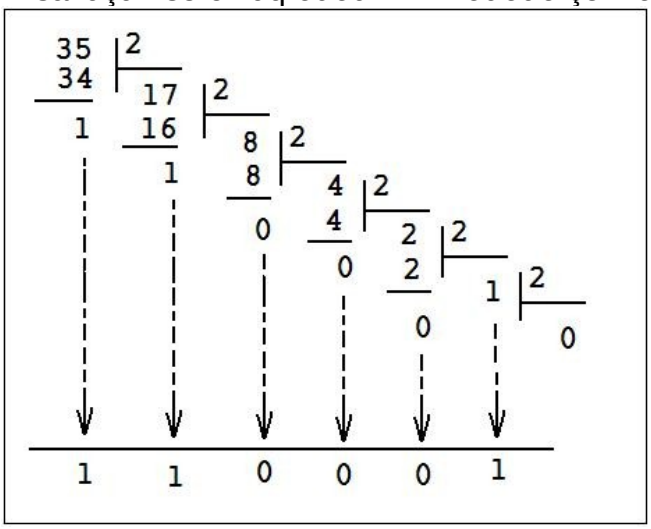
Kompüter yaddaşında bütün məlumatlar bu şəkildə olur, baytlar ardıcıllığı, onlar da öz növbəsində bitlər ardıcıllığı şəklində. Məlumatın bitlərdə yerləşdirlimiş forması onun ikili forması adlandırılır. Burada məlumat dedikdə söhbət yalnız ədədlərdən gedir. Bütün məlumatlar onluq və ya 16-lıq ədədlərlə ifadə olunur, bu ədədlər isə yaddaşda ikili say sistemində yerləşdirilir.

İkili say sistemi

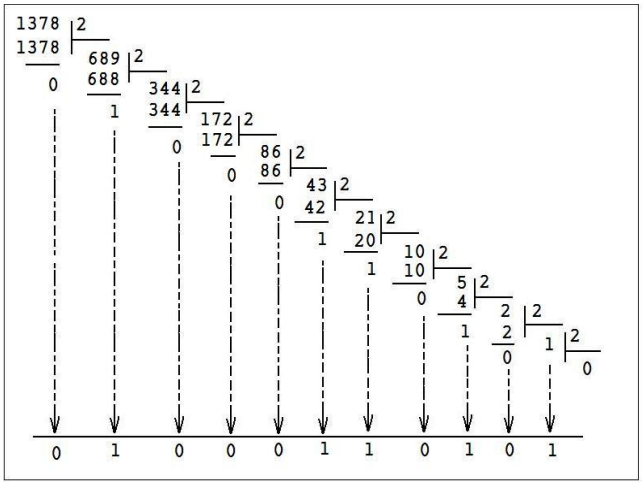
Adından da göründüyü kimi ikili say sistemində bütün ədədlər cəmi 2 rəqəm: 0 və 1 vastəsilə ifadə olunur. İkili ədədlərə misal olaraq **0001010010, 1111, 011001111, 0 ,11** v.s. misal göstərmək olar.

Aşağıdakı qaydadan istifadə etməklə verilmiş ikili ədədi onluq ədədə çevirə bilərik. Tutaq ki, hər hansı **[x(n)][x(n-1)][x(n-2)]...[x2][x1]** ikili ədədi verilmişdir. Burada hər bir **x(k), 1<=k<=n** 0 və ya 1 qiyməti ala bilər. İkili ədədi onluq ədədə çevirmək üçün

aşağıdakı düsturdan istifad' edirik.
onluq_eded = x(n)*2^(n - 1) + x(n-1)*2^(n-2)+ ... x(2)*2^1 + x(1)*2^0;
Burada **2^k** ifadəsi 2 üstü k mənasını verir, yəni **2*2*2 ... *2** k dəfə.
Misal üçün **1000101001** ikili ədədini onluq ədədə çevirək. Burada rəqəmlərin sayı = 10. Ona görə yazırıq:
1000101001 = 1*2^9 + 0*2^8 + 0*2^7 + 0*2^6 + 1*2^5 + 0*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0
1000101001 = 1*2^9 + 1*2^5 + 1*2^3 + 1*2^0 = 512 + 32 + 8 + 1 = 553
Başqa bir ikili ədədi onluq ədədə çevirək, **00000100111101**. Bu halda rəqəmlərin sayı 15 -dir.
İlk rəqəmlər 0 olduğundan onların 2-nin üstünə hasili 0 edəcək. Ona görə onları buraxa və birbaşa 1 ilə başlayan rəqəmlərdən hesablamanı apara bilərik.
00000100111101 = 1*2^8 + 0*2^7 + 0*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 =
= 256 + 32 + 16 + 8 + 4 + 1 = 317
Indi isə onluq ədədin ikili ədədə çevrilməsi ilə məşğul olaq.
Onluq ədədi ikili ədədə çevirmək üçün aşağıdakı qaydadan istifadə edirik:
Onluq ədədi ikiyə bölürük, qalığı yadda saxlayırıq (qalıq 0 və ya 1 ola bilər), qismәti isə yenidən ikiyə bölürük. Yenidən qalığı yadda saxlayırıq və qismәti 2 -yə bölürük. Bu prosesi qismətdə 0 qiyməti alınana kimi davam etdiririk. Sonda bütün aldığımız qalıqları əks sıra ilə düzsək verilmiş onluq ədədə müvafiq ikili ədədi almış olarıq.
Misal üçün 35 onluq ədədini ikili ədədə çevirək.



Qalıqları sondan əvvələ düzsək alarıq: **100011**
Başqa bir misal, **1378** onluq ədədini ikili ədədə çevirək:



Qalıqları sondan əvvələ düzsək alarıq: **10101100010**

16 - lıq say sistemi.

Yaddaş ünvanları ilə işləyərkən ədədlərin 16-lıq say sistemindəki ifadəsindən istifadə etmək çox rahatdır. 16 - say sistemindəki rəqəmlər 16 simvol vastəsilə ifadə olunur. Bu simvollar aşağıdakılardır:

0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f

Burada **0,1,2 ... 9** rəqəmləri onluq say sitemindəki müvafiq ədədlərə, **a, b, c, d, e, f** 16 -lıq rəqəmləri isə onluq sistemdəki **10, 11, 12, 13, 14** və **15** ədədlərinə uyğun gəlir.

16 -lıq ədədin 10 -luq ədədə çevrilməsi.

16 -lıq ədədi 10 -luq ədədə çevirmək üçün aşağıdakı qaydadan istiadə edirik. Tutaq ki,

[x(n)][x(n-1)]...[x(2)][x(1)] 16-lıq ədədi verilmişdir. Bu ədədi aşağıdakı düsturla 10 -lu ədədə çevirə bilərik:

[x(n)][x(n-1)]...[x(2)][x(1)] = x(n)*16^(n - 1) + x(n-1)*16^(n-2)+ ... x(2)*16^1 + x(1)*16^0;

Misal üçün **1256f** 16-lıq ədədi 10-luq ədədə aşağıdakı kimi çevirə bilərik.

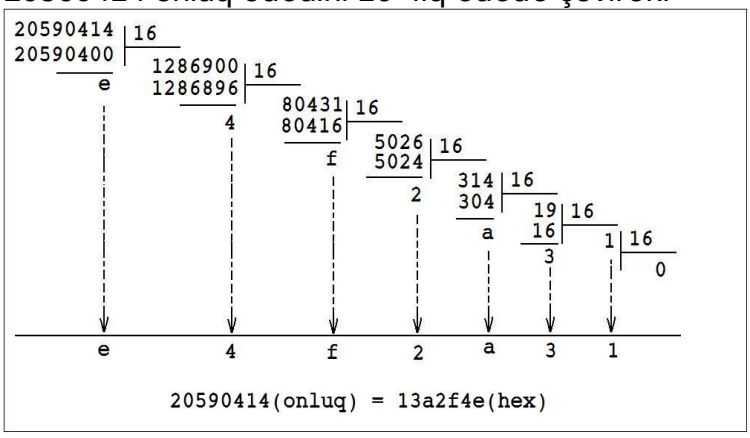
1256f = 1*16^4 + 2*16^3 + 5*16^2 + 6*16^1 + 15*16^0 = 65536 + 8292 + 1280 + 96 + 15 = 75219

Başqa bir 16-lıq ədədi 10-luq ədədə çevirək:

f3a034b1 = 15*16^7 + 3*16^6 + 10*16^5 + 0*16^4 + 3*16^3 + 4*16^2 + 11*16^1 + 1*16^0 =
= 4087362737

Onluq ədədlərin 16-lıq say sisteminə çevirilməsi.

Onluq ədədləri 16-lıq say sisteminə çevirərkən ikilik istemə çevidiyimiz qaydanı tətbiq edəcəyik, lakin bölünən olaraq 2 yox, 16 -dan istifadə edəcəyik. Misal üçün 20560414 onluq ədədini 16 -lıq ədədə çevirək:



16-lıq sistemdə verilmiş ədələri 10-luq ədədlərlə qarışdırmamaq üçün onların əvvəlinə **"0x"** işarələməsi artırılır. Misal üçün **0xff450012** şəklində. İkilik ədədlərin qarşısına isə **"0b"** işarələməsi artırılır. Misal üçün **0b11010011101001** şəklində.
Elə proqram tərtib edək ki, verilmiş ikili ədədi onluq ədədə çevirsin:

```
        #prog17.s
        #ikili ededi 10-luq edede ceviren program

        .data

setir1:
        .ascii "Ikili ededi daxil edin\n"

say1:
        .long 23

yeni_str:
        .ascii "\n"

str:
        .ascii "          "

eded1:
        .long 0

onluq_simvollar:
        .ascii "0123456789"

bufer:
        .rept 100
        .ascii "\0"
        .endr

.text

.globl _start
.type _start, @function
_start:
        # birinci ededin daxil edilmesini iste
        pushq say1
        pushq $setir1
        call  cap_et

        #ikili ededi setir sheklinde qebul et
        pushq $100
        pushq $bufer
        call  daxil_et

        #setri edede cevir
        pushq $bufer
        pushq $eded1
        call  ikili_setri_edede_cevir

        #neticeni cap etmek ucun yeniden setre cevir
        pushq eded1
        pushq $str
        call ededi_setre_cevir
```

```
        # setir formasinda olan ededi cap et
        pushq $10
        pushq $str
        call  cap_et

        # yeni setir simvolun cap et
        pushq $1
        pushq $yeni_str
        call  cap_et

son:
        movl $1, %eax
        int $0x80

#=========================================#
#                FUNKSIYALAR              #
#=========================================#

# cap_et funksiyasi
.type cap_et, @function
cap_et:
        pushq %rbp
        movq %rsp, %rbp

        movl $4, %eax
        movl $1, %ebx
        movq 16(%rbp),%rcx
        movq 24(%rbp),%rdx
        int  $0x80

        popq %rbp
        ret


# daxil_et funksiyasi
.type cap_et, @function
daxil_et:
        pushq %rbp
        movq %rsp, %rbp

        movl $3, %eax
        movl $0, %ebx
        movq 16(%rbp),%rcx
        movq 24(%rbp),%rdx
        int  $0x80

        popq %rbp
        ret


#setiri edede cevir
.type setri_edede_cevir, @function


ikili_setri_edede_cevir:

        pushq %rbp
```

```
        movq %rsp, %rbp

       # buferdeki setri ededе cevir

        subq $8, %rsp

        #simvollarin sayini hesablayaq
        #sonuncu simvol '\n' simvoludur

        movl $0, %ecx
        movl 24(%rbp), %esi
isec_dovr:
        movb (%esi, %ecx, 1), %ah
        cmpb $'\n', %ah
        je   isec_dovr_son
        incl %ecx
        jmp  isec_dovr

isec_dovr_son:


        movl $1, %ebx
        movl $0, -8(%rbp)

        movl $0, %eax
        decl %ecx

yeni_isec_dovr:

        movb (%esi, %ecx, 1), %al
        subb $'0', %al
        movl %ebx, %edx
        imull %eax, %edx
        addl %edx, -8(%rbp)
        shll $1, %ebx
        decl %ecx
        cmpl $0, %ecx
        jl yeni_isec_dovr_son
        jmp yeni_isec_dovr


yeni_isec_dovr_son:

        movl 16(%rbp), %ebx
        movl -8(%rbp), %eax
        movl %eax, (%ebx)

        addq $8, %rsp
        popq %rbp
        ret

#ededi setre cevir
.type ededi_setre_cevir, @function

ededi_setre_cevir:

        pushq %rbp
```

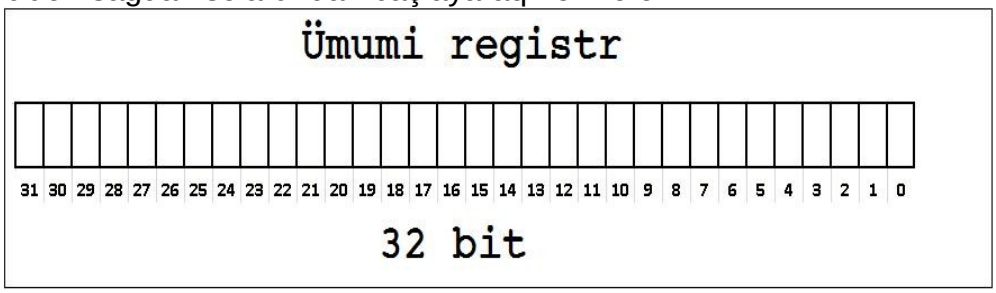
```
        movq %rsp, %rbp

        subq $8, %rsp
        movl $0, -8(%rbp)
e_s_c_dovr:
        movl $10, %edi
        movl $0, %edx
        movl 24(%rbp), %eax
        div  %edi

        movl %eax, 24(%rbp)

        xorq %rbx, %rbx
        movb onluq_simvollar(%edx), %bh
        pushq %rbx

        incl -8(%rbp)

        movl $0, %ebx
        movl 24(%rbp), %eax
        cmpl %eax, %ebx
        je   e_s_c_dovr_son

        jmp e_s_c_dovr

e_s_c_dovr_son:

        movl -8(%rbp), %ecx
        movl 16(%rbp), %eax
        xorl %ebx, %ebx
stekden_setre:
        cmpl %ebx, %ecx
        je e_s_c_son

        popq %rdx
        movb %dh, (%eax)
        incl %eax
        incl %ebx
        jmp  stekden_setre

e_s_c_son:
        addq $8, %rsp
        popq %rbp
        ret
```

Icra edək:

```
[user@unix progs_as]$
[user@unix progs_as]$ as prog17.s  -o prog17.o
[user@unix progs_as]$ ld prog17.o  -o prog17
[user@unix progs_as]$ ./prog17
Ikili ededi daxil edin
110101010100111
27303
[user@unix progs_as]$
[user@unix progs_as]$
```

$9 Bit Əməliyyatları.

Biz qeyd elədik ki, məlumatın ölçü vahidi bayt -dır. Lakin bəzən verilmiş baytın və ya baytlar ardıcıllığının hansısa mövqedə yerləşən bir və ya bir neçə bitinin qiymətini örgənmək, dəyişmək, sola - sağa sürüşdürmək v.s. əməliyyatlar aparmaq lazım gəlir. Bilirik ki, **%eax** registrinin ölçüsü 4 baytdır, başqa sözlə 32 bit. Reqistrin bitləri sağdan sola 0 -dan başlayaraq nömrələnir.



Sola - Sağa sürüşmə

Bitlər ardıcıllığını sola və ya sağa sürüşdürmək üçün shll və shrl instruksiyalarından istifadə olunur. Misal üçün shrl say, arq instruksiyası arq məlumatının bitlərini say vahid sağa sürüşdürür. Boşalan bitlərin yerinə 0-lar yazılır. Misal üçün **0011010** bitlər ardıcıllığını 1 vahid sağa sürüşdürsək **0001101** alarıq. **001110** bitlər ardıcıllığını 2 vahid sola sürüşdürsək isə, **111000** alarıq.

Və , Və ya əməliyyatları

Bitlər ardıcıllığı üzərində **VƏ** , **VƏ YA** əməliyyatları icra etmək üçün andl və orl instruksiyalarından istifadə edirlər.

and instruksiyası 2 arqument qəbul edir:

and arq1, arq2 şəklində. **arq1** ilə **arq2** -in müvafiq bitlərinə **VƏ** əməliyyatı tətbiq edərək nəticəni **arg2** -də saxlayır. Bitlərə **VƏ** əməliyyatının tətbiqi aşağıdakı kimidir

```
   BİT1 | BİT2 | BİT1 Və BİT2
-------------------------------
     1  |  1   |    1
     1  |  0   |    0
     0  |  0   |    0
     0  |  1   |    0
```

Cədvəldən gördüyümüz kimi iki bitin **VƏ** -si yalnız və yalnız onların hər ikisinin qiyməti 1 olduqda 1 qiyməti alır.

Misal üçün **0101** ilə **1100** bitlər ardıcıllığının and -i, **0100** -a bərabər olar,aşağıdakı kimi:

```
0|1|0|1
1|1|0|0
-------
0|1|0|0
```

Başqa misala baxaq, **0100010101011101** ilə **0110111100001011** in and -i **0100010100000101 1** olar, aşağıdakı kimi.

```
0|1|0|0|0|1|0|1|0|1|0|1|1|1|0|1|1
0|1|1|0|1|1|1|1|1|0|0|0|0|1|0|1|1
---------------------------------
0|1|0|0|0|1|0|1|0|0|0|0|0|1|0|1|1
```

Bitlərə **VƏ YA** əməliyyatının tətbiqi aşağıdakı kimidir:

```
   BİT1 | BİT2 | BİT1 Və BİT2
-------------------------------
     1  |  1   |    1
     1  |  0   |    1
     0  |  0   |    0
     0  |  1   |    1
```

Verilimiş iki bitə **VƏ YA** - orl əməliyyatının tətbiqi cədvəldən gördüyümüz kimi hər iki bit 0 olduqda 0 qiyməti alır, qalan bütün hallarda, yəni heç olmasa ikisindən biri və ya hər ikisi 1 olduqda 1 qiyməti alır.

Maskalama

Sola - sağa sürüşdürmə, **VƏ , VƏ YA** əməliyyatlarının birgə kompazisiyasından istifadə etməklə bitlər ardıcıllığı üzərində maraqlı əməliyyatlar yerinə yetirmək olar. Misal üçün tutaq ki, **%eax** registrinin ilk bitinin qiymətini örgənmək istəyirik. Bunun üçün 0-cı bitin maskasından istifadə etməliyik. 0 -cı bitin maskası aşağıadkı kimi olar:

00000000000000000000000000000001

Uyğun olaraq 1-ci bitin maskası

00000000000000000000000000000010

, 31 -ci bitin maskası

10000000000000000000000000000000,

15 -ci bitin maskası

00000000000000001000000000000000

kimi olar.

Bir daha qeyd edim ki, bitlərin nömrələnməsi 0-dan başlayır.

Qiymətini tapmaq istədiyimiz bitin maskasını **%ebx** -ə köçürək.

movl 0b00000000000000000000000000000001, %ebx

Artıq **%ebx** -də ilk bitin maskası yerləşir. Daha sonra **%eax** registrinə **%ebx** reqistri ilə və - and əməliyyatını tətbiq etsək ilk **%eax** registrinin ilk bitinin qiymətini alarıq.

andl %ebx, %eax

Belə ki, **%ebx** -in ilk bitindən başqa bütün yerdəqalan bitləri 0 olduğundan **%eax** -lə VƏ əməliyyatı zamanı **%eax** -in ilk bitindən savayı yerdə qalan bütün bitləri silinəcək, 0-ra bərabərləşəcək. Yekun qiymət isə **%eax** -in ilk bitinin qiymətindən asılı olacaq, belə ki, əgər **%eax** -in ilk biti 0-sa onda nəticə -də 0, əks halda isə 1 olar. Beləliklə biz **%eax** -in ilk bitinin qiymətini təyin etmiş olduq.

Gəlin konkret proqram nümunəsi ilə bunu test edək.

```
        #prog19.s

        # %eax registrinin ilk bitini cap eden proqram


        .data
```

```
setir1:
        .ascii " %eax registrinin qiymetini ikili formada daxil edin\n"

say1:
        .long 53

setir2:
        .ascii "ilk bitin qiymeti "

say2:
        .long 18

yeni_str:
        .ascii "\n"

str:
        .ascii "          "

eded1:
        .long 0

onluq_simvollar:
        .ascii "0123456789"

bufer:
        .rept 100
        .ascii "\0"
        .endr

.text

.globl _start
.type _start, @function

_start:
        # %eax registrinin qiymetinin daxil edilmesini iste

        pushq say1
        pushq $setir1
        call  cap_et

        #qiymeti setir sheklinde qebul et

        pushq $100
        pushq $bufer
        call  daxil_et

        #setri edede cevir

        pushq $bufer
        pushq $eded1
        call  ikili_setri_edede_cevir

        #neticeni %eax -e kocur

        movl eded1, %eax
```

```
        # ilk bitin maskasin %ebx -e yerleşdirek

        #eslinde bu halda  movl $1, %ebx yazmaq da kifayet eder

        movl $0b00000000000000000000000000000001, %ebx

        #eax -i #ebx -le maskalayaq

        andl %ebx, %eax

        #indi %eax -de ilk bitin qiymeti durur

        #onu cap edek

        movl %eax, eded1

        #neticeni cap etmek ucun yeniden setre cevir

        pushq eded1
        pushq $str
        call ededi_setre_cevir

        # melumatlandirici setir

        pushq say2
        pushq $setir2
        call  cap_et

        # setir formasinda olan ededi cap et

        pushq $10
        pushq $str
        call  cap_et

        # yeni setir simvolun cap et

        pushq $1
        pushq $yeni_str
        call  cap_et

son:
        movl $1, %eax
        int $0x80

#=========================================#

#               FUNKSIYALAR               #

#=========================================#

# cap_et funksiyasi

.type cap_et, @function

cap_et:
        pushq %rbp
        movq %rsp, %rbp
```

```
        movl $4, %eax
        movl $1, %ebx
        movq 16(%rbp),%rcx
        movq 24(%rbp),%rdx
        int  $0x80

        popq %rbp
        ret

# daxil_et funksiyasi

.type cap_et, @function

daxil_et:
        pushq %rbp
        movq %rsp, %rbp

        movl $3, %eax
        movl $0, %ebx
        movq 16(%rbp),%rcx
        movq 24(%rbp),%rdx
        int  $0x80

        popq %rbp
        ret

#setiri edede cevir

.type setri_edede_cevir, @function

ikili_setri_edede_cevir:
        pushq %rbp
        movq %rsp, %rbp

# buferdeki setri edede cevir

        subq $8, %rsp

        #simvollarin sayini hesablayaq

        #sonuncu simvol '\n' simvoludur

        movl $0, %ecx
        movl 24(%rbp), %esi
isec_dovr:
        movb (%esi, %ecx, 1), %ah
        cmpb $'\n', %ah
        je   isec_dovr_son
        incl %ecx
        jmp  isec_dovr

isec_dovr_son:
        movl $1, %ebx
        movl $0, -8(%rbp)

        movl $0, %eax
```

```
        decl %ecx

yeni_isec_dovr:
        movb (%esi, %ecx, 1), %al
        subb $'0', %al
        movl %ebx, %edx
        imull %eax, %edx
        addl %edx, -8(%rbp)
        shll $1, %ebx
        decl %ecx
        cmpl $0, %ecx
        jl yeni_isec_dovr_son
        jmp yeni_isec_dovr

yeni_isec_dovr_son:
        movl 16(%rbp), %ebx
        movl -8(%rbp), %eax
        movl %eax, (%ebx)

        addq $8, %rsp
        popq %rbp
        ret

#ededi setre cevir

.type ededi_setre_cevir, @function

ededi_setre_cevir:
        pushq %rbp
        movq %rsp, %rbp

        subq $8, %rsp
        movl $0, -8(%rbp)
e_s_c_dovr:
        movl $10, %edi
        movl $0, %edx
        movl 24(%rbp), %eax
        div  %edi

        movl %eax, 24(%rbp)

        xorq %rbx, %rbx
        movb onluq_simvollar(%edx), %bh
        pushq %rbx

        incl -8(%rbp)

        movl $0, %ebx
        movl 24(%rbp), %eax
        cmpl %eax, %ebx
        je   e_s_c_dovr_son

        jmp e_s_c_dovr

e_s_c_dovr_son:
        movl -8(%rbp), %ecx
        movl 16(%rbp), %eax
```

```
        xorl %ebx, %ebx
stekden_setre:
        cmpl %ebx, %ecx
        je e_s_c_son

        popq %rdx
        movb %dh, (%eax)
        incl %eax
        incl %ebx
        jmp  stekden_setre

e_s_c_son:
        addq $8, %rsp
        popq %rbp
        ret
```

Nəticəni yoxlayaq:

```
[user@unix progs_as]$ as prog19.s -o prog19.o
[user@unix progs_as]$ ld prog19.o -o prog19
[user@unix progs_as]$ ./prog19
 %eax registrinin qiymetini ikili formada daxil edin
10101011010101
ilk bitin qiymeti 1
[user@unix progs_as]$ ./prog19
 %eax registrinin qiymetini ikili formada daxil edin
1001010
ilk bitin qiymeti 0
[user@unix progs_as]$
[user@unix progs_as]$
```

$10 Sistem Proqramlaşdırma.

Biz assembler dilində dəyişən tipləri, dəyişənlərin elanı, registrlər, ümumişlək instruksiyalar, yaddaşa müraciət üsulları, stek və funksiyalar , say sistemləri, bit əməliyyatları ilə tanış olduq. Bütün bunlar istifadəçi proqramlaşdırmanı kifayət qədər əhatə edir. Assembler dilinin sistem proqramlaşdırmaya aid olan hissəsi isə həm mürəkkəbliyinə, həm də həcminə görə istifadəçi proqramlaşdırmadan qat-qat böyükdür. İşin problem tərəfi odur ki, sistem proqram kodları istifadəçi proqramları kimi istədiyimiz vaxt kompilyasiya və icra edə bilmərik. Sistem proqramları kodu əməliyyatlar sisteminin proqram kodudur və yalnız bir dəfə kompüter yüklənəndə yüklənir və kompüterin bütün fəaliyyəti boyu icra olunur, Kompüteri, onun resurslarını, avadanlıqlarını,fiziki yaddaşı, istifadəçi proqramlarını v.s. idarə edir. Sistem proqram kodlarının işinə misal olaraq prosessorun rejimlərini dəyişmək, kompüterin portlarına məlumat yazmaq(oxumaq), avadanlıqları sistemə tanıdıb konfiqurasiya etmək, avadanlıqların işini idarə etmək, kəsilmələrə cavab vermək v.s. göstərmək olar.

Assembler barəsində daha geniş məlumatı(istifadəçi və sistem proqramlaşdırma) **Intel® 64 and IA-32 Architectures Software Developer's Manual** - sənədindən əldə edə bilərsiniz.

İstifadəçi proqramlaşdırmaya aid mövzuların bir qisminə bu kitabda proqram nümunələri ilə birlikdə baxdıq. Sistem proqramlaşdırmaya aid proqram nümunələrini isə açıq qaynaqlı nüvə kodlarından əldə edə bilərsiniz. Açıq kodlu nüvələrə ən yaxşı nümunə **Linux** -u misal göstərə və çox məsləhət görə bilərəm. Həm proqram kodunun gözəlliyi, həm daha çox sənədlərin olması başlanğıc üçün **Linux** - u tək seçim edir. Ümumiyətlə mən deyərdim ki, sistem proqramlaşdırmaya gedən yol **Linux** - un kompilyasiyasından, ilk nüvə modulunun tərtibindən, ilk sistem funksiyasının hazırlanmasından, daha sonra artıq fayllar sistemi, virtual yaddaş v.s. məsələlərin örgənilməsindən başlayır.

Linux nüvə proqramlaşdırmaya aid aşağıdakı kitabları məsləhət görə bilərəm:

1. **Professional Linux Kernel Architecture**
2. **Understanding the Linux Kernel**
3. **Linux Kernel Development**
4. **Memory Management in Linux**
5. **Understanding The Linux Virtual Memory Manager**

$11 Problemlər.

Bu Assembler proqramlaşdırma dili kitabının ilk buraxılışıdır. Əlbəttdə ki, səhvlər qaçınılmazdır. Əvvəl nəzərdə tutduğumuz bəzi hissələri çaşqınlıq yaratmamaq və praktikilik nöqteyi nəzərindən dərsliyə salmamaq qərarına gəldik. Əsas diqqət yetirilməli məqam proqram kodlarıdır. Onları başa düşmək üçün dəfələrlə təkrar-təkrar kod tədqiq olunmalıdır.

Bu buraxılışın ən böyük çatışmamazlığı 64-32 bit problemləridir. Belə ki, proqram kodları 64 bitlik arxitektura üçün nəzərdə tutulub, ona görə 32 bitlik maşınlarda bu proqramları icra etmək mümkün deyil.

Əgər proqramların testi problem yaratsa onda rəsmi sayta ilkaddimlar.com müraciət edə bilərsiniz. Müraciətləri nəzərə alıb, biz proqramların 32 -bitlik versiyasını da kitaba əlavə edərik.

Texniki istənilən sual ilə bağlı cavabler.net saytına müraciət edə bilərsiniz.

Kitabın gələcək versiyalarının hazırlanmasında, mövcud mövzuların yenilənməsində, proqram nümunələrinin artırılmasında, səhvlərin düzəldilməsində könüllü köməklik göstərmək istəyənlər müəlliflə ahmed.sadikhov@gmail.com ünvanından əlaqə saxlaya bilərlər.

Əhməd Sadıxov.