Assembler

proqramlaşdırma dili

Əhməd Sadıxov

2– ci buraxılış

25.02.2014

Önsöz

Bu kitabda Assembler proqamlaşdırma dilindən bəhs olunur. Kitab assembler dilinə yeni başlayanlar üçün nəzərdə tutulsa da, mövzular orta peşəkar səviyyədə əhatə olunub . Kitabdan istifadə edə bilmək üçün oxucunun hər-hansı proqramlaşdırma dilini bilməsi, funksiyaların çağırılması, funksiyalara parametr ötürülməsi və göstəricilər mövzularını bilməsi mütləq tələb olunur. Yəni hazırki material hər-hansı proqramlaşdırma dilini mükəmməl bilən proqramçılar üçündür. Baxmayaraq ki, kitabda heç bir başqa proqramlaşdırma dilinə istinad verilmir, lakin bir çox mövzuların mənimsənməsi peşəkar səviyyəsinə yaxın, bəzən isə tam peşəkar proqramlaşdırma təcrübəsi tələb edir.

Arxitektura

Bu kitabda hal-hazırda çox geniş yayılmış X86 arxitekturalı prosessorların (İNTEL, AMD ...) assembler dili izah olunur.

Sintaksis

X86 arxitekturası üçün assembler proqramlaşdırma dilinin bir-birinə oxşar müxtəlif sintaksisləri mövcuddur. Bu kitabda daha asan başa düşülən və geniş yayılmış AT&T sintaksisindən istifadə olunur.

Oxucu kütləsi

Kitab assemblerlə maraqlanan hamı üçün faydalı ola bilər, lakin əsas oxucu kütləsi yeni başlayan sistem proqramçılar hesab olunur.

Əməliyyatlar sistemi və kompilyator

Nümunə kodlar GNU/Linux sistemlərində kompilyasiya və icra üçün nəzərdə tutulub. Assembleri örgənmək üçün ən əlverişli əməliyyatlar sistemi GNU/Linux sistemləridir. Windowsda assembleri örgənmək və başa düşmək nisbətən çətindir, çünki Windows sistemləri əsasən istifadəçi proqramlaşdırmada istifadə olunur. İstifadəçi interfeysi məsələsində Windows çox irəlidə olsada, performans və təhlükəsizlik məsələlərində Unix sistemlərindən geri qalır. Bu səbəbə görə sistem proqramlaşdırma ilə məşğul olmaq istəyənlər Unixə üstünlük verirlər. Klassik Unix sistemlərindən isə(Solaris, AİX, HP_UX, FreeBSD ...) proqramlaşdırma bir kənara qalsın, adicə istifadə etmək peşəkar təcrübə tələb edir. Bu səbəblə GNU/Linux klassik Unix tipli sistem olmasına rəğmən nisbətən asan interfeysli olduğuna görə yeni başlayan(və peşəkar) sistem proqramçılar üçün ideal seçim hesab olunur. GNU/Linux sistmlərinin və assembler kompilyatorlarının quraşdırılması barədə ətraflı məlumatı *asadikhov.net* saytından və ya *youtube.com/user/heliks85* səhifəsindən əldə edə bilərsiniz .

Müəllif hüquqları:

Arzu, irad və təkliflərinizi müəllifə aşağıdakı ünvandan çatdıra bilərsiniz.

ahmed.sadikhov@gmail.com

İçindəkilər

1
Giriş

Assembler dili dedikdə bir çoxlarında köhnə proqramlaşdırma dili təsəvvürü yaranır ki, bu da səbəbsiz deyil. Həqiqətən assembler 3-cü səviyyəli dillərdən əvvəl istifadə olunan dil olub. Hal-hazırda 3-cü səviyyəli dillərin (Paskal,C,C++,C#, JAVA v.s.) standartları kifayət qədər inkişaf etdirilib, ixtisaslaşdırılmış proqramlaşdırma mühitləri və kitabxanaları yaradılmışdır. Bu səbəbdən bugünkü gündə ***"köhnə"*** hesab olunan bir dili örgənməyi hansı səbələr vacib eliyə bilər. Bu sualın çox cavabı ola bilər lakin mən tək bir cavabla assembleri örgənməyin vacibliyini ifadə etmək istəyirəm:

Assembler dili sistem proqramlaşdırmanın açarıdır.

Buradan o aydın olu ki, bugünkü gündə assembler yalnız sistem proqramlaşdırmanı örgənmək istəyənlər üçün vacib ola bilər və bundan da əlavə sistem proqramlaşdırmanı örgənmək istəyənlər assembler çox mükəmməl bilməlidirlər.

Bugünkü gündə assembler əsasən sistem proqramçılar tərəfindən istifadə olunur sistem proqram kodlarının tərtibində. Sistem proqramlaşdırmada istifadəsi geniş yayılmış digər dil əlbəttdə ki C dilidir. Assembler əsasən C ilə tərtib olunması mümkün olmayan kritik kod hissələrinin tərtibində istifadə olunur. Buraya əsasən qurğu və proqram kəsilmələrini, giriş – çıxış portlarını v.s. idarə edən kod hissələri aiddir.

Əməliyyatlar sistemi kəsilmələr vastəsilə idarə olunur. Bu kəsilmələrin hər biri sistemin normal fəaliyyəti üçün olduqca vacibdir. Misal üçün sistem üçün ən vacib kəsilmələrdən hesab olunan saat kəsilməsi zamanı (timer interrupt – əməliyyatlar sisteminin ürək döyüntüsü) icra olunan kod yalnız assemblerdə realizə oluna bilər. Aşağıda linux nüvəsinin 1.0 buraxılışından assembler dilində müvafiq kod kəsimi göstərilir(x86):

```
 .align 2
_timer_interrupt:
        push %ds                # save ds,es and put kernel data space
        push %es                # into them. %fs is used by _system_call
        push %fs
        pushl %edx              # we save %eax,%ecx,%edx as gcc doesn't
        pushl %ecx              # save those across function calls. %ebx
        pushl %ebx              # is saved as we use that in ret_sys_call
        pushl %eax
        movl $0x10,%eax
        mov %ax,%ds
        mov %ax,%es
        movl $0x17,%eax
        mov %ax,%fs
        incl _jiffies
        movb $0x20,%al          # EOI to interrupt controller #1
        outb %al,$0x20
        movl CS(%esp),%eax
        andl $3,%eax            # %eax is CPL (0 or 3, 0=supervisor)
        pushl %eax
        call _do_timer          # 'do_timer(long CPL)' does everything from
        addl $4,%esp            # task switching to accounting ...
        jmp ret_from_sys_call
```

Avadanlıqlar və onların proqramlaşdırılması sistem proqramlaşdırma məsələləridir. Biz bu məsələlərə yeri gəldikcə toxunacayıq. Ümumilikdə isə məqsəd assembler və sistemlə bağlı təməl bilikləri izah etməkdir.

Əməliyyatlar sisteminin idarə etdiyi ən vacib vahidlərdən biri istifadəçi proqramlarıdır. Ümumiyyətlə əməliyyatlar sisteminin əsas məqsədi istifadəçi proqramlarının normal iş fəaliyyətini təşkil etmək, proqramların kompüterin resurslarından istifadəsini təmin etmək, onların müraciətlərini düzgün yerinə yetirmək, istifadəçi proqrmlarına aid məlumatları mühafizə etməkdir. Bu deyilənlərdən kompüterdə icra olunan hər-bir proqramın əməliyyatlar sistemi üçün nə qədər vacib olduğu anlaşılır. Gəlin istifadəçi proqramlarının strukturu ilə tanış olaq.

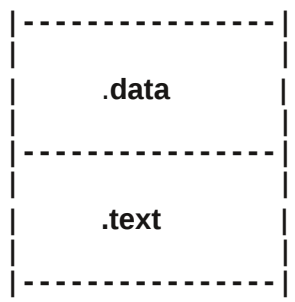
Proqramın strukturu

Kompüterdə icra olunan hər-bir proqram ***"hissə"*** adlandırılan struktur vahidlərindən ibarət olur. Yəni sadə dildə desək proqram müxtəlif məqsədlər üçün nəzərdə tutulmuş bir neçə ***hissə -*** dən ibarət olur. Əlbəttdə müasir proqramların hər-bir hissəsi və onların funksional xarakteristikası artıq təkmilləşdirilmiş sistem proqramlaşdırma kursunun mövzusudur. Biz isə hələlik başlanğıc səviyyədə proqramın ən vacib hissələr ilə tanış olacayıq.

Proqramın ən vacib hissəsi əlbəttdə ki onun instruksiyalarını özündə saxlayan ***.text*** hissəsidir. Proqramın digər vacib hissəsi proqram məlumatlarını özündə saxlayan **.data** hissəsidir. Proqramın digər vacib hissəsi **.stek** -dir, lakin steklə irəlidəki mövzularda tanış olacayıq.

Hələlik isə proqramı sadə halda aşağıdakı kimi iki hissədən ibarət təsəvvür edə bilərik, məlumatlar və instruksiyalar:

Proqramın sadə strukturu



Hissə adlarıının əvvəlinə (.**data, .text**) nöqtə qoyulduğuna diqqət yetirək. Assembler dilində əvvəli nöqtə ilə başlayan ifadələr ***"direktiv"*** adlanır. Direktivlər kompilyatora bu və ya digər məlumat ötürmək üçündür. Misal üçün **.data** direktivi kompilyatora məlumatlar hissəsinin başladığı yeri bildirir. **.text** direktivi isə instruksiyalar hissəsinin başlanğıcını bildirir. Digər direktivlərə misal olaraq **.globl**, **.bss** v.s. direktivləri göstərə bilərik. Bunlarla irəlidə tanış olacayıq.

Proqramda məlumatlar hissəsinin olması vacib deyil. Bu zaman **.data** direktivindən istifadə etmirik. Gəlin assembler dilində sadə proqram nümunəsi ilə tanış olaq.

Assembler dilində sadə proqram

Assembler dilində sadə proqram nümunəsi aşağıdakı kimi olar:

```
#assembler dilinde sade proqram

.data

.text

.globl _start
_start:

movl $5, %ebx

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

Proqramı sətir-sətir təhlil edək. Proqramın ilk sətri

```
.data
```

sətridir. Artıq bildiyimiz kimi burada **.data** direktivindən istifadə olunub və məlumatlar hissəsinin başlanğıcının elanını bildirir. Lakin hələlik məlumatlar hissəsində biz heç bir məlumat elən etməmişik. Ona görə proqramda biz bu direktivdən istifadə etməyə də bilərdik.

Növbəti sətir:

```
.text
```

Bu sətrin də mənası ilə biz artıq tanışıq, proqramın instruksiyalar hissəsinin başlandığı yeri bildirir.

Növbəti sətir:

```
.globl _start
```

Burada **.globl** direktivindən istifadə olunub. **.globl** direktivi kompilyatora əhəmiyyətli nişanlar barəsində məlumat verir. Burada kompilyatora **_start** nişanının əhəmiyyətli nişan olduğunu bildirilir.

Nöbvbəti sətir:

```
_start:
```

Assembler dilində sonu qoşanöqtə ilə bitən ifadələr ***"nişan"*** adlanır. Nişanlar hər-hansı məlumat və ya instruksiyanın balanğıc ünvanın bildirir və həmin ünvana istinad etmək üçün istifadə olunur. **_start** nişanı xüsusi nişandır və proqramın ilk icraolunmalı instrksiyasının yerini bildirir, başqa sözlə assembler proqramları **_start** nişanından icra olunmağa başlayır.

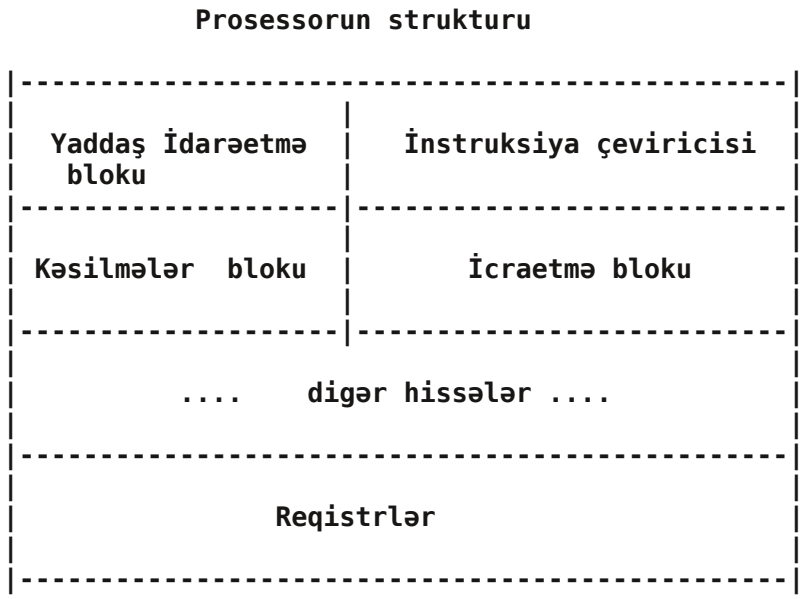
Proqramın növbəti 3 sətri aşağıdakı kimidir:

```
movl $5, %ebx

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

Bu sətirlər artıq prosessor tərəfindən icra olunan instruksiyalardır. İlk iki instruksiya müvafiq olaraq prosessorun **%ebx** və **%eax** reqistrlərinə **5** və **1** qiymətlərini yazır, sonuncu instruksiya isə **0x80** nömrəli kəsilməni çağırır. Bu instruksiyalarla ətraflı irəlidəki mövzularda tanış olacayıq.

Prosessorun strukturu.

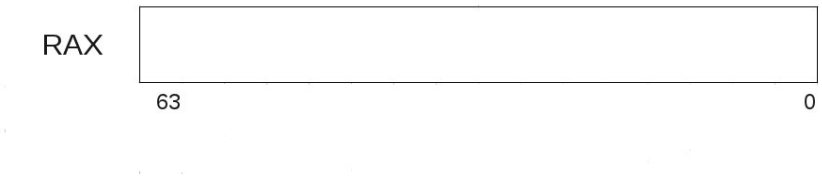
Baxmayaraq ki, prosessor olduqca mürəkkəb struktura malikdir və onu bütün incəliklərinə qədər izah eləmək üçün 10-larla kitab tələb olunur, yeni başlayan sistem proqramçılar üçün sadə halda prosessoru aşağıdakı kimi təsvir edə bilərik.



Bu hissələr içərisində bizim bilməli olduğumuz və istifadə edəcəyimiz hissə prosessorun Reqistrlərdir. Prosessorun digər hissələri ilə sistem proqramçılar məşğul olur.

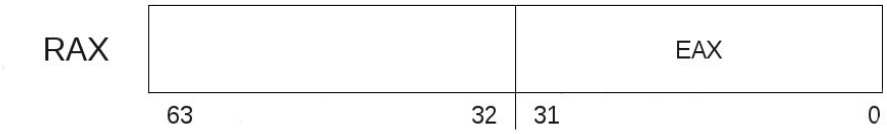
Registrlər prosessora aid çox kiçik yaddaş elementləridir. Prosessor kompüterin fiziki yaddaşından(RAM) məlumatı kiçik hissələrlə (4-8 bayt) reqistrlərə köçürür, sonra emal edir. Reqistrlərin hər-birinə öz adları ilə müraciət olunur.

Reqistrlərin ölçüsü və sayı konkret prosessor arxitekturasından asılı olur. x86 arxitekturasının əsas işçi reqistrləri **rax, rbx, rcx, rdx, rdi, rsi, rsp, rbp və rip** reqistrləridir. Bu reqistrlərdən **rax, rbx, rcx, rdx, rdi, rsi** məlumatlarla, **rsp, rbp** stek yaddaşı ilə işləmək üçün istifadə olunur. **rip** reqistri özündə icra olunan instruksiyanın ünvanın saxlayır. Bu reqistrlərin hər-birinin ölçüsü 8 baytdır (64 bit).

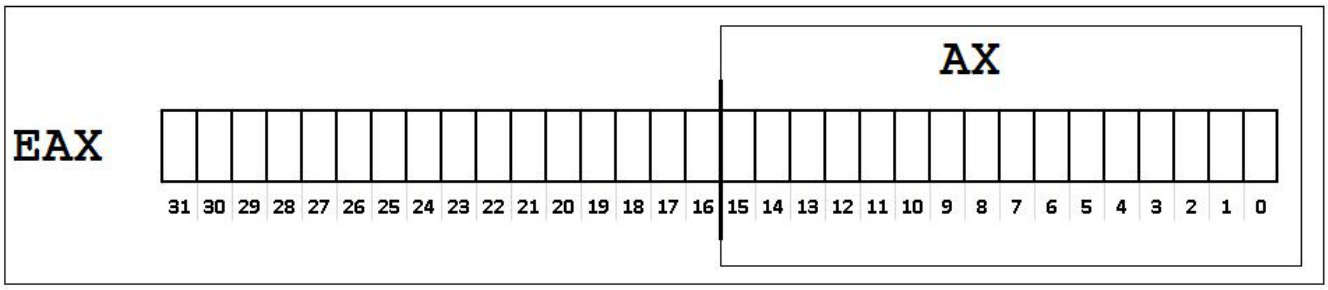


rax reqistri 64 bit.

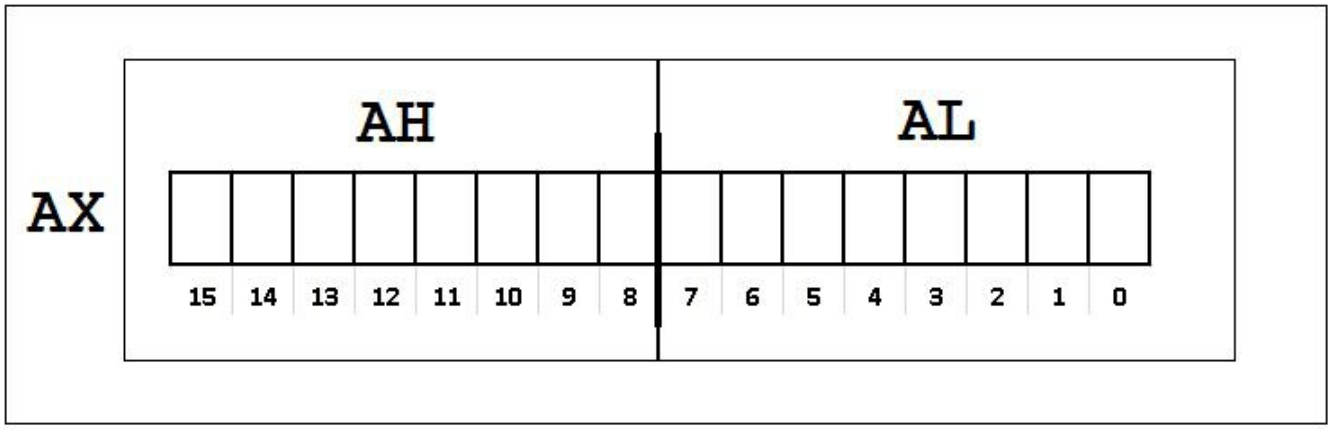
Bundan əlavə **rax, rbx, rcx, rdx** reqistrlərinin hər birinin ilk 32 bitinə uyğun olaraq **eax, ebx, ecx** və **edx** adı ilə müraciət etmək olar. 32 bitlik (4 baytlıq) əməliyyatlar zamanı bu reqistrlərlə işləmək daha əlverişlidir.



eax, ebx, ecx və **edx** reqistrlərin hər birinin də ilk 16 bitinə də **ax, bx, cx və dx** adları ilə müraciət etmək olar.



Öz növbəsində 1 baytlıq əməliyyatlar zamanı **ax, bx, cx və dx** reqistlərin hər birinin ilk 8 bitinə (0-7 bitlər) **al, bl, cl** və **dl**, növbəti 8 bitinə (8-15) isə **ah, bh, ch** və **dh** adları ilə müraciət etmək olar.



Qeyd. **rax, rbx, rcx, rdx, rdi, rsi, rsp, rbp və rip** reqistrləri 64 bitlik arxitekturalara aiddir, 32 bitlik maşınlarda bu reqistrlər mövcud deyil və **rax, rbx, rcx, rdx, rdi, rsi, rsp, rbp və rip** reqistrlərindən istifadə olunur.

Məlumatın köçürülməsi

Proqramın icrası zaman məlumat köçürmələri çox tez-tez baş verir. Müasir prosessorlar məlumat köçürmək üçün çox inkişafetmiş instruksiyalara malikdirlər. Bunlardan nisbətən mürəkkəbləri ilə yaddaş və ünvan mövzularını örgəndikdən sonra tanış olacayıq. İndi isə məlumat köçürmələrinin nisbətən sadə hallarını nəzərdən keçirəcəyik.

Məlumat köçürmək üçün **mov** instruksiyasından istifadə olunur. **mov** instruksiyasının sintaksisi aşağıdakı kimidir:

```
mov mənbə, mənsəb
```

mov instruksiyası məlumatı mənbədən mənsəbə köçürür. Köçürülən məlumatın ölçüsündən asılı olaraq **mov** instruksiyasının aşağıdakı formaları istifadə olunur: **movb**, **movw**, **movl** və **movq**. Bu instruksiyalar uyğun olaraq **1**, **2**, **4** və **8** bayt məlumat köçürmək üçün istifadə olunur. Misal üçün prosessorun **%eax** reqistrinə **1** qiyməti köçürən instruksiya aşağıdakı kimi olar:

```
movl $1, %eax
```

Assembler dilində reqistrlərin adının əvvəlinə faiz - **%** işarəsi, ədədlərin əvvəlinə isə dollar - **$** işarəsi artırılır. Burada biz **mov** instruksiyasının **movl** formasından istifadə etdik, çünki **%eax** reqistrinin ölçüsü **4** baytdır və **4** baytlıq əməliyyatlar zamanı **movl** -dən istifadə edirik.

Çalışma 1. Aşağıdakı instruksiyanın gördüyü işi müəyyənləşdirin:

```
movl $5, %ebx
```

Həlli: Prosessorun **%ebx** reqistrinə 5 qiyməti yazır.

Çalışma 2. Prosessorun **%edx** reqistrinə 7 qiymətini yazan instruksiya tərtib edin.

Həlli: Instruksiya aşağıdakı kimi olar:

```
movl $7, %ebx
```

Çalışma 3. **%eax** reqistrinə 1, **%ebx** reqistrinə 5, **%ecx** reqistrinə 20, **%edx** reqistrinə 30 qiyməti yazan proqram tərtib edin.

Həlli. Proqram kodu aşağıdakı kimi olacaq:

```
.data

.text

.globl _start

_start:

   movl $1,  %eax
   movl $5,  %ebx
   movl $20, %ecx
   movl $30, %edx

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

Proqramın sonuncu iki instruksiyasına nəzər salaq:

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

Bu iki instruksiya proqramı söndürmək üçündür.

Assembler proqramlarının kompilyasiyası

Assembler proqramlarını kompilyasiya etmək üçün əvvəlcə proqramın mətn faylını hər-hansı bir faylda, misal üçün **prg.s** faylında yadda saxlayırıq və kompilyasiya üçün aşağıdakı əmrləri daxil edirik:

```
as prg.s -o prg.o
ld prg.o -o prg
```

Nəticədə **prg.s** assembler faylından **prg** adlı icraolunabilən ikili proqram faylı alınır.

Assembler proqramlarının icrası

Assembler proqramlarını icra edərkən mütəmadi olaraq prosessorun reiqstrlərinin qiymətini,proqram yaddaşının müxtəlif hissəslərində (**stek, .data, .text , .bss** v.s.) məlumatları təhlil etmək tələb olunur. Bunun üçün müxtəlif təhlil proqramları mövcuddur. Biz hal-hazırda ən geniş yayılmış **gdb** təhlil proqramından istifadə edəcəyik.

gdb ilə proqramları təhlil etmək üçün aşağıdakı qaydalardan istifadə edirlər. Əvvəlcə proqram kodunun təhlil aparmaq istədiyimiz yerlərində nişan təyin edirik. Proqramı **gdb** ilə yükləyirlər. Hansı nişanlarda proqramın icrasını dayandırma istədiyimizi **gdb** -yə bildiririk və proqramın

icrasına başlayırıq. **gdb** proqramı bizim qeyd etdiyimiz ünvanlarda dayandırır və bizə prosessorun reqistrlərini və yaddaşı təhlil etməyə imkan verir. Təhlil işlərini bitirdikdən sonra proqramın icrasını davam etdiririk və proqram bizim təyin etdiyimiz növbəti nişanda icrasını dayandırır. Müvafiq əmrlər aşağıdakı kimi olar:

gdb ilə hər-hansı proqramı yükləmək üçün,

```
gdb proqramın_adı
```

əmrini, proqramın icrasını hər-hansı yerdə(nişan) dayandırmaq üçün,

```
break nişan
```

əmrini, reqistrlərin qiymətini örgənmək üçün,

```
info registers $reqisterin_adı
```

əmrini, növbəti instruksiyanı icra etmək üçün,

```
nexti
```

və ya qısaca

```
n
```

əmrini ,
Proqramın icrasını bərpa etmək üçün
continue

və ya qısaca

c

əmrini daxil etməliyik. **gdb** -nin yaddaşı təhlil etmə əmrləri ilə müvafiq bölmələrdə tanış olacayıq, hələlik isə yalnız reqistrlərin qiymətlərini yoxlamaqla kifayətlənəcəyik.

Çalışma 4. Aşağıdakı proqramın icraya başalama anında(_**start**) prosessorun reqistrlərinin qiymətlərini təyin edin.

```
#prg.s
.data

.text

.globl _start

_start:

   movl $1, %eax
```

```
        int  $0x80
```

Həlli. Əvvəlcə proqramın mətnini **prg.s** faylında yadda saxlayaq. Daha sonra

```
as prg.s -o prg.o
ld prg.o -o prg
```

əmrləri ilə **prg.s** -dən prg proqram faylını alırıq. **prg** -nı **gdb** ilə yükləyək:

```
gdb prg
```

Əvvəlcə proqramın icrasını dayandırmaq istədiyimiz yerləri **gdb** -yə bildirməliyik. Bizdən proqramın icraya başlama anında reqistrlərin qiymətlərini örgənmək tələb olunduğundan **_start** nişanından istifadə edə bilərik. Proqramın icrasını **_start** nişanında dayandırmaq üçün **break** əmrini daxil edək:

```
break _start
```

Bütün bunlar hazırlıq mərhələsidir. Proqramın icrasını başlaya bilərik, bunun üçün **run** əmrini daxil etməliyk.

```
run
```

Nəticədə proqram icra olunmağa başalayacaq və bizim təyin etdiyimiz yerlərdə gdb proqramın icrasını dayandıracaq (**_start** nişanı).

Bu anda prosessorun reqistrlərinin qiymətini örgənmək üçün aşağıdakı əmri daxil edirik:

```
      info registers $eax $ebx  $ecx  $edx  $esi  $edi
```

Nəticə:

```
(gdb) info registers $eax $ebx  $ecx  $edx  $esi  $edi
eax            0x0      0
ebx            0x0      0
ecx            0x0      0
edx            0x0      0
esi            0x0      0
edi            0x0      0
(gdb)
```

Çalışma 5. Prosessorun reqistrlərinə müxtəlif qiymətlər yazın. **gdb** proqramı ilə prosessorun reqistrlərinə yazılmış məlumatları yoxlayın.

Həlli. Proqram aşağıdakı kimi olar:

```
      #prg.s
     .data
```

```
     .text

     .globl _start

_start:

      movl $45, %eax
      movl $32, %ebx
      movl $9,  %ecx
      movl $12, %edx
    f:

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

Əvvəlcə proqramı kompilyasiya edək:

```
[ferid@fedora Documents]$ as tmp.s -o tmp.o
[ferid@fedora Documents]$ ld tmp.o -o tmp
```

Daha sonra proqramı **gdb** ilə yükləyək:

```
[ferid@fedora Documents]$
[ferid@fedora Documents]$ gdb tmp
GNU gdb (GDB) Fedora (7.2.90.20110429-36.fc15)
Reading symbols (no debugging symbols found)...done.
(gdb)
```

Proqramı **_start** və **f** nişanlarında dayandırmaq üçün **break** əmrindən istifadə edək:

```
(gdb)
(gdb) break _start
Breakpoint 1 at 0x400078
(gdb) break f
Breakpoint 2 at 0x40008c
(gdb)
```

Proqramın icrasını başlamaq üçün **run** əmrini daxil edək. Proqram icraya başlayacaq və **_start** nişanında dayanacaq.

```
(gdb) run
Starting program: /home/ferid/Documents/tmp

Breakpoint 1, 0x0000000000400078 in _start ()
(gdb)
```

Prosessorun **%eax, %ebx, %ecx** və **%edx** reqistrlərinin qiymətlərini yoxlayaq:

```
(gdb) info registers $eax $ebx  $ecx  $edx
eax            0x0      0
```

```
ebx            0x0      0
ecx            0x0      0
edx            0x0      0
(gdb)
```

Proqramın icrasını davam etsək, proqram f nişanında dayanacaq.

```
(gdb) c
Continuing.

Breakpoint 2, 0x000000000040008c in f ()
(gdb)
```

Reqistrlərin qiymətlərini yoxlayaq:

```
(gdb) info registers $eax $ebx  $ecx  $edx
eax            0x2d     45
ebx            0x20     32
ecx            0x9      9
edx            0xc      12
(gdb)
```

İzah: Proqramda biz **_start** və **f** adlı iki nişan təyin edirik. Proqram icraya başlayır və prosessorun reqistrlərinə müxtəlif qiymətlər yazır. Biz proqramı **gdb** ilə yükləyib tələb olunan yerlərdə proqramın icrasını dayandırmaq üçün **break** əmrindən istifadə edirik. Proqramın icrası **_start** nişanında dayandıqdan və biz reqistrlərin qiymətlərini yoxladıqdan sorna proqramın icrasını davam etmək üçün **c** əmrindən istifadə edirik. Proqram növbəti nişanda dayanır və biz reqistrlərin qiymətini yenidən yoxlayırıq.

Bəzi sadə instruksiyalar

Bu bölmədə bəzi sadə assembler instruksiyaları və onlardan istifadəyə aid proqram nümunələri daxil edəcəyik. Bu təcrübə bizə növbəti başlıqlarda tanış olacağımız prosessorun yaddaşa müraciət üsulları, stek və funksiyalar ilə işləyən zaman lazım olacaq.

ADD instruksiyası

Add instruksiyası iki məlumatı cəmləmək üçün istifadə olunur. Sintaksisi aşağıdakı kimidir:

```
    add məlumat1, məlumat2
```

Bu zaman məlumat1 məlumat2 -nin üzərinə əlavə olunur və nəticə məlumat2 -də saxlanılır. Qeyd edim ki, assembler dilində, ümumiyyətlə proqramlaşdırmanın aşağı səviyyəsində bütün məlumatlar ədədlərlə ifadə olunur, ikili, onluq və 16 -lıq formada. Bu barədə daha ətraflı 6-cı paraqrafda izah verilir.

Add instruksiyasına aid nümunələrə baxaq:

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

Birinci instruksiya **%eax** reqistrində olan məlumatı **%ebx** -dəki ilə cəmləyir və nəticəni **%ebx** -ə yazır, ikinci instruksiya **%ecx** reqistrinin qiymətin 56 vahid artırır.

Çalışma 5. Assembler dilində 23 ilə 5 ədədlərinin cəmini hesablayan proqram tərtib edin.

Həlli. Nümunə proqram aşağıdakı kimi olar:

```
#proq3.s

.text
.globl _start

_start:

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

add  %ecx, %ebx

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

İzahı: 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.

Çalışma 6. Çalışma 5 -də tərtib olunan proqramın nəticəsinin doğruluğunu yoxlayın.

Həlli. Proqram **ecx** -ə 23, **ebx** -ə 5 qiymətləri yazdıqdan sonra add instruksiyası ilə onların cəmini **ebx** -də saxlyır. Yəni **add %ecx, %ebx** instruksiysı icra olunn zaman **ebx** -ə 28 (23+5) qiyməti yazılmış olur. Proqramın nəticəsinin doğruluğunu yoxlamaq üçün **add %ecx, %ebx** instruksiyası icra olunduqdan sonra ebx -in qiymətinin 28 -ə bərabər olduğunu yoxlamalıyıq. Bunun üçün **add %ecx, %ebx** instruksiyasından sonra proqramın icrasını dayandırıb ebx reqistrinin qiymətini örgənməliyik. Proqramın icrasını **add %ecx, %ebx** instruksiyasından sonra dayandıra bilmək üçün proqram kodunda biraz dəyişiklik edək, tələb olunan instruksiyadan sonra yeni dayan adlı nişan yerləşdirək. Proqramın yeni kodu aşağıdakı kimi olar:

```
#proq3.s

.text
.globl _start

_start:

movl $23, %ecx
movl $5, %ebx
```

```
        add  %ecx, %ebx

        dayan:

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

Proqramı əvvəlcə kompilyasiya edək:

```
as prog3.s -o prog3.o
ld prog3.o -o prog3
```

Proqramı gdb ilə yükləyək:

```
 gdb prog3
```

dayan nişanında proqramın icrasını saxlamaq üçün **break dayan** əmrini daxil edək:

```
break dayan
```

Proqramı işə salmaq üçün **run** əmrini daxil edək və tələb olunan yerdə dayanmasını gözləyək:

```
run
```

Proqram icra olunancaq və dayan nişanında gdb proqramın icrasını dayandıracaq. Bu yerdə ebx reqistrinin qiymətini örgənmək üçün **info registers $ebx** əmrini daxil edək:

```
 info registers $ebx
```

Nəticə:

```
(gdb) info registers $ebx
ebx            0x1c     28
(gdb)
```

Sub instruksiyası

Sub - çıxma instruksiyasıdır. **sub** instruksiyası birinci arqumentin qiymətini ikincidən çıxıb nəticəni ikinciyə yerləşdirir.

Sub instruksiyasına aid nümunələrə baxaq:

```
sub %eax, %ebx
sub $5, %ecx
```

Birinci instruksiya **ebx** -dən **%eax** -i çıxır, ikinci instruksiya **ecx** -dən 5 çıxır.

Çalışma 7. Elə proqram tərtib edin ki, **eax** -ə 40 qiyməti yazsın, daha sonra onun qiymətini 35 vahid azaltsın.

Həlli. Müvafiq proqram kodu aşağıdakı kimi olar:

```
.text
.globl _start

_start:

# eax -e 40 qiymeti yazaq
movl $40, %eax

# eax -in qiymetini 35 vahid azaldaq
sub $35, %eax

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

Imul instruksiyası

Imul - vurma instruksiyasıdır. **Mull** instruksiyası iki arqument qəbul edir, birinci ilə ikinci arqumentin hasilini hesablayıb ikinciyə yerləşdirir.

Imul instruksiyasına aid nümunələrə baxaq:

```
imul %ecx, %edx
imul $5, %ebx
```

Birinci instruksiya **ecx** -i **%edx** -ə vurur, ikinci instruksiya **ebx** -in qiymətin 5 dəfə artırır.

Çalışma 8. Elə proqram tərtib edin ki, **eax** -ə 10, **ebx** -ə 5, qiyməti yazsın, daha sonra **eax** -lə **ebx** -in hasilini hesablasın.

Həlli. Müvafiq proqram kodu aşağıdakı kimi olar:

```
.text
.globl _start

_start:

# eax -e 10, ebx -e 5 qiymeti yazaq
movl $10, %eax
movl $5,  %ebx

# eax -le ebx -in hasilini hesablayaq
# cavab ebx -e yerleshir
```

```
imul %eax, %ebx

#cavabi (ebx) ecx -e köcurek
movl %ebx, %ecx

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

Div instruksiyası

Div - bölmə instruksiyasıdır. Qeyd edim ki, div instruksiyası toplama, çıxma və vurma instruksiyalarından bir qədər mürəkkəbdir. Buna görə **div** instruksiyasından istifadə etmənin sadə varintını təqdim edəcəm. Sadə halda **div** instruksiyasından istifadə etmək üçün əvvəlcə bölünəni **eax** -ə yerləşdiririk, **edx** -ə isə 0 qiyməti yazırıq. Daha sonra böləni hər-hansı başqa reqistrə yerləşdiririk. Bundan sonra bölməni yerinə yetirmək üçün **div bölən_reqistr** instruksiyasını icra edə bilərik. Nəticədə qismət **%eax**, qalıq isə **%edx** reqistrlərinə yerləşdirilir.

Çalışma 9. 20 -ni 4 ə bölən proqram kodu tərtib edin.

Həlli. Burada bölmək istədiyimiz ədəd 20(bölünən), böldüyümüz ədəd isə (bölən) 4-dür. Bunun üçün əvvəlcə edx -ə 0 qiyməti yazaq. Bölünəni eax, böləni isə hər-hansı (eax və edx -dən fərqli) reqistrə, misal üçün ecx reqistrinə köçürək.

```
movl $0,  $edx
movl $20, $eax
movl $4, %ecx
```

İndi bölməni yerinə yetirə bilərik. Bunun üçün **div %ecx** instruksiyasını icra etməliyik (çünki böləni ecx reqistrinə yerləşdirmişik).

```
div %ecx
```

Nəticədə qismət (5) **eax** -də, qalıq isə (0) **edx** -də olacaq. Yekun proqram kodu aşağıdakı kimi olar:

```
.text
.globl _start

_start:

movl $0,  %edx
movl $20, %eax

movl $4, %ecx

div %ecx

dayan:

movl $1, %eax
```

```
int  $0x80
```

İnc instruksiyası

inc instruksiyası operandının qiymətini 1 vahid artırır. incb, incw və incl formaları mövcuddur, hansı ki, uyğun olaraq 1,2 və 4 baytlıq operandlarla işləmək üçün istifadə olunur. Sintaksis aşağıdakı kimidir:

```
inc(b/w/l) operand
```

Misal üçün əgər eax reqistrinin qiymətini 1 vahid artırmaq istəsək onda

```
incl %eax
```

yazmalıyıq. eax reqistrinin ölçüsü 4 bayt olduğundan incl formasından istifadə etdik.

Başqa misallara baxaq:

```
incb %bh
```

bh reqistrinin qiymətini 1 vahid artırır(**ebx** -in ilk 8 biti).

Dec instruksiyası

dec instruksiyası operandının qiymətini 1 vahid azaldır. **decb**, **decw** və **decl** formaları mövcuddur, hansı ki, uyğun olaraq 1,2 və 4 baytlıq operandlarla işləmək üçün istifadə olunur. Sintaksis aşağıdakı kimidir:

```
decc(b/w/l) operand
```

Misal üçün əgər ecx reqistrinin qiymətini 1 vahid azaltmaq istəsək onda

```
decl %ecx
```

yazmalıyıq. ecx reqistrinin ölçüsü 4 bayt olduğundan decl formasından istifadə etdik.

Başqa misallara baxaq:

```
decw %dx
```

dx reqistrinin qiymətini 1 vahid azaldır(edx -in ilk 16 biti).

Çalışma 10. Aşağıdakı kod icra olunduqda ecx reqistrinin qiyməti neçə olar?

```
movl $6, %ecx
```

```
incl %ecx
incl %ecx
incl %ecx
incl %ecx
```

Həlli. Əvvəlcə **%ecx** -ə 6 qiyməti yazılır, sonra ardıcıl olaraq 4 dəfə **ecx** -in qiyməti 1 vahid artırılır. Nəticədə **ecx** -in qiyməti 10 -a bərabər olacaq.

Jmp - keçid instruksiyası

Assembler proqramı ***_start*** nişanından başlayaraq *"yuxarıdan – aşağı"* icra olunur. Bəzən isə proqramın icrasını növbəti instruksiyadan yox, *"başqa yerdən"* davam etdirmək tələb olunur. Bu zaman ***jmp*** *– keçid* instruksiyasından istifadə olunur.

jmp instruksiyasının sintaksisi aşağıdakı kimidir:

```
jmp nişan
```

Bu zaman icraolunma artıq *"növbəti"* instruksiyadan yox, ***jmp*** -instruksiyasında göstərilən *"nişan"* -dan davam edəcək.

Çalışma 11. Aşağıdakı kod icra olunduqda ***%ebx*** reqistrinin qiyməti neçə olar?

```
 movl $10, %ebx
 incl %ebx
 incl %ebx
 jmp a
 incl %ebx
 incl %ebx
a:
 decl %ebx
```

Həlli. Əvvəlcə **ebx** -ə 10 qiyməti yazılır. Daha sonra ardıcı olaraq 2 dəfə **ebx** -in qiyməti 1 vahid artırılır. Nəticədə **ebx** -in qiyməti 12 -yə bərabər olur. Daha sonra **jmp a** instruksiaysı icra olunur və **a** nişanına keçid baş verir, **jmp** instruksiyası ilə **a** nişanı arasında qalan instruksiyalar icra olunmur. a nişanında isə **decl %ebx** instruksiyası icra olunur və **%ebx** -in qiyməti 1 vahid azalır, cavab 11.

Çalışma 12. Aşağıdakı kod icra olunduqda ***%edx*** reqistrinin qiyməti neçə olar?

```
    movl $5, $edx
    decl %edx
    jmp a
    decl %edx
    decl %edx
a:
```

```
    jmp b
    movl $7, %edx
    incl %edx
b:
    movl $0, edx
```

Həlli. Əvvəlcə **edx** -ə 5 qiyməti yazılır. Daha sonra onun qiyməti 1 vahid azaldılır və **a** nişanına keçid baş verir. **a** nişanında isə **b** nişanına keçid instruksiyası yerləşdiyindən **b** -yə keçid baş verir. **b** nişanına **edx** -ə 0 qiyməti yazılır. Cavab 0.

Cmp - müqaisə instruksiyası

Yuxarıda biz proqramın icra istiqamətini dəyişmək üçün jmp – keçid instruksiyası ilə tanış olduq. Bu zaman jmp instruksiyası sadəcə istiqaməti bir yerdən başqa yerə yönəldirdi. Bəzən isə elə hallar ola bilər ki, bir yerdən başqa yerə keçid etmək hər-hansı şərtdən asılı olaraq yerinə yetirilsin. Bu zaman müqaisə və keçid instruksiyalarından birgə istifadə olunur, aşağıdakı kimi:

müqaisə qiymət1, qiymət2
şərti_keçid nişan

Bu zaman müqaisə instruksiyasına parametr kimi ötürülən qiymət1 ilə qiymət2 -nin müqaisəsindən asılı olaraq şərti_keçid instruksiyasında göstərilən nişana keçid yerinə yetirilir.

Müqaisə instruksiyası cmp kimi işarə olunur. Misal üçün tutaq ki, %eax reqistri ilə %ebx reqistrlərinin qiymətlərini müqaisə etmək istəsək yazarıq:

```
cmp %eax, %ebx
```

və ya %ecx reqistrinin qiymətini 4 ilə müqaisə etmək istəsək yazmalıyıq:

```
cmp %ecx, $4
```

Bir daha yada salaq ki, assemblerdə ədədlərin adları əvvəlinə dollar - $ işarəsi artırılır.

Qeyd edək ki, cmp instruksiyası sadəcə ona ötürülən parametrlərin qiymətlərini müqaisə edir və nəticə flags reqistrində qeydə alınır. Müqaisənin nəticəsindən asılı olaraq bu və ya digər əməliyyatı icra etmək üçün ***cmp*** instruksiyasından dərhal 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 cmp instruksiyasının nəticələrini uyğun olaraq aşağıdakı kimi nəzərə alır: keç əgər ikinci arqument birincidən böyükdürsə, böyük bərabərdirsə, kiçikdirsə, kiçik bərabərdirsə, bərabərdirsə, fərqlidirsə.

Çalışma 13. Aşağıdakı proqram icra olunduqda ***a*** nişanında ***%edx*** reqistrinin qiyməti neçə olar?

```
  .text

  .globl _start

 _start:

   movl $10, %edx
   movl $5, %eax
   cmp  $12, %eax
   jg  a
   incl %edx
a:

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

Həlli. Əvvəlcə %eax reqistrinə 5 qiyməti yazılır və 12 ilə müqaisə olunur. Cmp -in ikinci arqumenti birincidən böyük olmadığına görə və cmp -dən sonra jg (keç əgər ikinci birincidən böyükdür) instruksiyası icra olunduğuna görə tələb olunan şərt ödənmir və deməli jg -da göstərilən nişana keçid baş vermir. Proqram jg -dan sonra gələn instruksiyadan davam edir. Burada isə %edx -in qiyməti 1 vahid artırılır. Cavab 11.

Çalışma 14. Aşağıdakı proqram icra olunduqda a nişanında %edx reqistrinin qiyməti neçə olar?

```
  .text

  .globl _start

 _start:

   movl $10, %edx
   movl $25, %eax
   cmp  $12, %eax
   jg  a
   incl %edx
a:

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

Həlli. Baxdığımız bu halda isə ***%eax*** reqistrinin qiyməti 25 olduğundan və 25 12 -dən böyük olduğuna görə a nişanına keçid baş verir. Cavab 10.

Çalışma 13. ***eax*** və ***ebx*** reqistrlərinin qiymətlərini müqaisə et. Əgər ebx -in qiyməti ***eax*** -dən böyükdürsə ***ecx*** -ə 5 qiyməti yaz, əks halda ***ecx*** -ə 0 qiyməti yaz.

Həlli. **eax** və **ebx** reqistrlərinin qiymətlərini müqaisə etmək üçün **cmp %eax, %ebx** instruksiyasını icra etməliyik. Əgər **ebx eax** -dən böyükdürsə onda **ecx** -ə 5 qiyməti yazmalıyıq, əks halda 0. Bunun üçün aşağıdakı koddan istifadə edə bilərik.

```
   cmp %eax, %ebx
   jg  a
   movl $0, %ecx
   jmp b
a:
   movl $5, %ecx
b:
```

İzahı. Bu kod parçasında tələb olunan məsələnin həlli üçün biz müqaisə (**cmp**), şərti keçid(**jg**), şərtsiz keçid (**jmp**) və iki nişandan (**a, b**) istifadə etdik. Proses aşağıdakı kimi baş verir:

Əvvəlcə **cmp** instruksiyası **eax** -lə **ebx** -in qiymətini müqaisə edir. Dərhal sonra **jg a** instruksiyası gəlir. Əgər **ebx eax** -dən böyükdürsə onda **jg** instruksiyası **a** nişanına keçid edir və icraolunma **a** nişanından davam edir. **jg** -ilə **a** nişanı arasında qalan instruksiyalar (**movl $0, %ecx; jmp b**) icra olunmur. Keçid instruksiyalarının mahiyyəti budur.

a nişanında yerləşən instruksiya icra olunduqda **ecx** reqistrinə 5 qiyməti yazılır və beləliklə məsələnin birinci şərti təmin olunur.

Əks halda, yəni **ebx eax** -dən böyük olmazsa onda **jg** instruksiyası **a** nişanına keçid etmir və **jg** -dan sonra gələn instruksiyalar icra olunur. Bu zaman əvvəlcə **ecx** -ə 0 yazılır və **b** nişanına keçid edilir(**jmp b**). Burada əlavə **b** nişanı təyin etməyin və **ecx** -ə 0 yazdıqdan sonra **jmp** ilə həmin nişana keçməkdə məqsəd **a** nişanında olan instruksiyanı icra etməməkdir.

Əgər **ecx** -ə 0 yazdıqdan sonra jmp ilə **b** nişanına keçməsək onda **a** nişanında yerləşən instruksiya icra olunar və **ecx** -ə 5 qiyməti yazılar. Beləliklə **ebx** -lə **eax** -in müqaisəninin nəticəsindən asılı olmayaraq **ecx** -ə həmişə 5 qiyməti yazılar. İstifadə etdiyimiz bu yanaşma isə tələb olunan şərtlərin ödənməsini təmin edir.

Proqram kodun tam başa düşənə qədər təkrar-təkrar yazıb tədqiq etməyiniz məsləhətdir.

Çalışma 14. İki ədədin böyüyünü tapan proqram tərtib edin.

Həlli. Bunun üçün əvvəlcə müqaisə etmək istədiyimiz qiymətləri misal üçün **12** və **45** ədədlərinin uyğun olaraq **%eax** və **%ebx** reqistrlərinə köçürək. Müqaisənin nəticəsini isə **%ecx**-ə köçürərik. **cmp** instruksiyası ilə **%eax** -lə **%ebx** -in qiymətlərini müqaisə edib **jg** instruksiyası ilə ikincinin birincidən böyük olma halın yoxlayacağıq. Daha ətraflı məlumat aşağıda, proqramın izahında verilir. Proqram kodu aşağıdakı kimi olar:

```
        # 2 ededin boyuyunu tapan proqram

        .data

        .text

        .globl _start
        .type _start, @function


        _start:


        movl $12, %eax
        movl $45, %ebx

        # cmp ile ededleri muqaise edek
        cmp %eax, %ebx

        # müqaisənin nəticəsini yoxlamaq üçün
        # şərti keçid instruksiyalarından istifadə etməliyik
        # ikinci ededin birinciden boyukluyunu yoxlamaq ucun
        # jg instruksiyasindan istifade edek

        jg a

        # eger ikinci eded birinciden boyukdurse
        # onda keç a: nişanına ve z -e ebx -i yaz
        # eks halda yeni eax ebx -den boyukdurse (ve ya ber.)
        # onda eax -i kocur z -te ve son -a kec
        movl %eax, %ecx
        jmp son
a:
        movl %ebx, %ecx

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

Proqramın izahı:

Əvvəlcə **eax** və **ebx** reqistrlərinə müqaisə etmək istədiyimiz ədədləri köçürdük. Daha sonra cmp vastəsilə bu qiymətləri müqaisə etdik. Əgər **ebx eax** -dən böyük olarsa bu zaman **jg** instruksiyası **a** nişanına keçəcək və burada **ecx** -ə **ebx** -i yazacaq, əks halda (**eax** >= **ebx**) olarsa **jg** keçid etmir və proqramın icrası növbəti instruksiyadan davam edir. Burada **eax** -in qiyməti **ecx** -ə yazılır və **son** -a keçid edilir. Proqram sona çatır.

Dəyişənlər

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

```
nişan:
.tip
```

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 **long** tipli **x** və **y** adlı iki dəyişən elan etmək istəsək, aşağıdakı kimi yazmalıyıq

```
x:
.long

y:
.long
```

Bu zaman yaddaşda **x** və **y** 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

```
nişan:
.tip ilkin_qiymət
```

Misal üçün **int** tipli **z** dəyişəni elan edək və ona ilkin **34** qiyməti mənimsədək:

```
z:
.int 34
```

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

Başqa bir misala baxaq. **ascii** tipli **c** dəyişəni elan edək və ona ilkin olaraq **"A"** qiyməti mənimsədək:

```
c:
.ascii "A"
```

Çalışma 1. **long** tipli **x** dəyişəni elan edin. **x** dəyişəninə ilkin olaraq **45** qiyməti mənimsədin. **x** dəyişəninin qiymətini **%ebx** reqistrinə köçürün.

Həlli. Proqram kodu aşağıdakı kimi olar:

```
 .data

x:
.long 45

  .text

.globl _start

_start:

   movl x, %ebx

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

Çalışma 1. Dəyişənlərdən istifadə etməklə iki ədədin cəmini hesablayan proqram tərtib edin.

Həlli. Əvvəlki paraqrafda biz reqistrlərdən istifadə etməklə 2 ədədin cəmini hesablayan proqram tərtib etmişdik. Bu çalışmada biz əvvəlcə proqramın **.data** hissəsində **.long** tipindən olan **x, y** və adlı **z** dəyişən elan edəcəyik və onlaraq uyğun olaraq 10, 24 və 0 qiymətləri mənimsədəcəyik. Daha sonra **x** və **y**-in qiymətlərini toplamaq üçün əvvəlcə dəyişənlərdən reqistrlərə köçürəcəyik (**mov**), daha sonra onları toplayıb yekun qiyməti reqistrdən **z** dəyişəninə köçürəcəyik. Proqram kodu aşağıdakı kimi olar:

```
# assembler dilinde 2 ededin cemini
# hesablayan proqram
.data

x:
.long 10

y:
.long 24

z:
.long 0

.text
.globl _start
.type _start,@function

_start:

# x ve y -in qiymetlerini reqistrlere kocurek
movl x, %eax
movl y, %ebx
```

```
# qiymetleri cemleyek
addl %eax, %ebx

# neticeni z -te kocurek
movl %ebx, z

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

Proqramın nəticəsini yoxlamaq üçün ***son*** nişanında icranı dayandırmalı və ***print z*** əmri ilə ***z*** -in qiymətini öyrənmək olar, aşağıdakı kimi:

```
(gdb) print z
$1 = 34
(gdb)
```

Çalışma 2. Dəyişənlərdən istifadə etməklə iki ədədin ən böyüyünü hesablayan proqram tərtib edin.

Həlli. Biz reqistrlərdən istifadə etməklə bu proqramı tərtib etmişdik. İndi isə dəyişənlərdən istifadə etməklə eyni proqramı tərtib edəcəyik. Müvafiq kod aşağıdakı kimi oalr:

```
 .data

x:
.long 74

y:
.long 156

max:
.long 0

  .text

  .globl _start

_start:

   # evvelce deyishenlerin qiymetlerini
   # reqistrlere kocerek
   movl x, %eax
   movl y, %ebx

   # en boyuk qiymeti tapaq
   cmp %eax, %ebx
   jg  a
   movl %eax, max
   jmp b
```

```
 a:
   movl %ebx, max
 b:

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

Proqramı test etmək üçün kompilyasiya edib, gdb ilə yükləyirik və b nişanında dayanma təyin edirik(break b) və proqramı icra edirik(run). Proqram b nişanında icrasını dayandırdıqda print max əmri ilə max dəyişəninin qiymətini yoxlayırıq, aşağıdakı nümunədəki kimi:

```
[linux]$
[linux]$ as tmp.s -g -o tmp.o
[linux]$ ld tmp.o -g -o tmp
[linux $ gdb tmp
GNU gdb (GDB) Fedora (7.2.90.20110429-36.fc15)
Reading symbols ... done.
(gdb) break b
Breakpoint 1 at 0x4000d2: file tmp.s, line 34.
(gdb) run
Starting program:

Breakpoint 1, b () at tmp.s:34
34          movl $1, %eax
(gdb) print max
$1 = 156
(gdb) c
Continuing.
[Inferior 1 (process 28447) exited with code 0234]
(gdb) quit
[linux]$
```

x və y dəyişənlərinə hər dəfə müxtəlif qiymətlər verməklə proqramın nəticəsini test etmək olar.

İndi isə bir qədər mürəkkəb nümunə ilə tanış olaq.

Çalışma 3. Dəyişənlərdən istifadə etməklə 3 ədədin ən böyüyünü hesablayan proqram tərtib edin.

Həlli. Proqram kodu aşağıdakı kimi olar:

```
 .data

 x:
 .long 789

 y:
 .long 1291
```

```
z:
.long 455

max:
.long 0

  .text

  .globl _start

_start:

   # evvelce deyishenlerin qiymetlerini
   # reqistrlere kocerek
   movl x, %eax
   movl y, %ebx
   movl z, %ecx

   # en boyuk qiymeti tapaq
   cmp %eax, %ebx
   jg  a   # %ebx boyukdurse onu %ecx -le yoxla
   # eks hal, %eax -i %ecx -le muqaise et
   cmp %eax, %ecx
   jg c   # eger %ecx %eax -den boyukdurse onda o en bpyukdur
          # onu max -a kocurt, eks halda %eax en boyukdur onu
          # max -a kocurt
   movl %eax, max
   jmp son
 a:
   cmp %ecx, %ebx
   jg b  # eger %ebx %ecx-den de boyukdurse demek o en boyukdur

   # %ebx en boyukdur, onu max -a kocurt
 b:
   movl %ebx, max
   # ecx en boyukdur, onu max -a kocurt
   jmp son
 c:
   movl %ecx, max

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

Proqramı test etmək üçün ***son*** nişanında dayanma təyin edib max dəyişəninin qiymətini yoxlaya bilərik. ***x, y*** və ***z*** dəyişənlərinə hər dəfə fərqli qiymətlər verməklə proqrmın nəticəsini yoxlaya bilərik. Nümunə nəticə aşağıdakı kimi olar:

```
[linux]$
[linux]$ as tmp.s -g -o tmp.o
[linux]$ ld tmp.o -g -o tmp
[linux]$ gdb tmp
```

```
GNU gdb (GDB) Fedora (7.2.90.20110429-36.fc15)

Reading symbols from  ...done.
(gdb) break son
Breakpoint 1 at 0x4000ea: file tmp.s, line 52.
(gdb) run
Starting program:

Breakpoint 1, son () at tmp.s:52
52          movl $1, %eax
(gdb) print max
$1 = 1291
(gdb) c
Continuing.
[Inferior 1 (process 28725) exited with code 013]
(gdb) quit
[linux]$
```

Suallar:

1. Proqramın hissələrindən bəzilərinin adını sadalayın.

2. Proqramın instruksiyaları yerləşən hissəsi necə adlanır?

2. Proqramın məlumatları yerləşən hissəsi necə adlanır?

2. Direktivlər nə üçün istifadə olunur?

2. Hansı ifadələr nişan adlanır?

2. Nişanlar nə üçün istifadə olunur?

2. Aşağıdakı kodda hansı nişanlar təyin olunub?

```
  s:
    movl %eax, %ecx
  d:
    jmp s
  f:
    ret
 mx:
```

```
ff:
   int $126
```

10. Prosessorun təşkil olunduğu hissələrdən bəzilərinin adlarını sadalayın?

12. Reqistrlər prosessorun daxilində yerləşir, yoxsa yaddaşda ?

11. Reqistrlər nə üçün istifadə olunur?

10. x86 arxitekturalı prosessorların hansı reqistrlərini tanıyırsınız?

%rax reqistri -nin ölçüsü neçə bitdir?

%rax reqistrinin ilk 32 biti necə adlanır ? (Cavab %eax).

%eax reqistrinin ilk 16 biti necə adlanır?

%ax reqistrinin ilk və son 8 biti necə adlanır?

Aşağıdakı reqistrlərin ölçüləri neçə baytdır?
%rbx, **%eax**, **%cx**, **%dh**, **%al**.

3. Assembler dilində məlumat köçürmək üçün hansı instruksiyadan istiadə olunur?

4 baytlıq məlumat köçürmək üçün mov instruksiyasının hansı forması istifadə olunur?

2 baytlıq məlumat köçürmə zamanı mov instruksiyasının hansı forması istifadə olunur?

6. %eax reqistrində olan məlumatı **%ebx** reqistrinə köçürən instruksiya tərtib edin.

7. %ecx reqistrinin qiymətini 1 vahid atıran instruksiya tətib edin.

8. **%edx** və **%ecx** reqistrlərinin qiymətlərini cəmləyən instruksiya tərtib edin.

9. Aşağıdakı proqram icra olunduqda a nişanında **%ecx** reqistrinin qiyməti neçə olar?

```
        .data
        .text
        .globl _start

        _start:

        movl $2, %eax
        movl $5, %ebx

        add  %eax, %ebx
        movl %ebx, %ecx
a:

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

10. Assembler prqramlarını təhlil etmək üçün hansı proqramdan istifadə olunur?

11. break əmri nə məqsəd üçün istifadə olunur?

13. Hər-hansı nişanda proqramın icrasını dayandırmaq üçün nə etmək lazımdır?

14. %ecx reqistrinin qimətini örgənmək üçün hansı gdb əmrindən istifadə olunur.

Proqramın icrasını davam etmək üçün hansı əmrdən istifadə olunur?

15. 9-cu çalışmada daxil olunmuş proqramın a nişanında icrasını dayandırın və %ecx reqistrinin qimətini təyin edin.

16. Aşağıdakı proqram icra olunduqda b nişanında %edx reqistrinin qiyməti neçə olar. Cavabı izah edin, proqramı test edin.

```
        .text
        .globl _start

        _start:

        movl $0, %ecx
        movl $0, %edx
```

```
        movl $5, %ebx

a:
        cmp %ebx, %ecx
        jg  b
        addl $4, %edx
        incl %ecx

b:

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

17. Aşağıdakı proqram icra olunduqda b nişanında %edx reqistrinin qiyməti neçə olar. Cavabı izah edin, proqramı test edin.

```
        .text
        .globl _start

        _start:

        movl $0, %ecx
        movl $0, %edx
        movl $5, %ebx

a:
        cmp %ebx, %ecx
        jg  b
        addl $4, %edx
        incl %ecx
        jmp a

b:

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

1. Proqramda dəyişənin qiymətin örgənmək üçün hansı əmrdən istifadə olunur?

Çalışmalar.

1. %ebx reiqstrinə 3 qiyməti yazan proqram tərtib edin. Gdb ilə proqramı yükləyib, %ebx -in qiymətini test edin.

2. Yalnız reqistrlərdən istiadə etməklə 23, 45, 12 ədədlərini cəmləmək üçün proqram tərtib edin.

3. Yalnız reqistrlərdən istiadə etməklə 45 ilə 123 ədədlərinin böyüyünü tapan proqram tərtib edin.

4. Dəyişənlərdən və reqistrlərdən istiadə etməklə 4 və 67 ədədlərinin cəmini hesablayan proqram tərtib edin.

5. Dəyişənlərdən və reqistrlərdən istiadə etməklə 34 və 12 ədədlərinin böyüyünü tapan proqram tərtib edin.

6. Dəyişənlərdən və reqistrlərdən istiadə etməklə 56 , 67 və 89 ədədlərinin böyüyünü tapan proqram tərtib edin.

7. Dəyişənlərdən və reqistrlərdən istiadə etməklə 55, 32, 11 və 45 ədədlərinin böyüyünü tapan proqram tərtib edin.

8. 5 ilə 9 ədədlərinin hasilini hesablayan proqram tərtib edin.

9. 56 -ın 33 -ə bölünməsindən alınan qalığı hesablayan proqram tərtib edin.

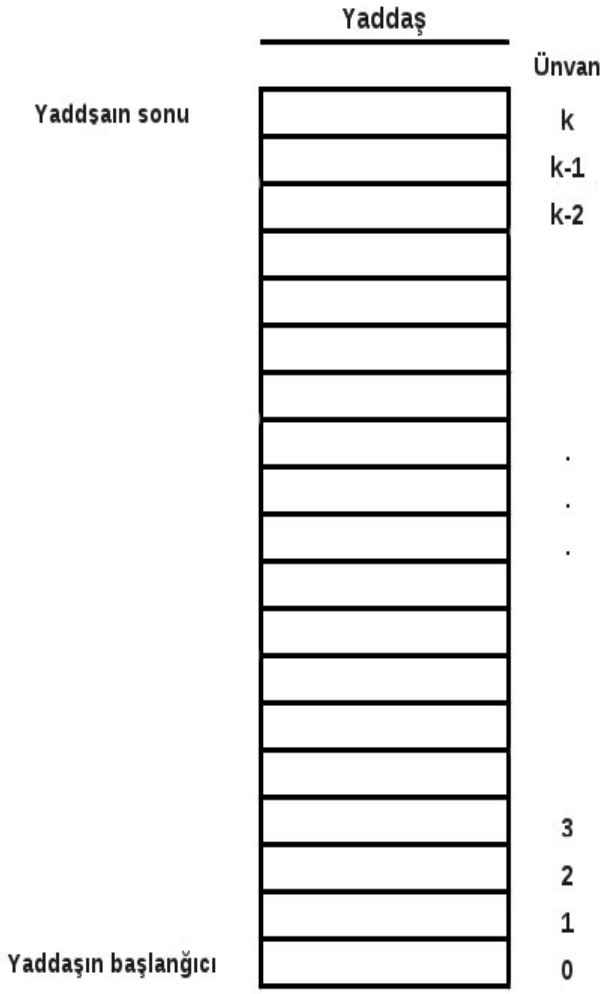
10. 456 -ın 23 -ə bölünməsindən alınan tam hissəni hesablayan proqram tərtib edin.

2
Yaddaş

Bu başlıqda assemblerin ən mürəkkəb və vacib mövzularından biri olan yaddaş ilə tanış olacayıq. Prosessorun yaddaşda yerləşən məlumatı əldə etmə yollarını örgənəcəyik. Bu başlıqda örgənəcəyimiz biliklər stekin və funksiyaların, eləcə də cərgələrin örgənilməsi zamanı bizə lazım olacaq. Bundan əlavə yaddaşın strukturunu, prosessorun yaddaşa müraciət qaydalarını mükəmməl bilmək əməliyyatlar sitemi və proqram qəzalarını təhlil etmək üçün mütləq vacibdir.

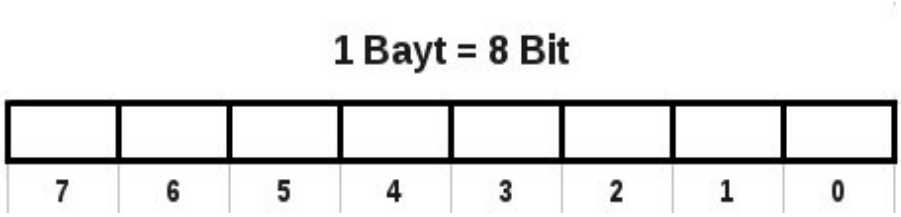
Yaddaşın strukturu

Kompüterin yaddaşını ardıcıl düzülmüş və 0 -dan başlayaraq nömrələnmiş kiçik yaddaş yuvaları şəklində təsəvvür etmək olar.

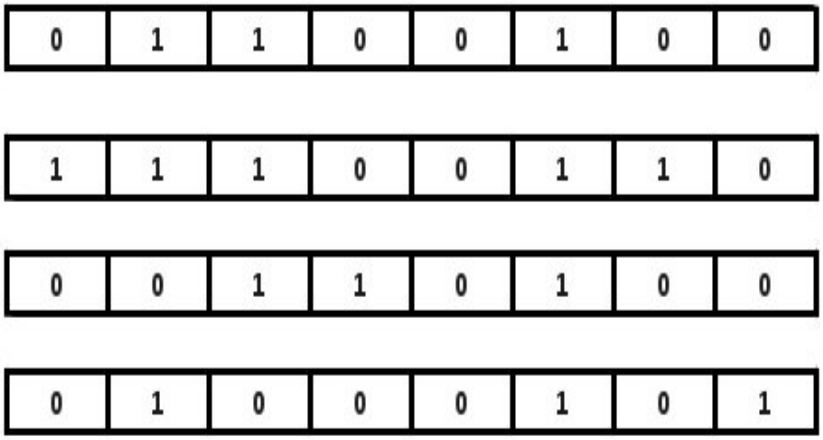


Sonuncu yaddaş yuvasının indeks nömrəsi(k) yaddaşın həcmi ilə müəyyən olunur. Hər-bir yaddaş yuvasının ölçüsü 1 baytdır və bu yaddaş yuvalarında yalnız və yalnız 0-dan 255 -ə kimi tam ədədlər yerləşdirmək olar.

Hər-bir bayt öz növbəsində 8 bitdən ibarətdir.



Bu bitlərin hər birində 0 və ya 1 qiyməti yerləşdirilə bilər, aşağıdakı kimi:



Bu bitlər ardıcıllığının hər-biri 0-dan 255 -ə kimi (hər ikisi daxil olmaqla) hər-hansı ədədə uyğun gəlir. Bu barədə daha ətraflı **Say sistemləri** bölməsində tanış olacayıq.

Buradan aydın olur ki, kompüterin yaddaşında ədədlərdən savayı heçnə yerləşdirmək olmaz. İstənilən tipli məlumat yaddaşda ədədlər ardıcıllığı şəklində yerləşir. Sonradan proqramlar həmin ədədlər ardıcıllığını istifadəçiyə tələb olunan şəkildə (musiqi, şəkil, mətn v.s.) göstərir.

Kompüter yaddaşında yerləşən məlumata müraciət etmək üçün onun ünvanın bilmək tələb olunur.

Ünvan

Hər bir yaddaş yuvasının ***indeks nömrəsi*** onun "***ünvanı"*** adlanır. İndeks nömrələri 0-dan başlayaraq nömrələndiyindən, yaddaşın ilk baytının ünvanı 0, növbəti baytının ünvanı 1 v.s. olar. Gördüyümüz kimi yaddaş ünvanları da öz növbəsində ədədlər vastəsilə ifadə olunur. Hər bir məlumat və instruksiya yaddaşda müəyyən bir ünvanda yerləşir. Dəyişənin ünavnın əldə etmək üçün onun adının əvvəlinə dollar - **'$'** işarəsi artırmaq lazımdır. Misal üçün aşağıdakı koda baxaq:

```
.data
```

```
x:
 .long 10
```

Bu kodda **x** dəyişəni elan olunub və onun yaddaş sahəsinə **10** qiyməti yazılıb. Əgər biz **x** dəyişəninin ünvanın əldə etmək istəyiriksə onda yuxarıda izah edildiyi kimi dəyişənin adının əvvəlinə **$** işarəsi artırmalıyıq, aşağıdakı kimi:

```
movl  $x, %ebx
```

x -in ünvanı **%ebx** -ə köçürülür.

Əgər **x** -in əvvəlindən **$** işarəsini götürsək onda **x**-in qiymətinə müraciət etmiş olarıq.

```
movl  x, %ebx
```

x -in qiyməti **%ebx** -ə köçürülür.

Çalışma 1. Sadə bir proqram tərtib edin və o proqramda bir dəyişən elan edin. Daha sonra həmin dəyişənin ünvanını **mov** instruksiyası ilə **%ebx** reqistrinə köçürün və **gdb** ilə bu ünvanı çap edin.

Həlli. Sadə proqram aşağıdakı kimi olar:

```
.data

x:
 .long 10

 .text
 .globl _start

_start:

# x nishaninin istinad etdiyi ünvani
# %ebx reqistrine kocurek

 movl $x, %ebx

 # %ebx reqistrinin qiymetini orgene bilmek
 # ucun son nishanindan istifade edek

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

Proqramı komilyasiya edib **gdb** ilə yükləyək və **son** nişanında proqramın icrasını dayandırıb, **%ebx** reqistrinin qiymətini örgənək.

Nəticə:

```
(gdb) info registers $ebx
ebx            0x6000bc 6291644
(gdb)
```

x nişanının ünvanı **6291644** və ya **0x6000bc** -dir. **info reqisters** əmri reqistrin qiymətini həm **16**-lıq(**hex**), həm də **10** -luq(**dec**) say sistemində göstərir. Say sistemləri ilə paraqraf 8-də tanış olacayıq. Ünvanlarla işləyərkən 16-lıq formadan istifadə etmək əlverişlidir.

Yuxarıda tanış olduğumuz məlumatlar yaddaş və ünvan barəsində təsəvvür yaratmağımıza kömək olur. Bu dərsin əsas məqsədi isə bununla yanaşı həm də prosessorun yaddaşa müraciət etmə üsulları ilə tanış olmaqdır. Prosessorun yaddaşa müraciət qaydasını yaxşı mənimsəmək assemblerin növbəti bölmələrini başa düşmək üçün vacibdir.

Prosessorun yaddaşa müraciət üsulu

Prosessor yaddaşdakı məlumatı əldə etmək üçün onun ünvanın bilməlidir. Ünvanı göstərmək üçün prosessorun təqdim elədiyi müxtəlif imkanlardan istifadə edə bilərik. Gəlin bu qaydalarla tanış olaq, sadədən mürəkkəbə doğru prinsipinə əməl edərək.

1) Üsul 1. Yəqin ki, məlumatın prosessora ötürmənin ən sadə üsulu məlumatın birbaşa instruksiyaya yerləşdirilməsi üsuludur. Bu üsul ilə biz artıq tanışıq, aşağıdakı kimi:

```
movl $45, %ebx
```

Yuxarıdakı kodda 45 qiyməti %ebx reqistrinə köçürülür. Bu zaman 45 qiyməti birbaşa **movl $45, %ebx** instruksiyasına yerləşdirilir, prosessor onu yaddaşın hansısa ünvanından əldə etmir. Bir şeyə diqqət yetirək ki, baxdığımız üsulda ədədin əvvəlinə dollar - $ işarəsi artırılıb. Növbəti üsullarla tanış olaq.

2) Üsul 2. Asandan mürəkkəbə getmə qaydasını nəzərə alsaq yəqin ki, məlumatı əldə eymənin ikinci üsulu nişandan istifadə etməkdir. Misal üçün tutaq ki, yaddaşda **long** tipli **y** dəyişəni elan etmişik və ona **3** qiyməti mənimsətmişik.

```
.data

y:
 .long 3
```

Bu zaman həmin məlumatı **%ebx** dəyişəninə köçürmək istəsək yazarıq:

```
movl y, %ebx
```

Nəticədə **%ebx** reqistrinə **y** dəyişəninin-in ***"qiyməti"*** - **3** yazılmış olaq. Bu dəfə artıq məlumayt prosessora yaddaşdan köçürülür. Dəyişənin adının əvvəlində **$** işarəsinin olmamasına diqqət yetirək.

3) Üsul 3. Məlumat əldə etmək üçün digər sadə üsul məlumatı bir reqistrdən digərinə köçürməkdir:

movl %eax, %ebx

Bu zaman prosessorun **%eax** reqistrində olan məlumat **%ebx** reiqstrinə köçürülər.

4) Üsul 4. Yuxarıdakı üsullar ilə biz əvvəlki dərslərimizdə istifadə etmişdik. Növbəti tanış olacağımız üsulda isə məlumata müraciət etmək üçün ünvandan istifadə edəcəyik. Tutaq ki, hər-hansı **x** dəyişənin ünvanı **%eax** reqistrinə yerləşdirilib. Bu zaman ünvana görə müraciət etmək üsulu ilə aşağıdakı kimi **%eax** ünvanında yerləşən məlumata müraciət edə bilərik.

movl (%eax), %ebx

Gördüyümüz kimi **%eax** reqistrini mötərizə içərisinə yerləşdirmişik. Bu instruksiya icra olunan zaman prosessor **%eax** reiqstrinə yerləşdirilmiş ünvandakı məlumatı **%ebx** -ə köçürər. Qeyd edim ki, əgər yuxarıdakı instruksiyada mötərizələrdən istifadə etməsəydik, onda **%ebx** -ə yaddaşdan hər-hansı məlumat yox, sadəcə **%eax** köçürülərdi(3-cü üsul).

Nümunəyə baxaq, tutaq ki hər-hansı **x** dəyişəni elan etmişik:

```
.data

x:
 .long 20
```

Əvvəlcə aşağıdakı kod ilə **x** dəyişəninin ünvanın **%eax** reqistrinə yazaq. Artıq bildiyimiz kimi dəyişənin ünvanın əldə etmək üçün onun adının əvvəlinə dollar - **$** işarəsi artırırıq.

```
movl $x, %eax
```

Daha sonra aşağıdakı instruksiya ilə ünvanı **%eax** -də yerləşən məlumatı **%ebx** -ə köçürə bilərik.

```
movl (%eax), %ebx
```

Nəticədə **%ebx** -ə **x** -in qiyməti – **20** köçürülmüş olar. Bu nümunə məlumata müraciət etmək üçün yaddaşdan istifadəyə nümunə üçün göstərilmişdir. Real proqramda biz **x** dəyişəninin qiymətini **%ebx** -ə köçürmək üçün onun ünvanın **%eax** -ə köçürüb, sonra ünvana görə müraciət üsulundan istifadə etmirik, elə birbaşa 2-ci üsuldan istifadə edərik, **movl x, %ebx**. Lakin steklə, cərgələrlə işləyərkən, funksiyalara parametr ötürərkən v.s. hallarda ünvana görə müraciət üsulu yaddaşdakı məlumatı əldə etmək üçün yeganə mümkün yol olur.

Yaddaşa müraciət üçün tanış olduğumuz bu yeni üsul aşağıda daxil edəcəyimiz ümumi üsulun bir halıdır. Bu ümumi üsul aşağıdakı kimidir.

Yaddaşda yerləşən məlumata müraciətin ümumi qaydası aşağıdakı kimidir:

```
Nişan( %sürüşmə_reqistri, %index_reiqstri, əmsal)
```

Bu qaydaya əsasən yekun ünvan aşağıdakı düstur ilə hesablanır:

```
Yekun_ünvan = Nişan + %sürüşmə_reqistri + %index_reqistri * əmsal;
```

Ünvanın göstərilmə qaydasında iştirak edən parametrlərdən istənilən biri buraxıla bilər. Bu zaman yekun ünvanın hesablanma düsturunda həmin parametrin qiyməti 0 götürülür. Əmsal yalnız **1,2,4** və **8** qiymətləri ala bilər.

İndi isə unvanın hesablanma qaydasına aid çalışmalara baxaq.

Tutaq ki, **x** nişanının ünvanı **10**, **%eax** reqistrinin qiyməti **5**, **%ebx** reqistrinin qiyməti **3** - dür. Aşağıdakı çalışmalarda yekun ünvanın hesablanmasına aid müxtəlif hallar üçün çalışmalar verilib.

Çalışma 2. **x(%eax, %ebx, 6)** -ifadəsinin istinad etdiyi yekun ünvanı hesablayın.

Həlli. Yekun ünvanın hesablanma düsturuna görə verilmiş ifadənin istinad elədiyi ünvanı aşağıdakı kimi hesablaya bilərik:

yekun ünvan = x + %eax + %ebx*6 = 10 + 5 + 3*6 = 33;

Cavab **33** - nömrəli bayt;

Çalışma 3. **x(%eax, %ebx, 10)** -ifadəsinin istinad elədiyi ünvanı hesablayın.

Həlli. Düstura əsasən unvanı aşağıdakı kimi hesablaya bilərik.

yekun ünvan = x + %eax + %ebx*10 = 10 + 5 + 3*10 = 45;

Cavab **45**.

Çalışma 4. x(%eax) -ifadəsinin istinad elədiyi ünvanı hesablayın.

Həlli. Bu dəfə **%index_reiqstri** və **əmsal** buraxıldığından düsturda onların qiyməti **0** götürülür.

yekun ünvan = x + %eax + 0*0 = 10 + 5 = 15

Cavab **15**.

Çalışma 5. **(%eax)** -ifadəsinin istinad elədiyi ünvanı hesablayın (bu hal artıq bizə tanışdır 4-cü üsul).

Həlli. Bu dəfə **nişan**, **%index_reiqstri** və **əmsal** buraxıldığından düsturda onların qiyməti **0** götürülür.

yekun ünvan = x + %eax + 0*0 = 0 + 10 + 0 = 10

Cavab **10**.

Çalışma 6. x(, %ebx, 8) -ifadəsinin istinad elədiyi ünvanı hesablayın.

Həlli. **%sürüşmə_reqistri** buraxıldığından onun qiyməti **0** götürülür. Düstura əsasən unvanı aşağıdakı kimi hesablaya bilərik.

yekun ünvan = x + 0 + %ebx*8 = 10 + 0 + 3*8 = 34;

Cavab 34.

Çalışma 7. Tutaq ki, **long** tipli **x** dəyişəni verilmişdir.

```
x:
 .long
```

Aşağıdakı koddan sonra **%ebx** reqistri yaddaşın hansı ünvanına istinad edəcək?

```
 movl $x, %ebx
```

Həlli. Dəyişənin adının əvvəlinə **$** işarəsi artıran zaman onun ünvanı qaytarılır. Buna görə yuxarıdakı kod icra olunan zaman **%ebx** reqistrində **x** -in ünvanı yerləşər, başqa sözlə **%ebx** reqistri yaddaşda **x** dəyişəninə istinad edər.

Çalışma 8. Tutaq ki, **long** tipli **x** dəyişəni verilmişdir.

```
x:
 .long
```

Aşağıdakı koddan sonra **%ebx** reqistri yaddaşın hansı ünvanına istinad edəcək?

```
 movl $x, %ebx
 incl %ebx
```

Həlli. İlk kod x dəyişənin ünvanını %ebx -ə köçürdüyündən %ebx x-ə istinad edir. Növbəti kod %ebx -in qiymətini 1 vahid artırdığından o yaddaşda x-dən sonrakı bayta istinad edir.

Yaddaşa müraciət üsullarını örgəndik, indi isə bu üsulların köməyi ilə yaddaş tədqiqi məsələləri ilə məşğul olaq.

Qeyd: yaddaşı tədqiq etmək üçün göstərilən aşağıdakı üsullar 64 bitlik maşınlar üçün

keçərlidir.

gdb proqramı ilə yaddaşın hər hansı ünvanında yerləşən məlumatı yoxlamaq üçün **x** əmrindən istifadə etməliyik. **x** əmrinin istifadə qaydası aşağıdakı kimidir:

x /SaySay_sistemiFormat Ünvan

Gördüyümüz kimi **x** əmri **4** patrametr qəbul edir: **Say**, **Say_sistemi**, **Format** və **Ünvan**.

Say parametri verilmiş ünvandan başlayaraq neçə hissə məlumatın çap edilməli olduğun bildirir.

Say_sistemi parametri məlumatın hansı say sistemində çap olunmalı olduğun göstərir. Onluq say sistemi üçün **d, 16** -lıq üçün **x** , ikili say sistemi üçün isə **t** qiymətindən istifadə edə bilərik.

Format parametri məlumatın neçə baytlıq hissələrlə çap edilməli olduğun bildirir. **1**- baytlıq hissələr üçün **b, 2** baytlıq hissələr üçün **h, 4** baytlıq hissələr üçün **w, 8** baytlıq hissələr üçün isə **g** qiymətindən istifadə etməliyik.

Bütün bunlarla daha ətraflı irəlidəki mövzularda tanış olacayıq. Hələlik isə sadə hallara baxaq.

Ünvan parametri məlumatlarını çap eləmək istədiyimiz yaddaş ünvanını bildirir.

Misal üçün yaddaşın **0xfffaa90** ünvanından başlayaraq növbəti bir bayt məlumatı **10** -luq say sistemində çap eləmək istəsək aşağıdakı əmri daxil etməliyik:

x /1db 0xfffaa90

Burada Say parametri 1(hissələrin sayı), Say_sistemi parametri d (10 -luq), Format parametri isə b (hər-bir hissənin ölçüsü 1 bayt) -dir.

Çalışma 7. Proqramda **long** tipli **s** dəyişəni elan edin və ona başlanğıc olaraq **47** qiyməti mənimsədin. **s** dəyişəninin ünvanın **%rbx** reqistrinə köçürün. Daha sonra həmin ünvanda olan məlumatı çap edin.

Həlli. Nümunə kod aşağıdakı kimi olar:

```
.data

s:
.long 47

.text
.globl _start

_start:
```

```
  movq $s, %rbx

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

Proqramı test edək:

```
[ferid@fedora Documents]$ gdb tmp
Reading symbols ...(no debugging symbols found)...done.
(gdb)
(gdb) break son
Breakpoint 1 at 0x4000b7
(gdb) run
Starting program:

Breakpoint 1, 0x00000000004000b7 in son ()
(gdb) info registers $rbx
rbx            0x6000c0 6291648

(gdb) x /1dw 0x6000c0
0x6000c0 <s>:   47
(gdb)
```

İzahı: Proqramda **long** tipli **s** dəyişəni elan edirik və ona **47** qiyməti mənimsədirik. Daha sonra **s** dəyişəninin yaddaşdakı ünvanın **%rbx** reqistrinə köçürürük. **info registers** əmri ilə **%rbx** -də olan məlumatı, **s** -in ünvanın çap edirik. **x** əmri ilə həmin ünvandan başlayaraq **4** bayt məlumatı (**long** tipi) 10 -luq say sistemində ekranda çap edirik.

Suallar.

1. Hər-bir yaddaş yuvasının ölçüsü neçə baytdır?
2. Yaddaş yuvaları bir-birinə nəzərən necə yerləşib?
3. Yaddaş yuvaları necə nömrələnir?
4. Ünvan nədir?

5. Yaddaşın ilk baytının ünvanı neçədir?

6. Dəyişənin ünvanın əldə etmək üçün onun adının əvvəlinə hansı simvol yazmaq lazımdır?

7. Məlumata müraciət etmənin ümumi qaydası necədir?

8. Yekun ünvanın hesablanması qaydası necədir?

3
Cərgələr

Bu başlıqda cərgələr ilə tanış olacayıq. Cərgələrin özəlliyi odur ki, onu təşkil edən elementlər yaddaşda ardıcıl düzülür və hər-biri eyni ölçüdə yer tutur. Bu imkan verir ki, cərgənin ilk elementinin və ya hər-hansı başqa elementinin ünvanını bilməklə digər elementlərin ünvanın əldə edə biləк.

Bu başlıqda əvvəlki başlıqda örgəndiyimiz yaddaşa müraciət etmək biliklərindən istifadə edəcəyik. Çalışma kimi təqdim olunan nümunə proqramlar həm yaddaşdan istifadə təcrübəsi, həm də ümumilikdə assembler proqramlaşdırma təcrübəsi toplamağa hesablanıb. Bu təcrübə növbəti bölmələrdə tanış olacağımız stek və funksiyalar kimi mürəkkəb assembler mövzularının mənimsənilməsində yardımçı olar.

Cərgənin elanı

Dəyişənlərlə tanış olarkən qeyd etdik ki, dəyişənə elan zamanı qiymət mənimsədərkən tipdən sonra həmin qiyməti yazmalıyıq, misal üçün aşağıdakı elanda **long** tipli **y** dəyişəninə başlanğıc **56** qiyməti mənimsədirik.

```
.data

y:
.long 56
```

Biz **56** rəqəmindən sonra vergüllə ayırmaqla istənilən sayda qiymət əlavə edə bilərik. Bu zaman həmin qiymətlər ardıcıllığından ibarət cərgə alarıq:

```
.data

y:
.long 56, 45, 7, 890, 21, 9
```

Başqa sözlə dəyişənlərə bir elementdən ibarət cərgə kimi baxmaq olar. Ümumilikdə isə cərgələr aşağıdakı kimi elan olunur: əvvəlcə cərgənin adı göstərilir, daha sonra tipi, daha sonra isə cərgənin elementləri vergüllə ayrılmaqla sıralanır.

```
cərgənin_adı:

  .tip element1, element2, … , elementn
```

Aşağıdakı çalışmalarda müxtəlif tiplərdən cərgələr elanına aid nümunələr göstərilir.

Çalışma 1. **long** tipindən olan **3, 4, 45, 56** elementlərindən ibarət **x** cərgəsi elan edin.

Həlli. Cərgənin elanı sintaksisinə əsasən **x** cərgəsini aşağıdakı kimi elan edə bilərik.

```
x:

.long   3,  4,  45,  56
```

Çalışma 2. **byte** tipindən olan özündə **'a', 'f', 'r', 'q'** və **'d'** simvollarını saxlayan **s** cərgəsi tərtib edin.

Həlli.

```
s:
.byte  'a', 'f', 'r',  'q', 'd'
```

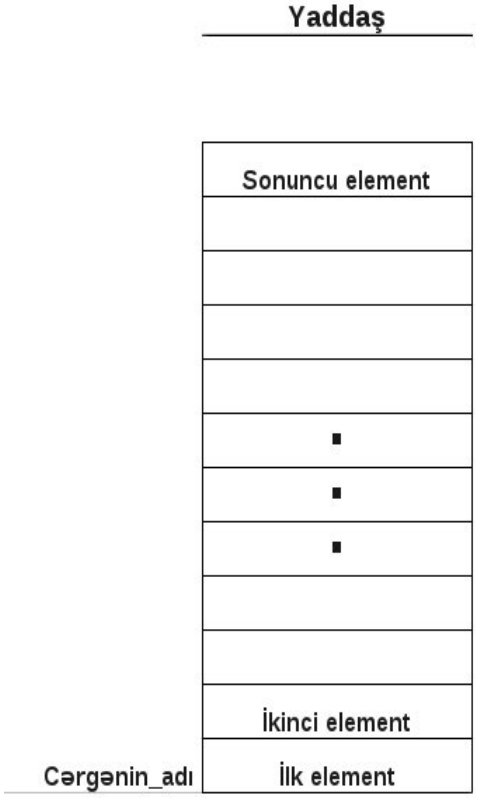
Qeyd . Yuxarıdakı elanda simvolların hamısı tipi **ascii** elan edərək cütdırnaq arasında da elan etmək olar. Aşağıdakı kimi:

```
s:
.ascii  "afrqd"
```

Bu zaman cütdırnaq arasında cərgənin elementləri ardıcıl sıralanır(arada vergül, məsafə olmadan). Bu qayda yalnız **ascii** tipindən olan cərgələrə, başqa sözlə sətirlərə aiddir.

Cərgənin yaddaşdakı vəziyyəti

Gəlin cərgələrin yaddaşda necə yerləşməsi və onun elementlərinin ünvanlarının necə müəyyən olunması ilə tanış olaq.



Gördüyümüz kimi Cərgənin adı cərgənin birinci elementinə istinad edir. Bütün elementlər yaddaşda eyni ölçüdə yer tutur və ardıcıl yerləşiblər. İndi elementlərin ünvanlarının necə müəyyən olunması ilə tanış olaq.

Tutaq ki, cərgənin ilk elementinin ünvanını bilirik, onu şərti olaraq **ÜNVAN** ilə işarə edək. Tutaq ki, cərgənin elementləri **long** tiplidir , yəni cərgənin hər bir elementi yaddaşda **4** bayt yer tutur. Bu zaman cərgənin ikinci elementi birincidən **4** bayt ***"yuxarıda"*** yerləşir. Başqa sözlə əgər birinci elementin ünvanı **ÜNVAN**-dırsa, onda ikinci elementin ünvanı **ÜNVAN + 4** olar. Analoji olaraq 3 -cü elementin ünvanı ikinci elementin ünvanından **4** vahid çox, yəni **ÜNVAN + 4 + 4** olar. Beləliklə cərgənin **k**-cı elementinin ünvanı **ÜNVAN + (k-1)*4** olar.
Bu dediklərimizdən cərgənin ilk elementinin ünvanı və ölçüsünü bildiyimiz zaman, cərgənin **n**-ci elementinin ünvanını əldə etmək üçün aşağıdakı düsturu almış oluruq:

n-ci_elementin_ünvanı = ilk_elementin_ünvanı + (n-1)*cərgənin_tipinin_ölçüsü

Nəzərə alsaq ki, cərgənin adı cərgənin ilk elementinə istinad edir, yuxarıdakı düsturu

aşağıdakı kimi yaza bilərik:

n-ci_elementin_ünvanı = cərgənin_adı + (n-1)*cərgənin_tipinin_ölçüsü

Bu deyilənlər cərgənin elementlərinin ünvanlarının təyin olunmasının mahiyyətini izah etmək üçündür, praktiki çalışmalar zamanı biz cərgənin bu və ya digər elementinə müraciət etmək üçün yaddaşa müraciət üsullarından istifadə edəcəyik. Əvvəlki bölmədə yaddaşa müraciət üçün daxil etdiyimiz qaydaya nəzər salaq:

Nişan(%sürüşmə_reqistri, %index_reiqstri, əmsal)

Qeyd etdiyimiz kimi yaddaşa müraciət üçün istifadə etdiyimiz bu ifadəyə görə ünvan aşağıdakı düstur ilə hesablanır:

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

Bu qaydadan istifadə etməklə cərgənin elementlərinə müraciət etmək istəsək aşağıdakı kimi yaza bilərik:

cərgənin_adı (, %index_reqistri, ölçü)

Burada **%index_reqistri** elementin indeks nömrəsi, **ölçü** isə cərgənin tipinin ölçüsünə uyğun gəlir.

Yuxarıdakı düsturuun uyğun gəldiyi ünvanı hesablayaq:

Yekun_ünvan = cərgənin_adı + %index_reqistri * ölçü

(Burada **%sürüşmə_reqistri** iştirak etmədiyindən 0 götürülmüşdür.) Gördüyümüz kimi əgər **%index_reqistri** -nə cərgənin **n**-ci elementinin indeks nömrəsini, ölçü parametrinə isə cərgənin tipinin ölçüsünü mənimsətsək onda bu ünvan cərgənin n-ci elementinin ünvanı ilə üst-üstə düşür. Beləliklə biz cərgənin adı, tipinin ölçüsündən istifadə etməklə istənilən elementinə müraciət edə bilərik.

Calışma . **long** tipli **5** elementdən ibarət **x** adlı cərgə elan edin. **x** cərgəsinin **3** -cü elementinin qiymətini çap edin.

Həlli. Əvvəlcə test üçün istifadə edəcəyimiz proqram kodunu daxil edək. Daha sonra test proseduru ilə tanış olaq. Sonda izaha nəzər salarıq.

Nümunə kod:

```
.data

# long tipli 5 elementli x cergesi elan edek
x:
.long 34, 768, 89, 33, 10
```

```
 .text
 .globl _start

 _start:

 # 3 -cu elementin indeks nomresi 2 -dir
 # indeks nomreleri 0-dan hesablanir
 # indeks nomresini %ecx -e kocurek

 movl $2, %ecx

 # long tipinin olcusu 4 baytdir
 # 3 -cu elementi %ebx reqistrine kocurek

 movl x(,%ecx,4), %ebx

 #yuxaridaki instrusiya x cergesinin 3-cu
 # elementini %ebx -e kocurer, gdb ile
 # onun qiymetini yoxlaya bilerik
  son:
    movl $1, %eax
    int  $0x80



 Test:


[ferid@fedora Documents]$ gdb tmp

Reading symbols  ...done.
(gdb) break son
Breakpoint 1 at 0x4000bd
(gdb) run
Starting program:

Breakpoint 1, 0x00000000004000bd in son ()
(gdb) info registers $ebx
ebx            0x59     89
(gdb)
```

Izahı: Proqramın data hissəsində long tipli **5** elementdən ibarət **x** cərgəsi elan edirik və bu cərgənin elementlərinə müftəlif qiymətlər mənimsədirik. Qiymətini çap etmək istədiyimiz element 3 -cü element olduğundan onun indeks nömrəsini **%ecx** reqistrinə yazırıq. Cərgənin elementlərinin indeks nömrələri 0-dan başlayaraq nömrələndiyindən 3-cü elementin indeks nömrəsi 2 olur (birinci elementin indeksi 0-dır). Daha sonra **movl x(,%ecx,4), %ebx** instruksiyası ilə **x** cərgəsinin 3-cü elementini **%ebx** -ə yazırıq. Proqramı test etmək üçün onu gdb ilə yükkləyib **son** nişanında icrasını dayandırırıq və **%ebx** reqistrinin qiymətini yoxlayırıq.

Çalışma. 8 **ascii** simvolundan ibarət **s** cərgəsi elan edin. **s** cərgəsinin **5**-ci elementini çap

edin.

Həlli: Əvvəlcə nümunə kodu daxil edək.

```
 .data

 s:
 .ascii "akdfgrty"

 .text
 .globl _start

 _start:

 # 5 -ci elementin indeks nomresi 4 -dur
 # indeks nomreleri 0-dan hesablanir
 # indeks nomresini %ecx -e kocurek

 movl $4, %ecx

 # ascii tipinin olcusu 1 baytdir
 # 5 -cu elementi %bh reqistrine kocurek

 movb s(,%ecx,1), %bh

 #yuxaridaki instrusiya s cergesinin 5-ci
 # elementini %bh -e kocurer, gdb ile
 # onun qiymetini yoxlaya bilerik
  son:
    movl $1, %eax
    int  $0x80
```

Proqramı test edək:

```
(gdb) run
Starting program:

Breakpoint 1, son () at tmp.s:31
31          movl $1, %eax
(gdb) info registers $rbx
rbx            0x6700   26368
(gdb)
rbx            0x6700   26368
(gdb) info registers $bh
bh             0x67     103
(gdb)
```

İzahı: Proqramın data hissəsində biz **ascii** tipli **8** elementdən ibarət **s** cərgəsi elan edirik. Qeyd edək ki **ascii** tipinin ölçüsü **1** baytdır. Daha sonra **5**-ci elementə mütraciət etmək üçün əvvəlcə onun indeks nömrəsini (**4) %ecx** -ə yazırıq. Aşağıdakı instruksiya **s** cərgəsinin **5**-ci

elementini **%bh** reqistrinə yazır.

movb s(,%ecx,1), %bh

Burada biz **s(,%ecx,1)** ünvanında yerləşən bir bayt məlumatı köçürmək istədiyimizə görə **movb** iinstruksiyasından istifadə etmişik. Yada salaq ki (1-ci başlıq) **%bh** reqistrinin ölçüsü 1 baytdır. **Info registers** əmri ilə **%bh** reqistrinin qiymətini çap etdikdə aşağıdakı nəticəni alırıq:

```
(gdb) info registers $bh
bh             0x67     103
(gdb)
```

Burada **%bh** reqistrinin qiyməti **103** göstərilmişdir. Bu **ASCİİ** cədvəlində **'g'** simvoluna uyğun gəlir(ASCİİ cədvəli simvolları Əlavə fix -də verilmidir).

Calışma. **long** tipli **7** elementdən ibarət **y** cərgəsinin elementləri cəmini hesablayan proqram tərtib edin.

Həlli:

```
 .data

 y:
 .long 15, 18, 1, 45, 78, 243, 89, 10

 cem:
 .long 0

 .text
 .globl _start

 _start:


  movl $0, %edx
dovr:
  movl y(,%edx,4), %eax
  addl %eax, cem
  incl %edx
  cmpl $7, %edx
  je son
  jmp dovr

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

Proqramı test edək:

```
[ferid@fedora Documents]$ gdb tmp

(gdb) break son
Breakpoint 1 at 0x4000cd

(gdb) run
Starting program:

Breakpoint 1, 0x00000000004000cd in son ()
(gdb) print cem
$1 = 489
(gdb)
```

Proqramın izahı:

Əvvəlcə proqramda **7** elementdən ibarət **y** cərgəsi elan edirik. Daha sonra **%edx** reqistrinə **0** qiyməti yazırıq. **%edx** reqistrində biz hal-hazıra kimi nəzərdə keçirdiyimiz elementlərin sayını yadda saxlayırıq. Daha sonra dovr nişanı elan edirik. **movl y(,%edx,4), %eax** instruksiyası **y** cərgəsinin **%edx** indeksli elementini **%eax** reqistrinə köçürür. cem dəyişəninə elanda **0** qiyməti mənimsətmişik. Dövr hər-dəfə təkrar olunduqda cərgənin növbəti elementi **%eax** reqistrinə köçürülür və **addl %eax**, cem instruksiyası ilə həmin qiymət cem -in üzərinə əlavə olunur. Daha sonra **%edx** -in qiymətini bir vahid artırırıq(**incl %edx**) və yeni qiyməti **7** ilə müqaisə edirik(**cmpl $7, %edx**). **%edx 7** qiyməti alanda artıq bütün elementlər nəzrədən keçirilib və biz dövrdən çıxırıq(**je son**). Əks halda dovrün əvvəlinə qayıdırıq(**jmp dovr**).

Çalışma 3. Tutaq ki, **y** adlı hər-hansı cərgə verilmişdir. Məlumdur ki, bu cərgənin sonuncu elementi **453** -ə bərabərdir. Cərgənin elementlərinin sayını tapan proqram tərtib edin.

Həlli. Əvvəlcə proqramı daxil edək, daha sonra proqramın ətraflı izahını verərik.

```
 # cergenin elementlerinin sayini tapan proqram
 .data

 y:
  .long 45, 23, 67, 2, 12, 78, 90, 453

  .text

   .globl _start

   _start:

   #ilk olaraq indeks = 0 goturek
   movl $0, %edx

   #dovre bashliyiriq
dovr:
   #novbeti indeksi %edx olan elementi
```

```
   #%eax -e kocurek
   movl y(, %edx, 4), %eax

   #muqaise edek
   cmpl $453, %eax

   #beraberdirse son
   je son

   #eks halda indeksi 1 vahid artir
   incl %edx

   #qayit evvele
   jmp dovr

son:
   # say = indeks + 1
   incl %edx

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

Izahi: əvvəlcə **%edx** reqistrinə **0** qiyməti yazırıq. Cərgənin elementlərinə müraciət etmək üçün **y(, %edx, 4)** yaddaşa müraciət üsulundan istifadə edirik. Bu zaman göstərilən ifadə indeks nömrəsi edx olan elementin ünvanına istinad edəcək. İlk elementdən başalayaraq (index nömrəsi **0**) elementləri **%eax** reqistrinə köçürürük. Daha sonra **%eax** -in qiymətini **453** ədədi ilə müqaisə edirik. Əgər bərabərdisə onda deməli cərgənin sonuna çatmışıq dövrü tərk edirik, əks halda indeksi bir vahid artırıb dövrün əvvəlinə qayıdırıq.

Çalışma 4. Tutaq ki, **long** tipli **q** cərgəsi verilmişdir. Cərgədəki elementlərin sayının **8** olduğu məlumdur. Cərgənin elementləri içərisində qiyməti **56**-ya bərabər olan elementin indeks nömrəsini tapan proqram tərtib edin.

Həlli. Əvvəlcə proqram kodun daxil edək, sonra izahla tanış olarıq.

```
 # cergedeki elementin indeksini tapan proqram
 .data

 q:
  .long 234, 3, 90, 78, 56, 67, 19, 83

  .text

   .globl _start

   _start:

   #ilk olaraq indeks = 0 goturek
   movl $0, %edx

   #dovre bashliyiriq
dovr:
   #novbeti indeksi %edx olan elementi
   #%eax -e kocurek
```

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

    #muqaise edek
    cmpl $56, %eax

    #beraberdirse son
    je son

    #eks halda indeksi 1 vahid artir
    incl %edx

    #eger indeks = 8 demeli cergenin sonudur dovru terk et
    cmpl $8, %edx
    je son

    #qayit evvele
    jmp dovr

son:

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

Nəticə:

```
(gdb) run
Starting program

Breakpoint 1, 0x00000000004000cb in son ()
(gdb) info registers $edx
edx            0x4      4
(gdb)
```

İzahı: Əvvəlcə **%edx** reqistrinə **0** qiyməti yazırıq və ilk elementdən başlayaraq bir-bir bütün elementləri yoxlayırıq. Hər-dəfə **%edx** -in qiymətini **1** vahid artırırıq. **56**-ya bərabər olan element tapılanda dövrü tərk edirik. Dövrün sonun yoxlamaq üçün **%edx** -in qiymətini **8** ilə müqaisə edirik.

Çalışma 5. Elementlərinin tipi **char** olan **f** cərgəsi verilmişdir. **'s'** -ə bərabər olan elementin indeksini tapın. Cərgənin sonuncu elementi **'a'** -ya bərabərdir.

```
 # cergedeki elementin indeksini tapan proqram
 .data

 f:
  .byte 'd', 'q', 't', '+', '?', 's', 'w', 'a'

  .text

   .globl _start

   _start:
```

```
   #ilk olaraq indeks = 0 goturek
   movl $0, %ecx

   #dovre bashliyiriq
dovr:
   #novbeti indeksi %ecx olan elementi
   #%al -e kocurek
   movb f(, %ecx, 1), %al

   #muqaise edek
   cmpb $'s', %al

   #beraberdirse son
   je son

   #eks halda indeksi 1 vahid artir
   incl %ecx

   #qayit evvele
   jmp dovr

son:

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

Nəticə:

```
(gdb) run
Starting program

Breakpoint 1, 0x00000000004000ca in son ()
(gdb) info registers $ecx
ecx            0x5      5
(gdb)
```

Artıq yaddaşa müraciət üsulları, keçid və müqaisə instruksiyalarının tətbiqi ilə assembler dilində daha mürəkkəb proqramlar tərtib edə bilərik.

Çalışma 3. **241, 15, 242, 123, 50, 100, 240** elementlərindən ibarət **eded_ard** cərgəsinin ən böyük elementini tapan proqram tərtib edin.

```
        # en böyek qiymeti tapan proqram
        # proqram en boyuk qiymeti tapib
        #

        .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(j**mp 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 həddburaxıldığından 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.

Suallar.

1. Cərgənin elementləri bir birinə nəzərn yaddaşda necə yerləşir ?

2. Cərgənin ilk elementinin indeksi neçədir ?

3. Tutaq ki, ***long*** tipli ***x*** cərgəsi verilib

```
x:
 .long  5, 34, 98, 78, 435
```

aşağıdakı koddan sonra **%ebx** reqistrinin qiyməti neçə olar?

```
movl x, %ebx
```

4. Tutaq ki, ***long*** tipli ***x*** cərgəsi verilib

```
x:
 .long  5, 34, 98, 78, 435
```

aşağıdakı koddan sonra **%ebx** reqistrinin qiyməti neçə olar?

```
movl $0, %edx
movl x(,%edx,4), %ebx
```

5. Tutaq ki, ***long*** tipli ***x*** cərgəsi verilib

```
x:
 .long  5, 34, 98, 78, 435
```

aşağıdakı koddan sonra **%ebx** reqistrinin qiyməti neçə olar?

```
movl $3, %edx
movl x(,%edx,4), %ebx
```

6. Tutaq ki, ***long*** tipli ***x*** cərgəsi verilib

```
x:
 .long  5, 34, 98, 78, 435
```

aşağıdakı koddan sonra **%ebx** reqistrinin qiyməti neçə olar?

```
movl $x, %eax
movl (%eax), %ebx
```

7. Tutaq ki, ***long*** tipli ***x*** cərgəsi verilib

```
x:
 .long  5, 34, 98, 78, 435
```

aşağıdakı koddan sonra **%ebx** reqistrinin qiyməti neçə olar?

```
movl $x, %eax
addl $4, %eax
movl (%eax), %ebx
```

Çalışmalar.

1. long tipli cərgənin 5 -ci elementini təyin edən proqram tərtib edin.

2. ascii tipli cərgənin 3 -cü elementini təyin edən proqram tərtib edin.

3. long tipli 8 elementdən ibarət cərgənin ən kiçik elementini təyin edən proqram tərtib edin.

4. ascii tipli cərgənin sonuncu elementi 'h' simvoludur. Cərgənin 'a' -ya bərabər olan simvollarının sayını təyin edən proqram qurun.

5. long tipli cərgənin sonuncu elementi 43 -ə bərabərdir. Cərgənin cüt elementlərinin cəmini tapan proqram tərtib edin.

Yaddaş tədqiqi

long tipli **x** cərgəsi elan edək. **x** -in ünvanın örgənək və yaddaşı tədqiq etmə metodları ilə verilmiş ünvanda yerləşən məlumatları çap edək.

Proqram kodu aşağıdakı kimi olar:

```
.data
```

```
 x:
  .long  5, 34, 98, 78, 435, 43, 678, 2, 4545, 234, 678, 0, 321, 789

  .text

   .globl _start

   _start:

  #cergenin unvanin %ebx -e kocurek
  movl $x, %ebx

son:

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

Test edək:

```
(gdb) break son
Breakpoint 1 at 0x4000b5
(gdb) run
Starting program:

Breakpoint 1, 0x00000000004000b5 in son ()
(gdb) info registers $rbx
rbx            0x6000bc 6291644
(gdb) x /10dw 6291644
0x6000bc <x>:   5       34      98      78
0x6000cc <x+16>:        435     43      678     2
0x6000dc <x+32>:        4545    234
(gdb)
```

4
Stek

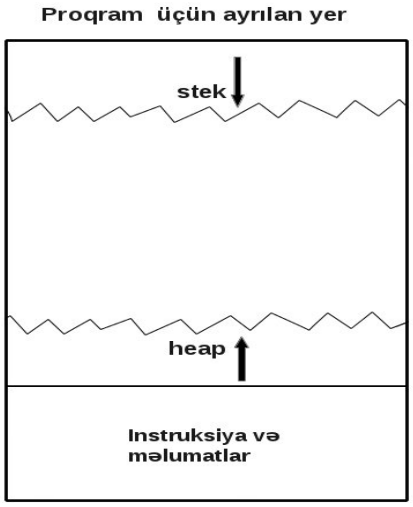
Bu paraqrafda stek ilə tanış olacayıq. Stek assemblerin ən vacib mövzularındandır.

Müasir kompüter arxitekturaları funksiyalara müraciəti stek vastəsilə təmin edir. Əməliyyatlar sisteminin ən zəif hissəsi də məhs stek sayılır. Buna gürə sistemə nəzarəti ələ keçirmək üçün ən geniş yayılmış hücumlar - buferi daşırma(buffer overflow) stekə müdaxilə vastəsilə həyata keçirilir.

Stek nədir ?

Əməliyyatlar sistemi hər-bir proqrama yaddaşda müəyyən sahə ayırır. Bu sahənin bir hissəsi proqramın instruksiya və məluatlarının yerləşdirilməsinə sərf olunur. Yerdə qalan sahə isə boş sahə.

Yerdə qalan boş sahəni proqram 2 məqsəd üçün istifadə edir: funksiya parametrləri və dinamik dəyişənlərlə işləmək üçün. Bu sahələr uyğun olaraq stek və heap adlanır.



Stek proqrama aid yaddaş sahəsidir və əsasən funksiyalar tərəfindən istifadə olunur. Stekə məlumat yerləşdirmək və stekdən məlumat götürmək xüsusi qayda ilə həyata keçirirlir. Bu dizayn funksiyaların çağırılmasını və funksiyalardan qayıtmanı təmin etmək üçündür.

Stekin proqrama aid bir yaddaş sahəsi olduğunu örgəndik. İndi stekə necə məlumat yerləşdirmək və stekdən məlumat götürmənin qaydaları ilə tanış olaq.

Əvvəla onu qeyd edim ki, **$2** -də örgəndiyimiz yaddaşa müraciət üsulundan istifadə etməklə stekdən və eləcə də yaddaşın istənilən digər hissəsindən məlumat əldə edə bilərik, lakin stek yaddaşı ilə işləmək üçün xüsusi instruksiyalar və əlavə reqistrlər tərtib olunmuşdur. Bu istruksiyalar **push** və **pop**, reqistrlər isə **%rsp** və **%rbp** reqistrləridir.

Diqqət! Bu və növbəti Funksiyalar mövzusunda steklə işləmək üçün istifadə olunan mexanizm 64 bitlik arxitektura üçündür. 32 birlik kompüterlərdə bu və növbəti mövzunun proqramları işləməz. Son zamanlar 64 bitlik kompüterlərin tədricən geniş yayıldığını nəzərə alaraq bu seçimi etdik. Lakin bəzi məqamları nəzərə alıb proqramların müvafiq 32 bitlik versiyalrını tərtib edə bilərsiniz, bunlar aşağıdakılardır:

Reqistrlər uyğun olaraq **%esp** bə **%ebp** -reqistrləridir. Stek əməliyyatları zamanı (**push, pop**) stek reqistrinin qiymməti **4** bayt dəyişir(azalır, artır).

Stekin iş prinsipi.

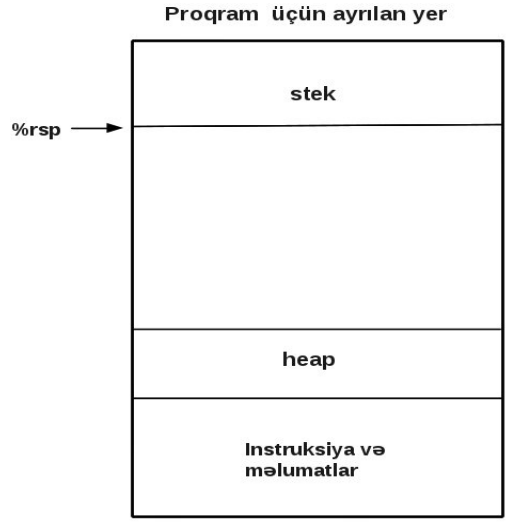
Steklə işləməyi örgənmək üçün aşağıdakıları bilməliyik:

Proqramın əvvəlində stek boş olur, yəni stekdə heç bir məlumat olmur. Proqramın icrası boyu

stekə məlumatlar yerləşdirilə (**push**) və stekdən məlumatlar götürülə (**pop**) bilər. Bu zaman stek yaddaşının həcmi müvafiq olaraq artır və azalır.

Stekin ən üstü

Bilməyimiz gərəkən ən vacib məqam və ümumiyyətlə stekin mahiyyətini təyin edən məqam məlumatların stekə necə gəldi yox, yalnız və yalnız bir yerdən əlavə olunması və götürülməsidir. Bu yer, başqa sözlə stekdə olan məlumatlara və ya stekə istinad yeri stekin ən üstü adlanır və **%rsp** reqistri ilə təyin olunur. Biz stekə məlumat yerləşdirərkən və ondan məlumat götürərkən prosessor **%rsp** reqistrin qiymətini avtomatik yeniləyir və **%rsp** reqistri həmişə stekin ən üstünə istinad edir.



Vəziyyəti çətinləşdirən amil

Bu yerdə başqa bir məsələni də bilməyimiz zəruridir, hansı ki, öz növbsində steklə işləməyi örgənməyi daha da çətinləşdirir. Stek yuxarıdan aşağıya doğru artır. Bu nə deməkdir ? Biz adət eləmişik ki, nəyəsə bir şey əlavə edən zaman onda artma baş verir və bu artım özünü rəqəmlər vastəsilə ifadə edir. Stekdə hər-şey tərsinədir. Proqramın əvvəlində (yəni stek boş olanda) stekin ən üstü özünün maksimum qiymətində olur. Tədricən stekə məlumat yerləşdirən zaman stek göstəricisinin qiyməti (**%rsp**) biz adət etdyimizin əksinə olaraq artmaq əvəzinə azalır. Hələlik bunu tam aydın başa düşməyə bilərsiniz. Assemblerin bəzi mövzuları tədriclə başa düşülür. Sadəcə bilmək kifayətdir ki, stekə məlumat yerləşdirən zaman stek göstəricisinin qiyməti azalır, stekdən məlumat götürdükdə isə artır.

Stekə məlumat yerləşdirmək - Push

Stekə məlumat yerləşdirmək üçün push instruksiyasından istifadə olunur. **32** bitlik sistemlərdə **pushl**, **64** bitlik sistemlərdə isə **pushq** instruksiyası istifadə olunur. **Push** instruksiyası bir arqument qəbul edir, stekə yerləşdirilməli olan məlumat. Bu konkret ədəd , reqistrdə və ya yaddaşda yerləşən məlumat ola bilər. Nəticədə həmin məlumat stekin ən üsünə yerləşdirilir və stek göstəricisi aşağıya doğru sürüşür. Stek göstəricisinin (**%rsp**) qiyməti **32** bitlik sistemlərdə **4, 64** bitlik sistemlərdə **(%rsp**) isə **8** vahid azalır.

Əvvəl

Stek

```
 _________
|_________|
|_________|
|_________| ← %rsp  köhnə
```

```
push məlumat
```

Sonra

Stek

```
 _________
|_________|
|_________|
|_________|
|_məlumat_| ← %rsp yeni
```

```
%rsp yeni = %rsp köhnə - 8
```

Stekdən məlumat götürmək - Pop

Stekdən məlumat götürmək üçün pop instruksiyasından istifadə olunur. **32** bitlik sistemlərdə **popl**, **64** bitlik sistemlərdə isə **popq** instruksiyası istifadə olunur. **Pop** instruksiyası bir arqument qəbul edir reqistr və ya ünvan. Nəticədə stekin ən üstündən **4** və ya **8** bayt məlumat götürülərək göstərilən yerə (reqistr , ünvan) yerləşdiriləcək. Stek göstəricisinin

(**%esp**) qiyməti **32** bitlik sistemlərdə **4**, **64** bitlik sistemlərdə (**%rsp**) isə **8** vahid artmış olur.

Əvvəl

Stek

```
 _________
|_________|
|_________|
|_________|
|_məlumat_| ← %rsp  köhnə
```

```
pop reqistr
```

Sonra

Stek

```
 _________
|_________|
|_________|
|_________| ← %rsp  yeni
```

Stek

```
%rsp yeni = %rsp köhnə + 8
```

Növbəti çalışmalarda push və pop instruksiyaları vastəsilə stekə məlumat yerləşdirilməsi və stekdən məlumat götürülməsi, bu zaman stek göstəricisinin qiymətinin necə dəyişməsi yoxlanılacaq. Bu bizə funksiyaların çağırılması və qayıtmasını başa düşməyə kömək edər.

Çalışma 1. Stekə **10** qiyməti yerləşdirən proqram tərtib edin.

Həlli. Nümunə kod aşağıdakı kimi olar:

```
.data

.text

.globl _start

_start:

pushq $10
```

```
son:

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

Çalışma 2. **%rax** reqistrini stekə yerləşdirən proqram kodu tərtib edin.

Həlli. Nümunə kod aşağıdakı kimi olar:

```
  .data

  .text

  .globl _start

  _start:

  pushq %rax

son:

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

Çalışma 3. stekin ən üstündə olan məlumatı **%rbx** reqistrinə köçürən proqram tərtib edin.

Həlli. Nümunə kod aşağıdakı kimi olar:

```
  .data

  .text

  .globl _start

  _start:

  popq %rbx

son:

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

Çalışma 4. Stekdən istifadə etməklə **%ebx** reqistrinə **456** qiyməti yazan proqram tərtib edin.

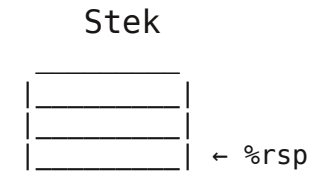
Həlli. Nümunə kod aşağıdakı kimi olar:

```
 .data

 .text

   .globl _start

   _start:

    pushq $456

    popq %rbx

son:

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

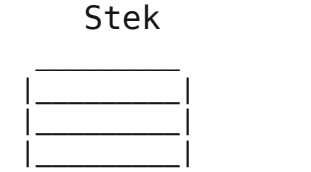
Test:

```
(gdb) break son
Breakpoint 1 at 0x40007e
(gdb) run
Starting program: /home/ferid/Documents/tmp

Breakpoint 1, 0x000000000040007e in son ()
(gdb) info registers $rbx
rbx            0x1c8    456
(gdb)
```

İzahı: Tutaq ki, proqramın əvvəlində stekin vəziyyəti aşağıdakı kimidir:



pushq $456 instruksiyası stekə **456** qiymətini yerləşdirər və stek göstəricisi aşağı sürüşər.



```
|___456___| ← %rsp
```

Daha sonra həmin qiyməti **%ebx** reqistrinə yazmaq üçün **popq %rbx** instruksiyasından istifadə edirik. Nəticədə stekin ən üstündə olan məlumat, **456** ədədi **%rbx** reqistrinə köçürülür və stek reqistri yuxarı sürüşür(əvvəlki vəziyyətinə qayıdır).

```
    Stek
 _________
|_________|
|_________|
|_________| ← %rsp
```

Çalışma 5. Aşağıdakı proqram icra olduqda **son** nişanında **%ecx** reqistrinin qiyməti neçə olar?

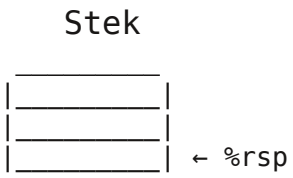
```
 .data

 .text

   .globl _start

   _start:

    pushq $25
    pushq $78
    pushq $45

    popq %rcx

son:

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

Test:

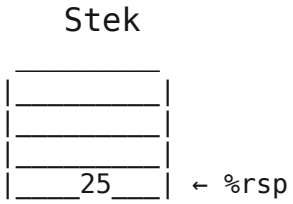
```
(gdb) break son
Breakpoint 1 at 0x40007f
(gdb) run
Starting program:

Breakpoint 1, 0x000000000040007f in son ()
(gdb) info registers $rcx
```

```
rcx            0x2d     45
(gdb)
```

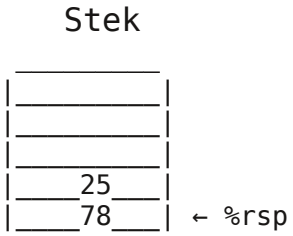
İzahı: Tutaq ki, proqramın əvvəlində (_**start** nişanı) stekin vəziyyəti aşağıdakı kimidir:



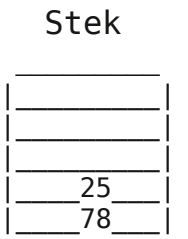
Proqramın ilk instruksiyası **pushq $25** stekə **25** qiymətini yerləşdirir və stek göstəricisi aşağı sürüşür.



Növbəti instruksiya stekə **78** qiymətini yerləşdirir, **pushq $78**.



Görgüyümüz kimi yeni qiymətlər stekə həmişə ən üstdən yerləşdirilir. Proqramın növbəti instruksiyası stekə 45 qiymətini yerləşdirir, **pushq $45**. Stek göstəricisi (**%rsp**) aşağı sürüşür .



```
|____45___| ← %rsp
```

Növbəti instruksiya stekin ən üstündə olan məlumatı **%ecx** reqistrinə yazır və stek reqistri yuxarı sürüşür, **popq %rcx.** Baxdığımız halda stekin ən üstündə **45** qiyməti dayandığından (ən üstdə həmişə stekə ən son yerləşdirilən məlumat yerləşir) **%ecx** -ə **45** qiyməti yazılar.

```
    Stek
 _________
|_________|
|_________|
|_________|
|____25___|
|____78___| ← %rsp
```

Bu çalışmada biz stek ilə işin mahiyyətini, yəni ən sonuncu yerləşdirilən ən birinci çıxır prinsipini izah etdik.

Çalışma 6. Stekdən istifadə etməklə **%eax** və **%ebx** reqistrlərinin qiymətini dəyişən proqram tərtib edin.

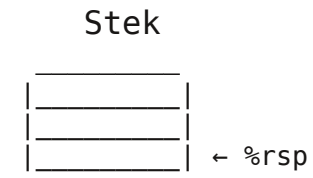
Həlli. Nümunə kod aşağıdakı kimi olar:

```
 .data

 .text

   .globl _start

   _start:

  movq $1, %rax
  movq $2, %rbx

  pushq %rax
  pushq %rbx

y:
  popq %rax
  popq %rbx

son:

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

Test:

```
(gdb) break y
Breakpoint 1 at 0x400088
(gdb) break son
Breakpoint 2 at 0x40008a
(gdb) run
Starting program:
Breakpoint 1, 0x0000000000400088 in y ()
(gdb) info registers $rax $rbx
rax            0x1      1
rbx            0x2      2
(gdb) c
Continuing.

Breakpoint 2, 0x000000000040008a in son ()
(gdb) info registers $rax $rbx
rax            0x2      2
rbx            0x1      1
(gdb)
```

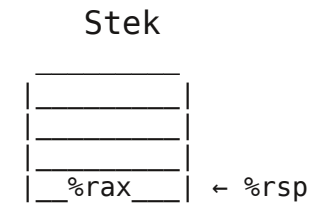
İzahı: Proqramın əvvəlində biz **%eax** və **%ebx** reqistrlərinə uyğun olaraq **1** və **2** qiymətləri yerləşdirirk. Proqramın sonunda isə həmin qiymətlər dəyişilmiş olur. Məqsəd sadəcə stekin necə işlədiyini izah etəkdir. Stekə ən son yerləşdirilən məlumat ən birinci çıxarılır. Tutaq ki, proqramın başlanğıcında stekin vəziyyəti aşağıdakı kimidir:



Əvvəlcə stekə **%rax** reqistrinin qiymətini yerləşdiririk:

```
pushq %rax
```

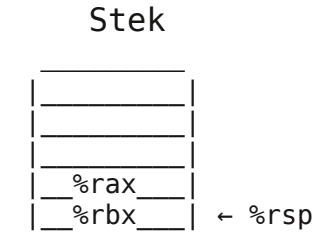
%rax reqistrinin qiyməti stekə yazılır. **%rsp** reqistrinin qiyməti **8** bayt ***"aşağı"*** sürüşür(azalır).



Daha sonra stekə **%rbx** reqistrinin qiymətini yerləşdiririk:

```
pushq %rbx
```

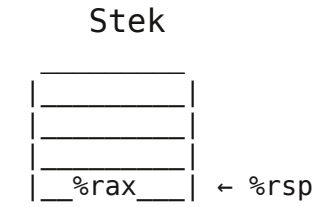
%rbx reqistrinin qiyməti stekə yerləşdirilir. **%rsp** reqistrinin qiyməti növbəti dəfə **8** bayt ***"aşağı"*** sürüşür(azalır).



Hal-hazırda stekin ən üstündə **%rbx,** ondan sonra isə (stekin yuxarısına doğru) **%rax** reqistrləri yerləşir. İndiki vəziyyətdə pop instruiksiyası ilə stekdən məlumat götürsək birinci olaraq stekin ən üstündə olan məlumat qaytarılacaq, yəni **%rbx** -in qiyməti və stek reqistri 8 bayt **"yuxarı"** sürüşəcək. Növbəti instruksiya bu işi yerinə yetirir.

```
popq %rax
```

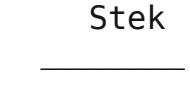
Yuxarıdakı instruksiya reqistrin ən üstündə olan məlumatı göterərək **%eax** reqistrinə yazır və stek göstəricisi **"yuxarı"** sürüşür.



Növbəti instruksiya ilə stekin ən üstündə olan **8** bayt məlumatı **%ebx** reqistrinə köçürürük.

```
popq %rbx
```

Stekin vəziyyəti aşağıdakı kimi olar:



%eax və **%ebx** reqistrlərini stekə yerləşdirdikdən sonra onları yerləşdirdiyimiz ardıcıllığın əks ardıcıllığı olaraq stekdən çıxarırıq.

Suallar.

1. Stek nədir ?
2. Stekə məlumat yerləşdirmək üçün hansı instruksiyadan istifadə olunur?
3. Stekdən məlumat götürmək üçün hansı instruksiyadan istifadə olunur?
4. Stek göstəricisi nədir ?
5. Prosessorun hansı reqistri həmişə stekin **"ən üstünə"** istinad edir ?
6. 64 bitlik sistemdı pushq instruksiyası icra edərkən **%rsp** reqistrinin qiyməti necə dəyişir?
7. 64 bitlik sistemdı popq instruksiyası icra edərkən **%rsp** reqistrinin qiyməti necə dəyişir?
8. Aşağıdakı kod icra olunduqda **%rbx** reqistrinin qiyməti neçə olar?

```
pushq $56
pushq $32
movq $4, %rax
pushq $rax
popq %rbx
```

9. Tutaq ki, **%rax** reqistrinin qiyməti **12** -yə, **%rbx** reqistrinin qiyməti isə **34** -ə bərabərdir.

```
movq $12, %rax
movq $34, %rbx
```

Aşağıdakı koddan sonra hər iki reqistrin qiyməti neçə olar?

```
pushq %rax
popq  %rbx
```

Yaddaş tədqiqi.

Çalışma 1. Stek göstəricisinin qiymətini müəyyənləşdirin.

Həlli. Sadə bir proqram tərtib edək və proqramı kod hissəsində hər-hansı nişan təyin edək.

Həmin nişanda proqramın icrasını saxlayaq və stek reqistrinin qiymətini örgənmək üçün **info registers $rsp** əmrini daxil edək.

Nümunə proqram aşağıdakı kimi olar:

```
.data

.text

.globl _start

_start:


movl $4, %ebx

son:

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

Nəticə:

```
(gdb) break son
Breakpoint 1 at 0x4000bc
(gdb) run
Starting program:

Breakpoint 1, 0x00000000004000bc in son ()

(gdb) info registers $rsp
rsp   0x7fffffffe208      0x7fffffffe208
(gdb)
```

Burada biz **son** nişanında proqramın icrasını dayandırırıq və stek göstəricisinin qiymətini yoxlayırıq. Stek registri **0x7fffffffe208** ünvanına istinad edir. Bu qiymət 16-lıq say sistemində verilib. Say sistemləri ilə 6-cı paraqrafda tanış olacayıq.

Çalışma 2. Stek göstəricisinin qiymətini təyin edin və yaddaşın həmin ünvanında (stekdə) yerləşən 64 bayt məlumatı çap edin .

Həlli. Əvvəlcə stekin göstəricisinin qiymətini örgənməliyik. Daha sonra yaddaşı oxu əmri ilə stek göstəricindən 64 bayt əvvəl gələn məlumatı oxuyacayıq. Nəzərə almalıyıq ki, stekə məlumatlar ünvanın azalması istiqamətində əlavə olunur.

Nümunə proqram kodu aşağıdakı kimi olar:

```
.data

.text

.globl _start

_start:


movl $4, %ebx

son:

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

Proqramı **gdb** ilə yükləyək və **son** nişanında proqramın icrasını dayandıraq. Daha sonra stek göstəricisinin qiymətini örgənək:

```
(gdb) break son
Breakpoint 1 at 0x40007d
(gdb) run
Starting program:

Breakpoint 1, 0x000000000040007d in son ()
(gdb) info registers $rsp
rsp            0x7fffffffe250   0x7fffffffe250
(gdb)
```

İndi isə bu ünvandan **64** bayt yuxarı (stek boyu yuxarı qalxdıqca yaddaş ünvanı azalır) hissəni təhlil etməliyik. Bunun üçün x əmrindən istifadə etməliyik. Əmr aşağıdakı kimi olar:

```
(gdb) x /8dg 0x7fffffffe250
```

Burada **gdb** -yə deyilir ki, **0x7fffffffe250** ünvanından başlayaraq **8** baytlıq hissələrlə yaddaşın növbəti **8** hissəsini çap et. Cəmi çap olunan yaddaşın ölçüsü **8*8 = 64** olur. Əmri daxil edək:

```
(gdb) x /8dg 0x7fffffffe250
0x7fffffffe250: 1       140737488348492
0x7fffffffe260: 0       140737488348518
0x7fffffffe270: 140737488348551 140737488348568
0x7fffffffe280: 140737488348584 140737488348617
(gdb)
```

Çalışma 3 . Stekə müxtəlif qiymətlər yerləşdirin və yaddaşı tədqiq edin.

Həlli. Nümunə proqram kodu aşağıdakı kimi olar:

```
.data

.text

.globl _start

_start:

#steke muxtelif qiymetler yerleshdirek
pushq $9
pushq $45
pushq $890
pushq $234
pushq $78
pushq $312
pushq $7
pushq $3466
pushq $555
pushq $809
pushq $3

son:

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

Əvvəlki qayda ilə stekin yaxın ətrafını tədqiq edək:

```
(gdb) break son
Breakpoint 1 at 0x4000a0
(gdb) run
Starting program:

Breakpoint 1, 0x00000000004000a0 in son ()
(gdb) info registers $rsp
rsp            0x7fffffffe1f8   0x7fffffffe1f8
(gdb) x /12dg 0x7fffffffe1f8
0x7fffffffe1f8: 3       809
0x7fffffffe208: 555     3466
0x7fffffffe218: 7       312
0x7fffffffe228: 78      234
0x7fffffffe238: 890     45
0x7fffffffe248: 9       1
(gdb)
```

5
Funksiyalar

Bu bölmədə Assemblerin ən vacib mövzusu – Funksiyalar, funksiyaların proqram kodlarının tərtibi, funksiyalara parametr ötürmək və eləcə də x86 arxitekturasında funksiyanın çağırılması və geri qayıtması mexanizmi ilə tanış olacayıq.

Funksiyanın proqram kodu

Funksiyalar proqramın **.text** hissəsində elan olunmuş nişanlardır. 1-ci başlıqda **jmp** – keçid instruksiyası ilə tanış olduq və qeyd etdik ki, **jmp** instruksiyası proqramın icrasını göstərilən nişandan davam etdirmək üçündür. Funksiyalardan da bu məqsəd üçün istifadə edirik. Fərq yalnız ondadır ki, funksiyalar çağırıldığı ünvanı ***"yadda"*** saxlayır və buna görə proqramın hansı yerindən çağırmağımızdan asılı olmayaraq, öz işini qurtardıqdan sonra funksiya həmin yerə ***"geri qayıda"*** bilir.

Bundan əlavə funksiya çağıran zaman biz stekdən istifadə etməklə funksiyaya parametrlər də ötürə bilərik. Qarşıdakı mövzuda bu və digər məsələlərin nə cür yerinə yetirildiyi müzakirə olunur.

Funksiyanı çağrmaq.

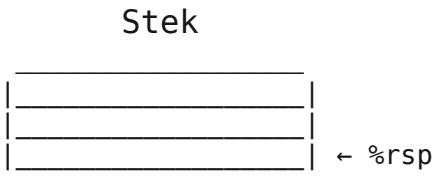
Funksiyanı çağırmaq üçün **call** instruksiyasından istifadə olunur. **call** instruksiyası bir arqument qəbul edir, çağırılmalı olan funksiyanın adını aşağıdakı kimi:

```
call  funksiyanın_adı
```

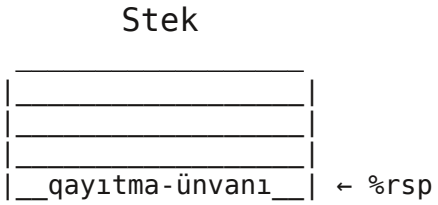
Nəticədə **funksiyanın_adı** funksiyası icra olunur, başqa sözlə **funksiyanın_adı** nişanına keçid edilir. **call** instruksiyası təkcə göstərilən nişana keçid etmir (əks halda bunun üçün sadəcə **jmp** instruksiyasından istifadə edərdik), həm də funksiyanın "***geri qayıda"*** bilməsi üçün funksiyanın ***"qayıtma ünvanını"*** stekə yerləşdirir.

Buradan bir məsələyə diqqət yetirmək tələb olunur: funksiyanın qayıtma ünvanı stekə yerləşdirilir, yəni **call** instruksiyası icra olunan zaman stekin vəziyyəti dəyişir. Bunu aşağıdakı fiqurdan görə bilərik.

Tutaq ki, **call** instruksiyasından əvvəl stekin vəziyyəti aşağıdakı kimidir:



call instruksiyasından sonra stekin vəziyyəti aşağıdakı kimi dəyişər:



Qayıtma ünvanı stekə yerləşdirilir və stek göstəricisi aşağı sürüşür.

Funksiyadan geri qayıtmaq.

Funksiyadan geri qayıtmaq üçün **ret** instruksiyasından istifadə olunur. **ret** instruksiyası heç bir arqument qəbul eləmir, aşağıdakı kimi:

```
ret
```

Bu instruksiya icra olunan zaman icraolunma funksiyanın çağırıldığı yerə qayıdır və həmin yerdən davam edir. Bunun üçün **ret** instruksiyası sadəcə stekin ən üstündə yerləşən məlumatı qayıtma ünvanı kimi qəbul edir və həmin ünvana keçid edir. Buradan bir məqam ortaya çıxır ki, funksiyadan qayıdan zaman stekin ən üstündə funksiyanın qayıtma ünvanı yerləşməlidir. Əks halda başqa yerə keçid olunar.

Deməli funksiyanı çağırdıqdan sonra və funksiya kodun icra etdikdə stek üzərində əməliyyat aparan zaman bir məsələdən əmin olmalıyıq ki, biz funskiyanın qayıtma ünvanın korlamırıq və bir də stek əməliyyatları stek göstəricisini yuxarı-aşağı sürüşdürdüyündən funksiyadan qayıdan zaman stek göstəricisinin funksiyanın qayıtma ünvanına istinad etməsini təmin etməliyik. Başqa sözlə **ret** instruksiyasını icra etməzdən öncə tam əmin olmalıyıq ki, stekin ən üstündə qayıtma ünvanı yerləşir.

Buna nəzarət etmək isə hətta kifayət qədər kiçik ölçülü proqram kodlarında belə müşkül məsələdir. Sistem mühəndisləri çıxış yolu olaraq **%rbp** reqistrindən istifadəni təmin etmişlər. %rbp reqistrindən istifadədə məqsəd stek göstəricisinin funksiya çağırılan andakı qiymətini qorumaq və funksiya qayıdan zaman həmin qiyməti bərpa etməkdir. Bunun üçün funksiya kodunun başlanğıcında ilk olaraq aşağıdakı iki instruksiya icra olunmalıdır:

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

Nəticədə **%rbp** reqistri stekə yerləşdirilir və onun qiyməti stek göstəricisinin qiyməti ilə əvəzlənir. Funksiyanın icrası zamanı **%rbp** -reqistrinin qiymətini dəyişmək olmaz. Beləlikə **%rbp** funksiyanın icrası boyu həmişə stekin üstünün funksiya çağırılan andakı qiymətinə istinad etmiş olur. Funksiyadan qayıtma zamanı isə biz bu instruksiyaları əksinə icra etməklə stek göstəricisinin və **%rbp** reqistrlərinin funksiya çağırılan andakı qiymətlərini bərpa etmiş olarıq, aşağıdakı kimi:

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

Artıq bu zaman **%rsp** reqistri funksiya çağırılan andakı vəziyyətinə qayıtmış olur və həmin vəziyyətdə stekin ən üstündə funskyanın qayıtma ünvanı yerləşdiyindən ret instruksiyasını icra edə bilərik. Bunları nəzərə alsaq assembler funksiyalarının ilk 2 sətri

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

son 3 sətri aşağıdakı kimi olmalıdır:

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

Beləliklə biz assembler funksiyasının prototipini almış oluruq:

```
funksiyanın_adı:
    #hazırlıq işləri
    pushq %rbp
    movq %rsp, %rbp

    #funksyanın proqram kodu


    #geri qayıtma kodu
    movq %rbp, %rsp
    popq %rbp
    ret
```

Yuxarıda biz **x86** arxitekturalı prosessorların funksyanı çağırılması və geri qayıtması mexanizmi ilə tanış olduq. Lakin bunları bilmək hələ assembler dilində funksiyalarla işləməyin ***"dadını çaxartmağa"*** imkan vermir. Həll olunması gərəkən bir məsələ və kifayət qədər vacib məsələ funksiyalara parametr ötürülməsi məsələsidir, hansı ilə ki biz növbəti mövzuda tanış olacayıq. Amma etiraf edim ki, bu call və ret instruksiyalarının işini başa düşmək qədər çətin deyil, əgər yaddaşa müraciət və stek əməliyyatları yaxşı mənimsənilibsə.

Funksiyaya parametr ötürmək

Stek bölməsində stek yaddaşının funkiyalar üçün çox vacib əhəmiyyət daşıdığını qeyd etmişdik. Bu yalnız funksiyanın qayıtma ünvanın stekdə yerləşdirilməsi ilə bitmir. Funksiyaya ötürülən parametrlər də stekə yerləşdirilir.

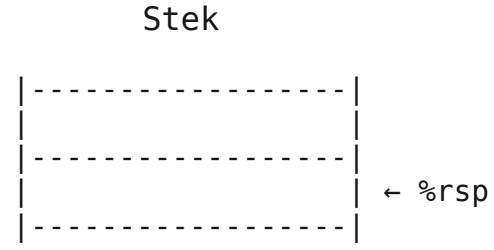
Funksiyaya parametr ötürmək üçün funksiyanı çağrmazdan öncə ona ötürmək istədiyimiz parametrləri stekə yerləşdirməliyik, daha sonra isə funkiyanı çağırmalıyıq. Funksiya kodunda isə yaddaşa müraciət etmə üsullarından istifadə etməklə həmin parametrləri əldə edə bilərik.

Bu dediklərimizi əyani göstərmək məqsədilə aşağıda yalnız bir parametr qəbul edən funksiya tərtib edəcəyik. Test məqsədi üçün tərtib etdiyimiz funksiya ona ötürülən parametrin qiymətini bir vahid artırır. Lakin bu tamamilə test məqsədi üçün tərtib olunan bir fuksiyadır və məqsəd sadəcə parametrin stekdən əldə olunması qaydası ilə tanış olmaqdır. Növbəti tərtib edəcəyimiz test proqramda isə funksiyaya parametr olaraq dəyişən ünvanı ötürəcək və yaddaşa müraciət üsullarından istifadə etməklə funksiyanın nəticəsini həmin dəyişənə mənimsədəcəyik. Bunu artıq funksiyalardan istifadənin nisbətən daha praktik nümunəsi hesab etmək olar.

Beləliklə aşağıdakı proqram nümunəsində art adlı funksiya tərtib edirik. Funksiya bir parametr qəbul edir və həmin parametrin qiymətini bir vahid artırır. Daha sonra proqram kodunun ətraflı izahını verəcəyik. Nümunə proqram kodu aşağıdakı kimi olar:

```
.data

.text

.globl _start

_start:

    #atr funksiyasina 65 qiymetini parametr olaraq
    #oturek
    pushq $65

    #atr funksiyasini cagiraq
    call art

son:

    movl $1, %eax
    int  $0x80

    #art funksiyasinin proqram kodu
art:
    #hazırlıq işləri
    pushq %rbp
    movq %rsp, %rbp

    #funksyanın proqram kodu
    #funksiyaya oturulen parametri stekden %rbx -e kocurek
    movq 16(%rbp), %rbx

    #parametrin qiymetini 1 vahid artiririq
    incq %rbx

    #geri qayıtma kodu
    movq %rbp, %rsp
    popq %rbp
    ret
```

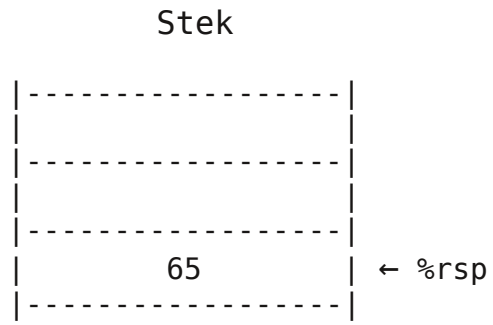
Izahı: Tutaq ki, proqramın əvvəlində stekin vəziyyəti aşağıdakı kimidir:



Əvvəlcə stekə 65 qiyməti yerləşdiririk

```
pushq $65
```

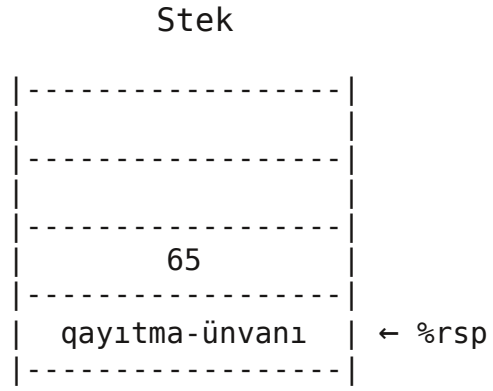
Stekin vəziyyəti:



Daha sonra art funksiyasını çağırırıq:

```
call art
```

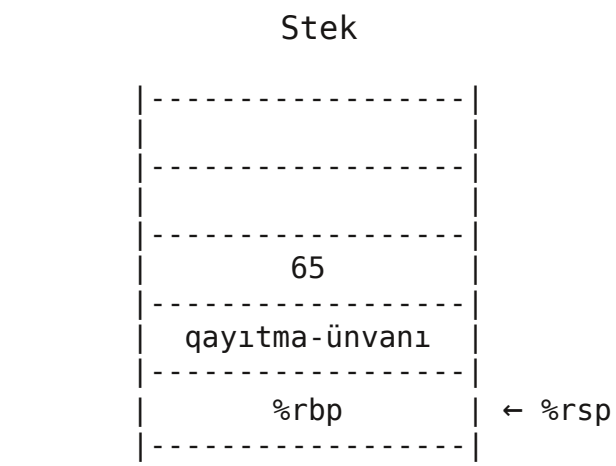
call instruksiyası yuxarıda qeyd etdiyimiz ki, funksiyanın qayıtma ünvanını stekə yerləşdirir. Ona görə call instruksiyasından sonra stekin vəziyyəti aşağıdakı kimi olar:



call instruksiyası qayıtma ünvanın stekə yerləşdirdikdən sonra proqramın icra istiqamətini art nişanına yönləndirir və proqram art nişanından(funksiyasından) icra olunmağa başlayır. art funksiyasında yuxarıda qeyd etdiyimiz kimi əvvəlcə hazırlıq işləri görülür və **%rbp** reqistri stekə yerləşdirilir.

```
pushq %rbp
```

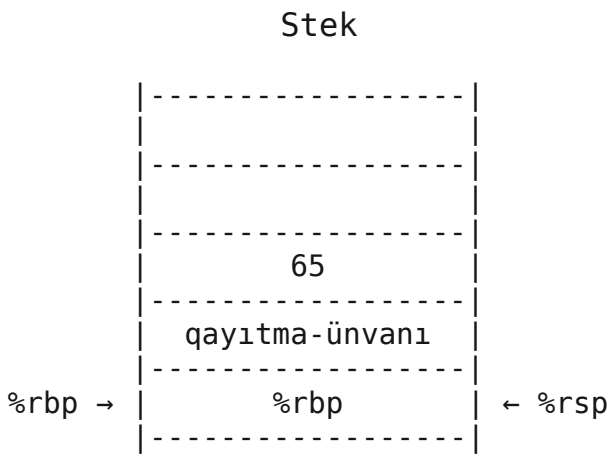
Stekin vəziyyəti aşağıdakı kimi olar:



Daha sonra isə **%rbp** -yə **%rsp** -ni köçürürük:

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

Nəticədə **%rbp** **%rsp** -nin istinad etdiyi ünvana, stekin ən üstünə istinad etmiş olur:



Daha sonra aşağıdakı əmr ilə funksiyaya örüdüyümüz parametri **%rbx** reqistrinə köçürürük:

```
movq 16(%rbp), %rbx
```

Yaddaşa müracət üsuluna görə `16`(%rbp) ifadəsi yaddaşın `%rbp + 16 ünvanına` istinad edir. Bu **%rbp** -dən 16 bayt yuxarı ünvandır. **%rbp** -nin hal-hazırda istinad elədiyi ünvanda **%rbp** -nin əvvəlki qiyməti, ondan 8 bayt yuxarıda qayıtma ünvanı, 16 bayt yuxarıda isə

funksiyaya ötürdüyümüz parametr yerləşir. 64 bitlik sistelərdə stekə yerləşdirilən hər-bir məlumat 8 bayt yer tutur(32 bitlik sistemlərdə 4 bayt).

Beləlikə yuxarıdakı instruksiya funksyaya ötürdüyümüz parametri **%rbx** reqistrinə köçürür.

Çalışma 1. İki dəyişənin cəmini hesablamaq üçün cem funksiyası tərtib edin. cem funksiyasından istifadə etməklə proqram qurun.

Həlli. Əvvəlcə proqram kodunu daxil edək, daha sonra izahı verərik:

```
.data

# long tipinden x ve y deyishenleri icra edek
x:
.long 5

y:
.long 14

s:
.long 0

.text

.globl _start

_start:
#s -in unvanin steke yerleshdir
pushq $s
# ikinci deyisheni steke yerleshdir
pushq y
# birinci deyisheni steke yerleshdir
pushq x
# cem funksiyasini cagir
call cem

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


#cem funksiyasi
cem:
pushq %rbp
movq %rsp, %rbp

# birinci parametri %rax -e yazaq
movq 16(%rbp), %rax

# ikinci parametri %rbx -e yazaq
movq 24(%rbp), %rbx

# %rax -le %rbx -i cemleyek
addq %rax, %rbx
```

```
# cemi s deyishenine yazaq
# evvelce s -in unvanin stekden %rax -e kocurek
movq 32(%rbp), %rax

# indi ise unvani %rax -de olan yere(s) neticeni yazaq
movq %rbx, (%rax)

# geri qayidiriq
movq %rbp, %rsp
popq %rbp
ret
```

Izahı:

Proqramda **long** tipindən **x, y** və **s** dəyişənləri elan edirik və onlara uyğun olaraq 5, 14 və 0 qiymətləri mənimsədirik. cem funksiyasını çağırmazdan öncə ona ötürülən parametrləri stekə yerləşdiririk. İlk olaraq stekə s dəyişəninin ünvanını yerləşdiririk. Daha sonra **y** və **x** dəyişənlərinin qiymətlərini yerləşdiririk. Bu halda stekin vəziyyəti aşağıdakı kimi olar:

```
s-in ünvani__|
y          __|
x          __| <-- %rsp
```

call cem instruksiyası ilə **cem** funksiyanı çağırırıq. **call** instruksiyası qayıtma ünvanın stekə yerləşdirir **cem** fumksiyasını çağırır. Stek belə olar:

```
s-in ünvani__|
y          __|
x          __|
qyt.ünv    __| <-- %rsp
```

cem funksiyası icra olunur və ilk olaraq **%rbp** -ni stekə yerləşdirir və **%rsp** -ni **%rbp-**yə köçürür. Stek belə olar:

```
s-in ünvani __|
y           __|
x           __|
qyt.ünv     __|
%rbp(kohne) __| <-- %rsp
```

Daha sonra birinci dəyişəni (x) srtekdən **%rax** -ə köçürürük.

```
movq 16(%rbp), %rax
```

Paraqraf 2 -də örgəndiyimiz unvanin hesablanma qaydasına əsasən **16(%rbp) = 16 + %rbp**. Bu ünvanda isə **x** dəyişəni yerləşir.

```
Stek
------------
```

```
s-in unvani __| <-- %rbp + 32
y           __| <-- %rbp + 24
x           __| <-- %rbp + 16
qay.ünv     __| <-- %rbp + 8
%rbp(kohne) __| <-- %rsp, %rbp(yeni)
-----------
```

Daha sonra **movq 24(%rbp), %rbx** instruksiyası ilə **y**-i **%rbx** -e yazırıq.

addq %rax, %rbx instruksiyası **%rax-**lə **%rbx** -i cəmləyir.

s-in ünvanın **%rax** -ə köçürürük.

movq 32(%rbp), %rax

%rbx -i unvanı **%rax** -de yerləşən yaddaşa köçürürük.

movq %rbx, (%rax)

Daha sonra **%rbp** və **%rsp** reqistrlərinin əvvəlki qiymətlərini bərpa edirik və funksiyadan geri qayıdırıq.

Çalışma 2. **Çalışma 1** -də tərtib olunmuş **cem** funksiyasını test edin.

Əvvəlcə proqramı kompilyasiya edib **gdb** ilə yükləyək, **cem** və **son** nişanlarında dayanma təyin edək:

```
(gdb) break cem
Breakpoint 1 at 0x4000d3
(gdb) break son
Breakpoint 2 at 0x4000c8
(gdb)
```

Proqramı icra edək. Proqram icra olunacaq və **cem** nişanında(funksiya) dayanacaq.

```
(gdb) run
Starting program:

Breakpoint 1, 0x00000000004000d3 in cem ()
(gdb)
```

Bu yerdə **s** dəyişənin qiymətini örgənmək üçün **print s** əmrini daxil edək:

```
(gdb) print s
$1 = 0
(gdb)
```

Daha sonra proqramı davam etmək üçün **c** əmrini daxil edək, proqram icra olunacaq və son nişanında dayanacaq.

```
(gdb) c
Continuing.

Breakpoint 2, 0x00000000004000c8 in son ()
```

```
(gdb)
```

Artıq cəm funksiyası icra olunmuşdur, **s** dəyişəninin qiymətini bir daha yoxlayıb funksiyanın düzgün işlədiyinə əmin olaq:

```
(gdb) print s
$2 = 19
(gdb)
```

Çalışma 3. İki ədədin ən böyüyünü hesablamaq üçün **en_boyuk** funksiyası tərtib edin. **en_boyuk** funksiyasından istifadə etməklə proqram qurun.

Həlli: Nümunə proqram aşağıdakı kimi olar:

```
.data

# long tipinden x ve y deyishenleri icra edek
x:
.long 123

y:
.long 56

s:
.long 0

.text

.globl _start

_start:
#s -in unvanin steke yerleshdir
pushq $s
# ikinci deyisheni steke yerleshdir
pushq y
# birinci deyisheni steke yerleshdir
pushq x
# en_boyuk funksiyasini cagir
call en_boyuk

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


#en_boyuk funksiyasi
en_boyuk:
pushq %rbp
movq %rsp, %rbp

# birinci parametri %rax -e yazaq
movq 16(%rbp), %rax

# ikinci parametri %rbx -e yazaq
movq 24(%rbp), %rbx
```

```
# ucuncu parametri %rcx -e yazaq
movq 32(%rbp), %rcx

# muqaise edek
cmp %rax, %rbx
jg birinci_boyuk
movq %rbx, (%rcx)
birinci_boyuk:
movq %rax, (%rcx)

# geri qayidiriq
movq %rbp, %rsp
popq %rbp
ret
```

Nəticə:

```
(gdb) break en_boyuk
Breakpoint 1 at 0x4000d3
(gdb) break son
Breakpoint 2 at 0x4000c8
(gdb) run
Starting program:

Breakpoint 1, 0x00000000004000d3 in en_boyuk ()
(gdb) print s
$1 = 0
(gdb) c
Continuing.

Breakpoint 2, 0x00000000004000c8 in son ()
(gdb) print s
$2 = 123
(gdb)
```

Suallar.

1. Funksiya çağırmaq üçün hansı instruksiyadan istifadə olunur ?

2. **call** instruksiyası ilə **jmp** instruksiyasının fərqi nədir?

3. Funksiyadan geri qayıtmaq üçün hansı isntruksiyadan istifadə olunur ?

4. **ret** instruksiyası geri qayıtma ünvanını necə müəyyən edir ?

5. **%rbp** reqistri nə üçün istifadə olunur ?

Çalışmalar.

Çalışma 1. Funksiyalardan istifadə etməklə cərgənin ən böyük elementini tapan proqram tərtib edin.

Çalışa 2. Funksyalardan istifadə etməklə sətrin uzunluğunu hesablayan proqram tərtib edin.

Çalışma 3. Funksiyalardan istifadə etməklə sətridə verilmiş simvolların sayını tapan proram tərtib edin.

Çalışma 4. Funksiylardan istifadə etməklə tam ədədlər cərgəsində verilmiş ədədin neçə dəfə təkrarlandığını hesablayan proqram tərtib edin

Çalışma 5. Funksiylardan istifadə etməklə cərgənin elementləri cəmini hesablayan proqram tərtib edin.

Yaddaşın təhlili.

Aşağıdakı çalışmanı yerinə yetirməkdə məqsəd call instruksiyasının funksiyanın qayıtma ünvanını stekə yerləşdirdiyini yoxlamaqdır.

Çalışma. **funk** adlı funksiya tərtib edin. Funksiyaya 3 parametr ötürün. Funksiyanın çağrılma yerində proqramın icrasını dayandırın. Daha sonra yaddaşı tədqiq edin və funksiyaya ötürülən parametrlərin stekə yerləşdirdiyinə əmin olun. Daha sonra növbəti icraolunmalı instruksiyanın ünvanın çap edin. Bu funksiyanın qayıtma ünvanıdır. Növbəti instruksiyanı (call) icra edin və steki yoxlayın. Call instruksiyasının funksiyanın qayıtma ünvanını stekə yerləşdirdiyini yoxlayın.

Nümunə proqram kodu aşağıdakı kimi olar:

```
.data

.text

.globl _start

_start:


    #funksiyaya parametrler oturek
    pushq $67
    pushq $89
```

```
    pushq $32


    #proqramin icrasini funk-u cagirmazdan once
    #dayandirmaq ucun stop nishani elan edek
evvel:

    #funk funksiyasini cagiraq
    call funk

sonra:

son:

    movl $1, %eax
    int  $0x80

    #atr funksiyasinin proqram kodu
funk:
    #hazırlıq işləri
    pushq %rbp
    movq %rsp, %rbp

    #funksyanın proqram kodu

    #geri qayıtma kodu
    movq %rbp, %rsp
    popq %rbp
    ret
```

Test:

Proqramın icrasını funksiyanı çağırmazdan öncə dayandıra bilmək üçün **evvel** adlı nişan təyin edirik. Funksiyanın qayıtma ünvanı call instruksiyasından sonra gələn instruksiyanın ünvanıdır və call instruksiyası məhs həmin ünvanı stekə yerləşdirir. Bu ünvanı əldə edə bilmək üçün call instruksiyasından sonra ***sonra*** nişanı təyin edirik. Gdb ilə proqramı yükləyirik və **evvel** və **funk** nişanlarında dayanma təyin edirik.

```
(gdb) break evvel
Breakpoint 1 at 0x40007e: file tmp.s, line 22.
(gdb) break funk
Breakpoint 2 at 0x40008e: file tmp.s, line 40.
(gdb)
```

Proqramı icra edirik. **evvel** nişanında proqramın icrası dayanır.

```
(gdb) run
Starting program: /home/ferid/Documents/tmp

Breakpoint 1, evvel () at tmp.s:22
22          call funk
```

```
(gdb)
```

Bu yerdə biz artıq funksiyaya parametrlər ötürmüşük, lakin **call** instruksiyası hələ icra olunmayıb. Steki yoxlayaq və funksiyaya ötürdüyümüz parametrlərin stekdə oluğuna əmin olaq.

```
(gdb) print sonra
$2 = {<text variable, no debug info>} 0x400083 <sonra>


(gdb) info registers $rsp
rsp            0x7fffffffe238   0x7fffffffe238
(gdb) x /8xg 0x7fffffffe238
0x7fffffffe238: 0x0000000000000020      0x0000000000000059
0x7fffffffe248: 0x0000000000000043      0x0000000000000001
0x7fffffffe258: 0x00007fffffffe54b      0x0000000000000000
0x7fffffffe268: 0x00007fffffffe565      0x00007fffffffe586
(gdb)
```

Funksiyanın qayıtma ünvanını örgənək:

```
(gdb) print sonra
$2 = {<text variable, no debug info>} 0x400083 <sonra>
(gdb)
```

Deməli **call** instruksiyası icra olunanda **0x400083** ünvanını stekə yerləşdirməlidir. Proqramın icrasını davam etsək o **funk** nişanında (funksiyasında) dayanacaq, yəni **call** -dan sonra. **Call** instruksiyasının stek göstəricisinin qiymətini dəyişdirdiyini nəzərə alsaq stek yoxlamaq üçün **%rsp** reqistrinin qiymətini yenidən örgənməli, daha sonra steki yoxlayaraq **0x400083** ünvanının stekə yerləşdirildiyinə əmin olmalıyıq.

```
(gdb)c

Continuing.

Breakpoint 2, funk () at tmp.s:40
40          movq %rbp, %rsp
(gdb) info registers $rsp
rsp            0x7fffffffe228   0x7fffffffe228
(gdb) x /8xg 0x7fffffffe228
0x7fffffffe228: 0x0000000000000000      0x0000000000400083
0x7fffffffe238: 0x0000000000000020      0x0000000000000059
0x7fffffffe248: 0x0000000000000043      0x0000000000000001
0x7fffffffe258: 0x00007fffffffe54b      0x0000000000000000
(gdb)
```

Gördüyümüz kimi **call** instruksiyası qayıtma ünvanını - **0x400083** stekə yerləşdirib. Bunu da yoxlamaq tələb olunurdu.

6
Say Sistemləri

Bu mövzuda biz 2-lik, 10-luq və 16-lıq say sistemlərinin mahiyyəti, bir say sistemindən digərinə keçid qaydalı ilə tanış olacayıq. Qeyd edim ki, 2-lik say sistemləri bizə əsasən bit əməliyyatları, 16-lıq say sistemləri isə ünvanlarla işləyərkən lazım olur.

İkili say sistemi

İkili say sistemində ifadə olunan ədədlər cəmi iki simvol, 0 və 1 simvollarından ibarət olur. Aşağıdakı iki say sistemində ifadə olunan ədədlərə, başqa sözlə ikili ədədlərə müxtəlif nümunələr göstərilir:

```
0,   1,    00,    10,    00101,   111111111,    0101011011
```

Gördüyümüz kimi bu ədədlər sadəcə 0 və 1 -lər ardıcıllığından ibarətdir. Bu ədədləri bizim yaxşı tanıdığımız 10-luq ədədlərə çevirmək üçün qaydalar mövcuddur. Gəlin bu qayda ilə tanış olaq.

İkili ədədin 10-luq ədədə çevrilməsi.

İkili ədədin 10-luq ədədə çevirmənin addımlarını izah edək və konkret misallar üzərində göstərək. Tutaq ki, aşağıdakı ikili ədədi onluq ədədə çevirmək tələb olunur.

1 0 0 1 0 1 0 1 1

Əvvəlcə verilmiş ikili ədədin rəqəmlərini sağdan sola 0-dan başlayaraq nömrələyirik:

1	0	0	1	0	1	0	1	1
8	7	6	5	4	3	2	1	0

Bundan sonra 2 ədədinin hər bir rəqəmin nömrəsinə uyğun qüvvətini hesablayırıq:

$$2^8=256$$
$$2^7=128$$
$$2^6=64$$
$$2^5=32$$
$$2^4=16$$
$$2^3=8$$
$$2^2=4$$
$$2^1=2$$
$$2^0=1$$

Daha sonra hər bir rəqəmə uyğun hesabladığımız qüvvəti həmin rəqəmə vurub alınan hasilləri cəmləyirik.

$$\mathbf{1}*2^8 + \mathbf{0}*2^7 + \mathbf{0}*2^6 + \mathbf{1}*2^5 + \mathbf{0}*2^4 + \mathbf{1}*2^3 + \mathbf{0}*2^2 + \mathbf{1}*2^1 + \mathbf{1}*2^0 =$$

$$\mathbf{1} * 256 + \mathbf{0} * 128 + \mathbf{0} * 64 + \mathbf{1} * 32 + \mathbf{0} * 16 + \mathbf{1} * 8 + \mathbf{0} * 4 + \mathbf{1} * 2 + \mathbf{1} * 1 =$$

$$256 + 0 + 0 + 32 + 0 + 8 + 0 + 2 + 1 = \mathbf{299}$$

Nəticədə alınan ədəd verilmiş ikili ədədin onluq qarşılığı olar:

(b) **1 0 0 1 0 1 0 1 1 = (d) 299**

Ədədlərin hansı say sisteminə aid olduğunu bildirmək üçün qarşısında mötərizədə müvafiq say sistemi işarəsini qeyd edirlər. İkilik say sisteminin işarəsi **b** – binary, 10 -luq say sisteminin işarəsi **d** – decimal, 16 – lıq say sisteminin iaşrəsi isə **h** – hexal -dır. Digər nümunələrə baxaq:

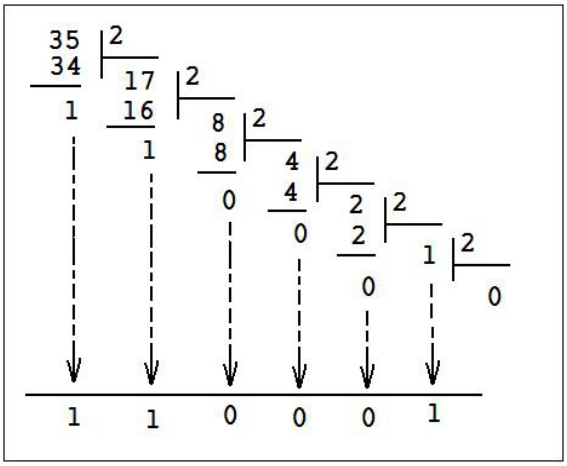
$$10010110 = 1*2^7 + 0*2^6 + 0*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 0*2^0 = 128 + 16 + 4 + 2 = 150$$

(b) 10010110 **=** (d) 150

Indi isə onluq ədədin ikili ədədə çevrilməsi ilə məşğul olaq.

Onluq ədədin ikili ədədə çevrilməsi

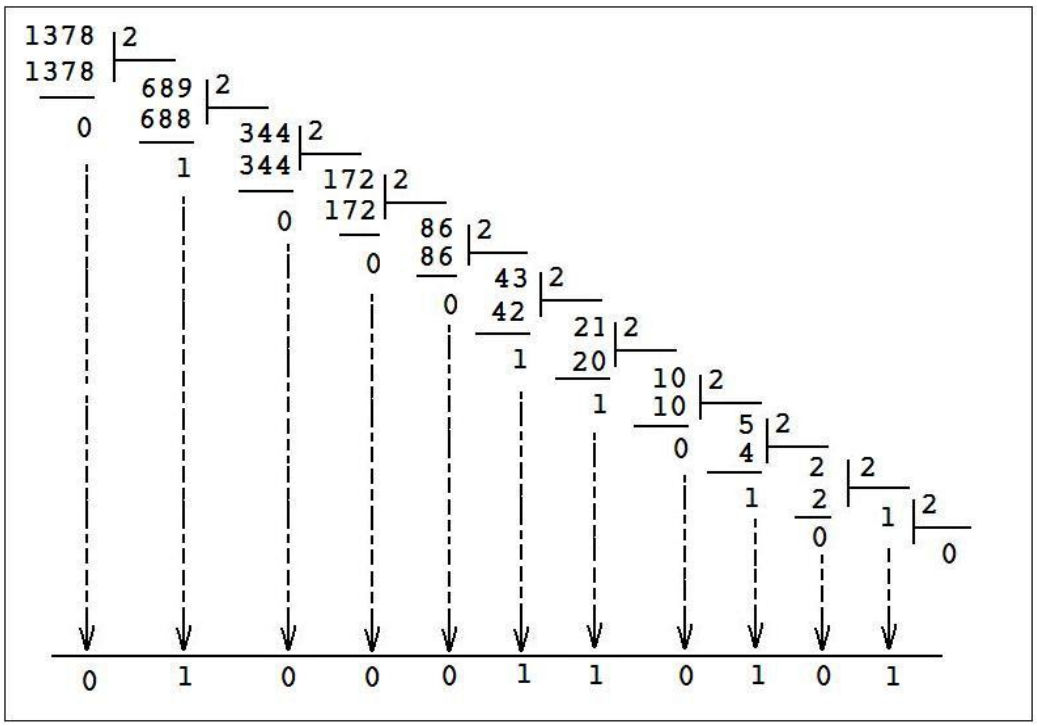
Onluq ədədi ikili ədədə çevirmək üçün aşağıdakı qaydadan istifadə edirik:
Onluq ədədi ikiyə bölürük, qalığı yadda saxlayırıq (qalıq 0 və ya 1 ola bilər), qismәti isə yenidən ikiyə bölürük. Yenidən qalığı yadda saxlayırıq və qismәti 2 -yə bölürük. Bu prosesi qismətdə 0 qiyməti alınana kimi davam etdiririk. 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**

(d) 35 = (b) 100011

Başqa bir misal, **1378** onluq ədədini ikili ədədə çevirək:



Qalıqları sondan əvvələ düzsək alarıq: **10101100010**

(d) 1378 = (b) 10101100010

Ikili ədədlərin toplanması

İkili ədədlər də onluq ədədlər kimi toplana, çıxıla, vurula və bölünə bilər. Biz sadəcə toplamanın qaydasını göstərəcəyik. Digər əməllər analoji qaydada aparılır. İkili ədədləri toplamaq üşün aşağıdakı qaydaları bilməliyik:

0 ilə 0-rı toplayanda 0 alırıq
1 ilə 0-rı toplayanda 1 alırıq
1 ilə 1-i toplayanda 0 alırıq 1 yadda qalır
1 ilə 1-i toplayanda və yadda 1 olanda 0 alırıq 1 yadda qalır.

İki ədədin müvafiq mövqedə yerləşən rəqəmlərin sağdan sola bu qayda ilə toplayırıq, onluq ədələrin toplanması qaydasına oxşar. Gəlin 0101 və 1101 ədədlərini toplayaq.

```
  0 1 0 1
+ 1 1 0 1
```

Sağdan sola yuxarıdakı qayda ilə bu ədədləri toplayaq. 1 ilə 1-i topladıqda 1 alınır, yadda bir qalır.

```
  0 1 0 1
+ 1 1 0 1
  ---------        (1 yadda)
        0
```

0 ilə 0-rı tolayırıq 0 eləyir üzərinə də yaddaşdakı 1-i əlavə edirik olur 1, yaddaşda heçnə qalmır.

```
  0 1 0 1
+ 1 1 0 1
  ---------        (0 yadda)
      1 0
```

Yenə 1 ilə 1 I toplayırıq 0 eliyir, yadda 1 qalır.

```
  0 1 0 1
+ 1 1 0 1
  ---------        (1 yadda)
```

```
  0 1 0
```

Nəhayət 0 ilə 1 -i toplayırıq 1 eliyir, yaddaşdakı 1 də üzərinə əlavə edirik olur 0 və yadda yenə 1 qalır.

```
  0 1 0 1
+ 1 1 0 1
  ----------        (1 yadda)
  0 0 1 0
```

Bütün ədədləri topladığımızdan yaddaşdakı 1-i yazırıq ən əvvələ.

```
  0 1 0 1
+ 1 1 0 1
  ----------        (1 yadda)
1 0 0 1 0
```

Beləlikə **0101** ilə **1101** ikili ədədlərinin cəmi **10010** -a bərabər oldu.

0101 + 1101 = 10010 (b)

Qeyd edim ki, əgər toplananalarda rəqəmlərin sayı eynbi deyilsə bu zaman rəməmlərinin sayı az olan ədədin əvvəlinə tələb olunan sayda **0** artırırıq. Misal üşüm **101010110** ilə **11011** -i toplasaq **11011** -in əvvəlinə 4 dənə **0** artırmalıyıq, **000011011**. Daha sonra onları yuxarıdakı qayda ilə toplaya bilərik.

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** rəqəmləri isə onluq sistemdəki **10, 11, 12, 13, 14** və **15** ədədlərinə uyğun gəlir. 16-lıq say sistemində ifadə olunan bəzi ədədlərlə tanış olaq:

af9900e5, 12, 4444a, ffffffff, 300dd1bfee

Proqramlaşdırmada adətən 16-lıq ədədlərin əvvəlinə **0x** işarəsi artırılır:

0x44, 0x666ffde21, 0x87aa

16 -lıq ədədin 10 -luq ədədə çevrilməsi.

16-lıq ədədlərin 10-luq ədədlərə çevrilməsi 2-lik ədədlərin çevrilməsinə oxşardır. Nümunə əsasında 16-lıq ədədin 10-luq ədədə çevrilməsi qaydası ilə tanış olaq. Tutaq ki, aşağıdakı 16-lıq ədədi 10-luq ədədə çevirmək istəyirik:

```
ff5a92ee
```

Əvvəlcə 16-lıq ədədi sağdan sola 0-dan başlayaraq nömrələyirik.

f	f	5	a	9	2	e	e
7	6	5	4	3	2	1	0

Daha sonra 16 ədədinin hər bir rəqəmin nömrəsinə uyğun qüvvətini hesablayırıq:

$$16^7 = 268435456$$

$$16^6 = 16777216$$

$$16^5 = 1048576$$

$$16^4 = 65536$$

$$16^3 = 4096$$

$$16^2 = 256$$

$$16^1 = 16$$

$$16^0 = 1$$

Daha sonra hər bir rəqəmə uyğun hesabladığımız qüvvəti həmin rəqəmə vurub alınan hasilləri cəmləyirik.

f	f	5	a	9	2	e	e
7	6	5	4	3	2	1	0

$$f * 16^7 + f * 16^6 + 5 * 16^5 + a * 16^4 + 9 * 16^3 + 2 * 16^2 + e * 16^1 + e * 16^0$$

= 15 * 268435456 + 15 * 16777216 + 5 * 1048576 + 10 * 65536 +

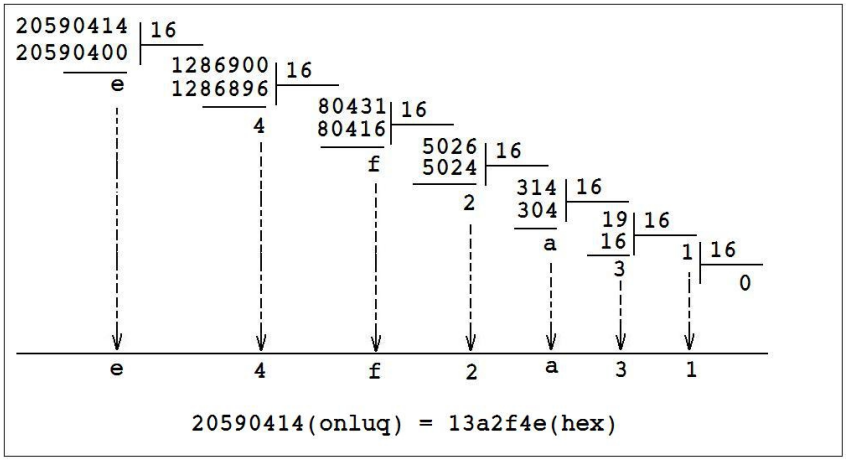
9 * 4096 + 2 * 256 + 14 * 16 + 14 * 1 = 4284125934

Beləliklə **(h) ff5a92ee = (d) 4284125934**

Burada hasili hesablayarkən 16-lıq simvolların 10-luq ədəd qarşılığından istifadə etdik.

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 ədədlərin toplanması.

16 -lıq ədədlər üzərində hesab əməlllləri 10-luq ədədlərə oxşardır. Bunun üçün aşağıdakı qaydadan istifadə etməliyik. Rəqəmləri cəmləmək üçün onların müvafiq 10-luq qarşılığından istifadə edirik. Əgər rəqəmlərin cəmi 15 -ə bərabər və ya aşağı olarsa müvafiq rəqəmi yazırıq və yadda heçnə saxlamırıq. Əks halda 15-dən böyük olduqda ondan 16 çıxırıq, nə qaldısa müvafiq yerə yazırıq və yadda bir saxlayırıq. Gəlin iki 16-lıq ədədin toplanması nümunəsi üzərində bu qaydalır ilə tanış olaq. Tutaq ki, **7fd41a** ilə **ad9e4c** 16-lıq ədədlərini toplamaq istəyirik.

```
  7 f d 4 1 a
+ a d 9 e 4 c
```

Sağdan sola bir -bir toplamağa başlayaq. İlk olaraq a ilə c -ni toplamalıyıq. a-nın 10-luq qarşılığı – 10, c -nin onluq qarşılığı isə 12-dir. Bu ikisinin cəmi 22 edir. Cəm 15 -i keçdiyindən cəmdən 16 çıxırıq alınır 6, yadda qalır 1.

```
  7 f d 4 1 a
+ a d 9 e 4 c
 ----------------------      (1 yadda)
                  6
```

1 ilə 4-ü toplayırıq edir 5, üzərinə yaddaşdakı 1-i əlavə edirik edir 6., yadda heçnə qalmır.

```
  7 f d 4 1 a
+ a d 9 e 4 c
 ----------------------      (0 yadda)
              6 6
```

Növbəti olaraq 4 ilə e-ni toplamalıyıq. e-nin 10-luq qarşılığı 14 olduğundan cəm 18 edir. 18 15 -dən böyük olduğuna görə ondan qaydaya uyğun olaraq 16 çıxırıq, qalır 2, yadda isə 1 olur.

```
  7 f d 4 1 a
+ a d 9 e 4 c
 ----------------------      (1 yadda)
```

```
      2 6 6
```

d ilə 9 -un cəmi 22 edir(d -ni 10-luq qarşılığı 13 bərabərdir). 22-nin üzərinə yaddaşdakı 1-i əlavə edirik olur 23. Daha sonra qaydaya uyğun olaraq 23-dən 16 çıxırıq alırıq 7 yadda isə 1 qalır.

```
  7 f d 4 1 a
+ a d 9 e 4 c
  ----------------------   (1 yadda)
    7 2 6 6
```

f ilə d-nin cəmi 28 olur, 1 -də yadda eliyir 29(f-in 10 -luq qarşılığı 15 -dir). 29 -dan 16 çıxırıq alırıq 13, hansı ki 16-lıq d-yə uyğundur, yadda isə 1 qalır.

```
  7 f d 4 1 a
+ a d 9 e 4 c
  ----------------------   (1 yadda)
  d 7 2 6 6
```

a ilə 7 -ni toplayırıq 17, üzərinə yaddaşdakı 1-i əlavə edirik 18. 18 -dən 16 çıxırq 2, yadda qalır 1.

```
  7 f d 4 1 a
+ a d 9 e 4 c
  ----------------------   (1 yadda)
  2 d 7 2 6 6
```

Sonda yaddaşdakı 1- ən əvvılı yazırıq.

```
  7 f d 4 1 a
+ a d 9 e 4 c
  ----------------------   (1 yadda)
1 2 d 7 2 6 6
```

Beləliklə, alırıq:

7fd41a + a99e4c = 12d7266

Suallar

1. Hansı say sistemlərini tanıyırsınız?
2. İkili ədədlər hansı rəqəmlərlə ifadə olunur?
3. 16-lıq say sistemindəki d rəqəmi 10-luq qarşılığı neçədir ?

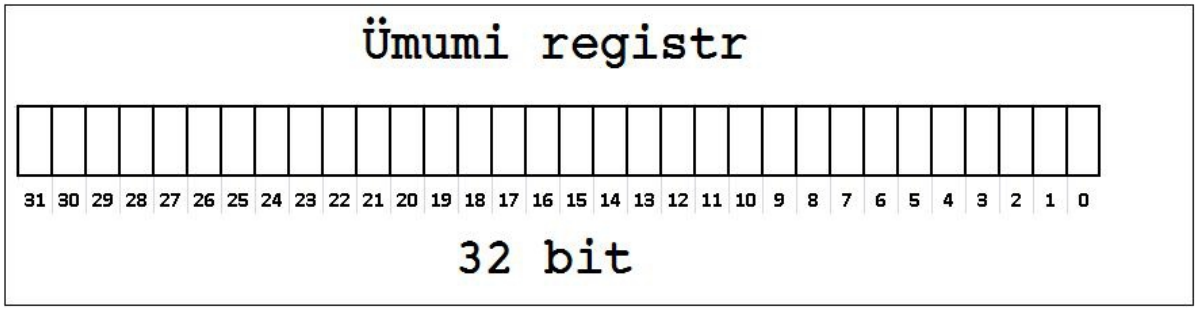
Çalışmalar

1. 101011 ikili ədədini 10-luq ədədə çevirin.
2. 234 10-luq ədədinin ikilik qarşılığını tapın.
3. 0100101 ilə 110101 ikili ədədlərinin cəmini hesablayın.
4. 0x645dda ədədinin 10-luq qarşılığını tapın.
5. 56785 onluq ədədinin 16-lıq qarşılığını tapın.
6. 0x878a ilə 0x76bb4 16-lıq ədədlərinin cəmini hesablayın.

7

Bit əməliyyatları

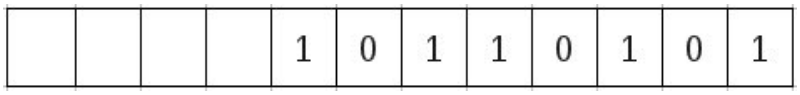
Biz qeyd elədik ki, məlumatın ölçü vahidi bayt -dır. Lakin bəzən verilmiş baytı təşkil edən bitlər üzərində hansısa əməliyyatlar aparmaq, o cümlədən müxtəlif bitlərin qiymətini örgənmək, dəyişmək, sola - sağa sürüşdürmək v.s. işlər görmək lazım gəlir. Nümunə testlərimizi **%rax** reqistrinin ilk yarısını təşkil edən **%eax** reqistri üzərində verək. Bilirik ki, **%eax** registrinin ölçüsü **4** baytdır, başqa sözlə **32** bit. Reqistrin bitləri sağdan sola 0 -dan başlayaraq nömrələnir.



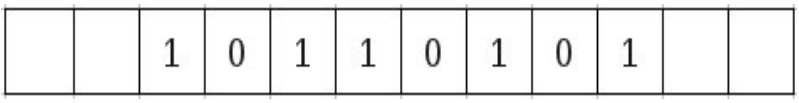
Ən məşhur bit əməliyyatlarına bitlərin cəm halda sağa və ya sola sürüşdürülməsini misal gətirmək olar.

Sürüşmə əməliyyatları
Sola sürüşmə

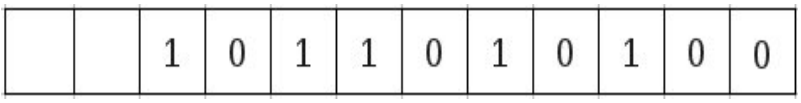
Verilmiş bitlər ardıcıllığını (hər hansı reqistr və ya yaddaş sahəsini ifadə edən bitlər) sola sürüşdürərkən bütün bitlər cəm şəkildə olduğu kimi göstərilən vahid sola sürüşür.



Bu bitlər ardıcıllığını 2 vahid sola sürüşdürək:



Boş qalan yerlərə 0 yazılır, aşağıdakı kimi:



Bitləri sola sürüşdürmək üçün **shll** instruksiyasından istifadə olunur. **shll** instruksiyasının istifadə qaydası aşağıdakı kimidir:

shll say, operand

Nəticədə operand ilə verilən bitlər ardıcıllığı, bu yaddaşda müəyyən sahə və ya hər-hansı reqistr ola bilər, göstərilən sayda sola sürüşər. Misal üçün **%eax** reqistrinin bitlərini **4** vahid sola sürüşdürən kod aşağıdakı kimi olar:

shll $4, %eax

Sağa sürüşmə sola sürüşməyə analojidir.

Bul cəbri əməliyyatları
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 **01101111100001011** 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 YA 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

Bir çox hallarda verilmiş bitlər ardıcıllığının bu və ya digər mövqedə yerləşən bitinin qiymətini (0/1) örgənmək tələb olunur.

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
00000000000000000100000000000000
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.

Qısa icmal

Biz bu kitabda assembler dilində proqram tərtibi, dəyişənlər, ünvanlar, yaddaşa müraciət üsulları, stekin iş prinsipi, funksiyaların çağırılması, geri qayıtması mexanizmləri, funksiyalara parametr ötürmə v.s. ilə tanış olduq. Bu biliklər hər-hansı assembler kodunun işini təhlil etmək üçün kifayət edər, lakin assemler bununla bitmir. Əksinə bütün bunları böyük bir yolun başlanğıcı hesab etmək olar. Söhbət sistem proqramlaşdırmadan gedir. Bu kitabda daxil olan instruksiya və reqistrlər istifadəçi proqramlaşdırmada istifadə olunur. Lakin prosessorun işini idarə edən, müxtəlif digər instruksiya və reqistrlər mövcuddur, hansı ki bu instruksiyaları yalnız sistem proqram kodları yerinə yetirə bilər. İstifadəçi proqramlarına sistemin işinə təsir göstərə biləcək hər-hansı instruksiya icra eləmək və hər-hansı sistem əhəmiyyətli məlumat yerləşən reqistr və ya yaddaşa müraciət etmək icazəsi verilmir. Bütün bu işləri prosessor108 tənzimləyir. Ümumiyyətlə sistem proqramlaşdırma örgənməli məsələləri qısa olaraq aşağıdakı kimi sadalamaq olar: arxitektura , sistemin yüklənmə prosesi, kəsilmələr, yaddaşın idarə olunması, fayllar sisteminin idarə olunması, proseslər(icra olunan proqramlar), şəbəkə , proseslərarası əlaqə vasitələri, sinxronizasiya v.s. C və Assembler dillərini mükəmməl bilənlər üçün Unix proqramlaşdırmanı örgənəyi məsləhət görürəm. Daha sonra açıq kodlu Linux nüvəsini örgənməyə başlamaq və yuxarıda sadaladığım məsələlərin C və assembler dillərində necə realizasiya olunması ilə tanış olmaq olar. Sistem proqramlaşdırmanı Windowsda örgənməyə başlamağı məsləhət görmürəm, öz təcrübəmnən.

Biraz müəllif barəsində:

Mən uzun müddətdir ki, (təxminən 5-6 il) Linux nüvəsi proqram kodlarını örgənirəm. Son 3 ildə paralel olaraq Diskret Riyaziyyatı örgənməyə başlamışam. Məqsədim Linuxun hər-hansı alqorimtlərindən birinin riyazi modelini qurub optimallığını tədqiq etməkdir. Hal-hazırda CFS alqoritmini örgənirəm – Complete Fair Scheduler. Növbəti mərhələlərdə Fayllar sistemi, yaddaşın idarə olunması və Sinxronizasiya alqoritmlərini də nəzərdən keçirməyi planlaşdırıram. Bunlar yaxın gələcəyə planlaşdırılan işlərdir, nisbətən sonrakı planlarda isə süni intellekt, paralel icraolunma, qarışıq məntiq v.s. kimi mövzuları yer alır.