# *Intel*i *x86* 32-bitine masinkood

*Intel* *x86-*instruktsiooni normaalformaat on põhimõtteliselt järgmine:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| pikkus | kood | d | w | mode | reg | R/M | disp |

0, 1 2..5 6 7 8..9 10..12 13..15 16..

Numbrid vektori all näitavad osiste algus- (ja lõpu)bitte. Meenutagem, et *disp* ei pruugi käsus osaleda, ja kui osaleb, siis on ta pikkus kas 8 või 16 bitti. Ja *x86* masinkoodi käsu pikkus on 2 kuni 4 baiti (16 kuni 32 bitti).

Näiteks valisime näiliselt lihtsaima direktiivi *mov* masinkoodi-ekvivalendi; *mov kuhu, kust*[[1]](#footnote-1). Sisuliselt on see lihtsaim instruktsioon, masinkoodis aga üks vähestest (kui mitte ainuke), kus kasutatakse kõiki adresseerimisvõimalusi. Kasutame järgnevas materjale [Virginia] ja [] saitidelt. Võimalikud operandid on *register → register*, *mälu → register* ja *register → mälu*, ent mitte *mälu → mälu* (mida *Intel* põhimõtteliselt ei võimalda)*.* Selle direktiivi masinkoodi-ekvivalendi pikkus on kaks, kolm või neli baiti. Kolm või neli baiti on käsk pikk siis, kui di-rektiivi mäluaadressi-komponendis kasutatakse *nihet*[[2]](#footnote-2)mäluaadressi absoluutväärtuse suhtes või on kasutatud *vahetut operandi*.

Vaadelgem *mov-*masinkoodi vasakult paremale, alustame vasakpoolseimast baidist[[3]](#footnote-3).

Esimese baidi kaks vasakpoolseimat bitti (0 ja 1):

* 00 -‒ instruktsioonis osaleva registri(osa) omapikkus on 8 bitti (näit. *AL*),
* 01 ‒ 16 (näit*. BX*) või
* 10 ‒ 32 bitti (näit. *ECX*), so, mitu bitti näidatud registrist käsus osaleb.

Bitid 2..5 on kood *mov =*  0010[[4]](#footnote-4). Kuues bitt on *destination* (*d*), suund:

* 0 on kirjutamine mällu (näit. *mov [n],eax*) ja
* 1 ‒ registrisse (näit. *mov eax,0*).

Parempoolseim (7.) bitt esimeses baidis (*w*) näitab, kuidas tuleb interpreteerida järgmise (teise) baidi koodi välja *reg* (pikkus 3 bitti, suhtpositsioonid 2..4).

Kui *w*=0, siis on tegemist 8 (x*L*)- või 16-bitise (xX) välja adresseerimisega ja *reg* kodeerib:

* 000: *AL* või *AX*
* 001: *CL* või *CX*
* 010: *DL* või *DX*
* 011: *BL* või *BX*
* 100: *AH* või *SP*
* 101: *CH* või *BP*
* 110: *DH* või *SI*
* 111: *BH* või *DI*

Kui *w*=1, siis on tegemist 32-bitise adresseerimisega ja *reg* kodeerib:

* 000:  *EAX*
* 001: *ECX*
* 010:  *EDX*
* 011: *EBX*
* 100: *ESP*
* 101: *EBP*
* 110: *ESI*
* 111: *EDI*

Teise baidi kaks vasakpoolseimat bitti on *mod*, mis näitavad, kuidas tuleb operandiks olevat mäluaadressi töödelda, ning toda töötlust täpsustavad teise baidi kolm viimast bitti (*R/M*)[[5]](#footnote-5). Kui *mod* on

* 00, siis operandi mäluaadress ongi tegelik, so. nihet pole. Aga, kui *R/M* = 110, siis nihe[[6]](#footnote-6) toimib.
* 01: nihe on olemas.
* 10: käsku indekseeritakse, võimalik on nihe.
* 11: mõlemad operandid on registrid, ei mingeid nihkeid. Masinkoodi-käsk on kahe-baidine.

Teise baidi kolm viimast bitti on *R/M* . Komponent *disp* on *nihe*[[7]](#footnote-7). *R/M* (*Register*/*Memory*) on

*register-register-*variandi puhul teine register, muidu aga

* 000: [BX+SI] või nihe [BX]+[SI]
* 001: [BX+DI] või nihe [BX]+[DI]
* 010: [BP+SI] või nihe [BP+SI]
* 011: [BP+DI] või nihe [BP+DI]
* 100: [SI] või nihe [SI]
* 101: [DI] või nihe [DI]
* 110: ainult nihe[[8]](#footnote-8) või nihe [BP]
* 111: [BX] või nihe [BX]

Näiteks, direktiivi *mov ebp,esp* kood on 89E5. Kirjutame selle bitikaupa lahti ja eraldame komponendid punktidega:

10.0010.0.1.11.100.101

Vasakult paremale:

* 10: täis-32-bitised registrid
* 0010: *mov*

0: register-register variandis tähenduseta(?)

* 1: 32-bitised operandid
* 11: register-register
* 100: *esp*
* 101: *ebp*

Magasinikäskude *push*, *pop*, *call* ja *ret*  formaadid erinevad tavaformaadist. Näiteks, *push ebp* kood on ühebaidine 16-ndarv 55=50+*rd[[9]](#footnote-9), EBP* kood *rd* on 5 ̶ vt. *w=*1 ja *reg=*101 [push].

# Kompilaator, komplekteerija ja paigaldaja

Programmeerimiskeelte realiseerimisel on kaks võimalust: tehakse kas interpre-taator, mis programmi analüüsi puu (tavaliselt tsüklilise) läbimise käigus seda programmi täidab või kompilaator, mis tolle puu ühekordsel läbimisel generee-rib teises, tavaliselt madalama taseme keeles resultaatprogrammi. Tänapäeval on „resultaatkeel“ kas C või objektarvuti assembler; kui esmane väljund on C-tekst, siis jätkab C-kompilaator, mis genereerib assembler-teksti. Arvutis täidetava programmi (*exe-*faili) saamiseks töötab assembleri translaator, mille väljund on vahekeelne objektfail, mida tuleb veel komplekteerija (*linker*) ja paigaldaja (loader) poolt modifitseerida.

Niisiis, programmeerides kompileeritavas kõrgtaseme keeles on programmi realiseerimise (.exe-faili tegemise) vältimatu vaheetapp masinkoodi kompileeri-mine assemblerist. Ja assembleri esmane väljund on vahekeelne objektfail.

Meie raamatu assembler-keel on NASM, mis töötab *gcc* keskkonnas. Viimane toetab mitut erinevat objektfaili-formaati, sh. *Windows*i operatsioonisüsteemile orienteeritud formaati *win32*, *Linux*i *elf32[[10]](#footnote-10)* või UNIXi *coff[[11]](#footnote-11).* Joonisel x.y (allikas: [x86asm])on esitatud assembleri, linkeri ja paigaldaja üldine skeem.



Joonis x.y. Asm-faili(de)st protsessori jaoks interpreteeritava masinkoodini.

Objekt- ja täidetava (*executable*) faili struktuurid on põhimõtteliselt sarnased ja nendega tutvumine pakub omaette huvi, ent paraku tutvustamine ei mahu nen-de kaante vahele[[12]](#footnote-12). Huvilistele soovitame näiteks *Wei Wang*i netimanuaali [Wang].

# Intel x86 mälujaotus

Siin jaotises esitame artikli[[13]](#footnote-13) „Memory Layout of C Programs“ [geeks] referee-ringu. Ehkki tolle pealkiri viitab üheselt C programmi lahendusaegsele mälujao-tusele, on pilt samasugune ka NASMi puhul; sisuliselt on C mälupilt dikteeritud NASMi omast, viimase määrab aga *x86-*le orienteeritud Windowsi kaitserežiimi-le (protected *mode*) tuginev mälujaotus.

Niisiis, tavapärane lahendusaegne operatiivmälu on jaotatud järgmisteks osa-deks (segmendid, sektsioonid):

1. „text“ – täidetav programm, so. masinkoodi käskude jada (NASMis *section .text*), juurdepääs on ainult lugemiseks (*Read Only, RO*);
2. väärtsustatud (*initialized*) andmed (NASMis *.section .data*) Juurdepääs nii lugemiseks kui ka kirjutamiseks (*RW*)[[14]](#footnote-14);
3. algväärtsustamate (*uninitialized*) andmed (NASM: *section .bss*) ̶ *RW*;
4. magasin (*stack*), mis toimib aparatuurselt, so. protsessori tasemel ning on näiteks *C*-programmeerija jaoks peidetud, assembleris programmeerijale aga vältimatu piirkond (*RW*);
5. kuhi (*heap*). Rakendusprogramm saab sellest piirkonnast mälu küsida, C-s ja NASMis funktsiooni *malloc* abil. Juurdepääs on *RW.*

Programmi lahendusaegse mälupidi esitame originaalis (vt. joonis x.1). Joonisel esitatu tõlge ja kommentaarid: pildil on väikseim mäluaadress all ja suurim üle-val. Näeme, et kaitserežiimis on kasutatava mälupiirkonna alguses täidetav programm ja selle järel *.data-*sektsioon ning et need laaditakse mällu *.exe-*faili paigaldamisel. Sama töö käigus eraldatakse mälu *.bss-*segmendile (see on paigaldamisel täidetud nullidega).

Järgmine piirkond on mõeldud *kuhjale* ja see piirkond saab kasvada aadresside suurenemise suunas.


Joonis x.1 Lahendamisaegne mälujaotus.

Programmi täitmisaegse mälu lõpus on käsurea-argumendid – meenutame C-keele *main-*mooduli kirjelduse vastavat varianti

Int main(int argc,char \*\*argv)

Ja kui käsureal on programm *vernam* välja kutsutud kui

>vernam bender.txt GSVernam.jpg

siis C-programm saab käsurea-andmed kätte järgmiselt:

argc=3

argv=→0)→vernam’\0’

 1)→bender.txt’\0’

 2)→GSVernam.jpg’\0’

Eelmine piirkond on aparatuurse magasini[[15]](#footnote-15) jaoks. Magasin „kasvab“ aadresside vähenemise suunas. Teoreetiliselt on võimalik, et „vastukasvav“ kuhi ja aadresside mõttes „vastukahanev“ magasin saavad kokku; sel juhul võivad uue-mad lahendused kasutada virtuaalmälu, kui aga mitte, siis tagastab *malloc* NULL-viida (vaba mälu pole), *push* aga annab signaali *stack overflow.*

# NASM-programmi osad

NASMis kirjutatud programmi struktuur on sisuldasa sarnane C-programmi omaga:

* Esimesele reale on heaks kombeks kirjutada kommentaar: teksti nimi, otstarve, kirjutamise kuupäev ja kirjutaja. Kui C kommentaari tunnus on kas /\* või // (esimene kehtib kuni paarini \*/, teine reavahetuseni), siis NASMi kommentaari mõjuulatus on nagu C ’//’ omal; kommentaar algab semikooloniga ’;’.
* C-s programmeerides tuleb seejärel kirjutada makrod, tavaliselt ja välti-matult vajalike moodulite kirjeldusteekide deklaratsioonid #*include<...h>* ja – kui see on otstarbekas, ka kasutaja-makromäärangud *#define.* NASMis pole vaja (ega võimalikki) deklareerida standardteeke (%include on teksti kopeerimiseks kasutatav – sellest hiljem), ent *#define* võimalusi pakub *%define.*
* C *#include-*teekide moodulid tuleb NASMis ükshaaval deklareerida neid *välisnimedeks* (*extern*) kuulutades, lisades nimele prefiksi ’\_’. Näiteks, kui kasutame C *stdio-*teegist moodulit *printf*, siis NASMis tuleb kirjutada *extern \_printf*. Oma programmi *main-*moodul (*exe-*faili sisendpunkt) tuleb samas deklareerida kui *global \_main.*
* C-tekstis võib makrodele järgneda (ja mooduli(te) teksti(de)le eelneda) osa, kus kirjeldatakse globaalseid muutujaid ja andmestruktuure ning võib eeldeklareerida mooduleid. NASMis moodulite eelkirjeldusi pole (nende järjekord tekstis on suvaline), küll aga saab kirjutada üks või kaks „sekt-siooni“: üks globaalsete konstantide ja eelväärtustatud muutujate jaoks – see on *section .data[[16]](#footnote-16)* – ja teine globaalsetele muutujatele – *section .bss.[[17]](#footnote-17)*
* *.data*-sektsioonis (*section .data* või – samaväärselt – *segment .data*) saab kasutada instruktsioone kujul

<märgend> [*times n*] *d*<pikkus> <väärtus(ed)>

’pikkusel’ on järgmised variandid:

*b* – *byte*, bait, C-keele *char*,

*w –* *word*, 2-baidine arv, C *short*,

*d – double word*, 4-baidine arv, C *int* või *float*,

*q – quad word*, 8-baidine ujupunktarv, C *double*

*t – ten bytes*, 10-baidine ujupunkt- või pakitud kümnendarv.

Mõned näited:

*Bee db ’b’* ;C: *char Bee=’b’;*

*Tere db ’T’,“e“,“r“,’e’,0 ;C: char Tere[5]=“Tere“;*

*Tere db ’Tere’,0 ;*sama,mis *Tere*. Lõpu-null on stringi terminaator.

*Halloo db ’Hallo world!*’,*10,0 ;C: Halloo char[]=“Hallo world\n“); ASCII 10* on C \n.

*Sada dd 100*  C: *int Sada=100;*

*Vektor dd 1,3,5,7*  C: *int Vektor[ ]={1,3,5,7};*

’*times n’* on fakultatiivne võimalus algväärtsustatud vektori deklareeri-miseks. Näiteks:

*V times 65 db 0* C: *char V[65]={’0’};*

Niisiis, algväärtus võib olla kas tekstiline, kas paari “..“ või ’..’ vahel, või arvuline, mida saab esitada ka kahend-, kaheksand- või kuueteistküm-nendarvuna (näit. 25, 11001b, 31o või 19h).

* *.bss-*sektsioonis saab muutujatele mälu reserveerida ilma võimaluseta neid algväärtustada. Sõltub realisatsioonist, ent tavaliselt on paigaldatud programmis neil 0-väärtused. Instruktsioonide kuju on:

<märgend> *res*<pikkus> <arv>

’pikkus’ on sama, mis .*data-*sektsiooni *d* puhul ning *arv* näitab, mitmele antud tüüpi elemendile ruum reserveeritakse. Näiteid:

*a resb 1* C: *char a;*

*b resd 10* C: *int b[10];*

* Sektsioonis *.text[[18]](#footnote-18)* on assembleri direktiivid, mis „tõlgitakse“ masinkoodi käskudeks. *Intel x86* on üheaadressiline masin: käsukoodi formaat on kas

*Kood register mäluaadress* (käsus on 1 aadress) või

*Kood mäluaadress register* või

*Kood register register* (käsus on 0 aadressi) või

*Kood register vahetu\_operand*  (samuti 0 aadressi) või

*Kood register* (näit. direktiivides *inc* ja *dec* (täisarv + 1, täisarv – 1) või

*Kood mäluaadress* (näit. *loop <etikett>*)või

*Kood* näit. *ret*.

Käskudes adresseeritav mäluväli on kas .*data-* või *.bss-*sektsioonis, suuna-miskäskude puhul on mäluaadressi rollis selle käsu aadress, kuhu antakse juhtimine. Assembleris kasutatakse mäluaadressidena *etikette* (C-keeles märgendeid). NASMis on etiketi väärtuseks viit antud objektile. Edasi anda saab nii viitu (aadresse) kui ka viidatud väärtusi (arvu vm. aadressilt) ning kirjutada saab viida järgi, so. aadressile. Keele tasemel tuleb neid seiku käsus ilmutatud kujul näidata: *nimi* on aadress ja [*nimi*] on miski sel-lel aadressil.

Kui *nimi* on seotud vektoriga, siis selle elemendid on indekseeritavad. Erinevalt C-st, kus *a[++i]* nihutas elemendi aadressi baitvektori korral 1 baidi ja *int-*vektori puhul 4 baidi võrra, tuleb assembleris indeksi samm ilmutatud kujul ise näidata.

Lisaks sulupaarile ’[’ ja ’]’ on NASMis veel üks metakeelne sümbol ’$’: selle käsu aadress, kus ta aadressosas esineb. Näiteks, lõigu

*Jutt db ’see on pikk jutt’*

*L equ $-jutt*

toimel omistab tranlaator muutuja *L* väärtuseks 11.

Mäluvälja operandina kasutamisel tuleb arvestada seigaga, et translaator „ei vaata“ ta kirjeldust andmesektsioonis (*.data, .bss*) ning käsus tuleb reeglina näidata operandi pikkus, kas bait, 2 või 4 baiti, vastavalt *byte, word* ja *dword.* Miks nii: näiteks, *.data-*sektsioonis võime deklareerida stringi baithaaval ja samamoodi võime kasutada selle stringi suvalist sümbolit omaette operandina. Kui operandiks on kogu string, siis kasuta-me neljabaidilist viita stringile (*dword*).

# Registrid

Arvuti protsessor (*CPU ̶* *Processing Unit*) koosneb jämedalt võttes juhti-misseadmest ( *CU*  ̶ *Control Unit*), aritmeetika-loogikaseadmest (*ALU ̶ Arithme-tic Logic Unit*) ja protsessori registritest.

Johtuvalt meie raamatu suunitlusest (32-bitine *Windows*ile orienteeritud *NASM*) tutvustame ainult „vajalikke“ registreid.



Joonis x.y. *NASM*is kasutatavad registrid [Yale].

Joonisel x.y näidatud registritest 6 esimest on üldotstarbelised (eriülesanded on allpool juurde märgitud):

* AL/AH/AX/EAX: Akumulaator, tehete operandi ja funktsiooni resultaadi tagastamise rollis;
* BL/BH/BX/EBX: Baas(i index) tööks vektoriga;
* CL/CH/CX/ECX: Tsükliloendaja (*counter*);
* DL/DH/DX/EDX: kombineeritakse EAX-iga korrutamis- ja jagamistehete tegemiseks (*data*);
* ESI: Lähteandmete vektori indeks (*Source index*);
* EDI: Resultaatide vektori indeks (*Destination index* );

Spetsiifilise rolli kõrval on üldotstarbelised registrid kasutatavad “tavatöödeks”. Registrid ESP (magasini tipu viida jaoks) ja EBP (aktiivse mooduli “*freimi*” baasi fikseerimiseks magasinis) on kasutatavad ainult sihtotstarbeliselt.

Joonisel pole kaht registrit.

* EFLAGS on ühebitiste signaalide jaoks. Enamus neist on protsessori oma-kasutuses või dokumenteerimata, rakendusprogrammeerijale olulised lipud on need, mida heisatakse käskude resultaadina. Ainuke programselt muudetav lipp on *DF* (*destination flag*), mille väärtus ‘0’ näitab, et stringi töödeldakse vasakult paremale, ja ‘1’ vastupidises suunas. Ülejäänuid saab rakendusprogrammides kasutada tingimusoperaatorites, näiteks

*ZF* (*zero*, null): 1, kui tehte resultaat = 0;

*SF* (*sign*, märk): 1 kui tehte resultaat < 0;

*OF* (*overflow*): 1, kui tehte resultaat ei mahu selleks määratud väljale.

* EIP: Instruction pointer. Järgmisena täidetava käsu aadress. Kaitserežiimis pole see register kasutajaprogrammidele kättesaadav. Märkigem, et ma-sinkoodi-käsud jagunevad klassidesse, nimedega *ring0* kuni *ring3.* 0-ringi käsud on kättesaadavad protsessori tuuma (*kernel*) programmeerijaile, nemad töötavad seal, kus tehakse protsessoreid ja neile pole mingeid piiranguid. Ning mida suurem on “ringi” number, seda rohkem võimalusi on varjatud. Tavaliselt kasutatakse *ring1-*privileege draiverite program-meerimiseks, *ring2* on sama tavaliselt täpsemalt piiritlemata ning *ring3* on rakendusprogrammeerijate päralt. Võimaluste peitmine johtub julge-olekukaalutlustest: tuumaprogrammid ei tohi arvutit “kinni jooksutada” ja rakendusprogrammidel on see silumise etapil tavaline, et nende töö päädib avariiga, ent arvuti töötab normaalselt edasi.

# Magasin (*stack*)

Aparatuurne magasin on *LIFO* (*Last In First Out*, viimasena sisse ̶ esimesena välja)-tüüpi. Sel printsiibil töötavad näit. (pool)automaatrelvad: padrunipide-messe või salve esimesena lükatud padrun teeb pauku viimasena ning viimasena salvestatud padrun esimesena. Programmeerimisse tõi magasinprintsiibi saks-lane *Friedrich Ludwig Bauer* [idsia].



Joonis x.y. *Friedrich Ludwig “Fritz” Bauer* (1924 ̶ 2015).

Assembleridon põhimõtteliselt orienteeritud toetama moodulprogrammee-rimist. Moodul pole kuskil täpselt defineeritud, ent programmeerimises on see tavalselt võimalikult lühike, ainult ühte tööd tegev, täpselt fikseeritud sisendi ja väljundiga (alam)programm, mis võib välja kutsuda teisi mooduleid, sh. ise-ennast. Assemblerid arvestavad neid seiku ning moodulite poole pöördumiseks ning nendega info vahetamiseks on mitmeid mooduseid; *NASM* kasutab neist varianti *cdecl[[19]](#footnote-19)* [wcall]. Nimi on lahti kirjutatult *C declaration* ̶ variant, mida kasutavad *x86 C-*kompilaatorid. Alamprogrammi argumendid pannakse magasi-ni, *int-* või viida-tüüpi resultaadid tagastatakse EAX-registris. Väljakutsuv moodul peab arvestama, et alamprogramm võib rikkuda registrite EAX, ECX ja EDX sisu. Ja magasini „puhastamine“ väljakutsutud mooduli argumentidest on väljakutsu-ja töö.

Aparatuurse magasini „laius“ (elemendi pikkus) on alati 4 baiti. Vaatleme allpool *cdecl-*versiooni olemust.

* Magasini tipu viit on alati registris ESP. Seda modifitseerivad käsud *push* (pane magasini) ja *pop* (loe ja eemalda magasinist). Sisuldasa on need direktiivid lahti kirjutatavad nii:

*push ebp*:

sub esp,4 ;nihutan magasini viita

 mov [esp],ebp ;registri ebp sisu → magasin

*pop ebp:*

 mov [ebp],esp ;magasini tipust viit esp registrisse ebp

 add esp,4 ;”kustutan” magasini tipmise elemendi

Ja samuti nagu *push* ja *pop* nihutavad magasiniviita *esp*, saab seda teha vajadusel ka *NASM-*programmis, direktiividega *sub* ja *add.*

* Olgu alamprogrammi *pluss* argumendid arvud 2 ja 3. (Kirjeldus *C-*s: *int pluss(int x,int y);*) Pöördumine *NASM*is (*x=*2 ja *y=*3):

 push 3 ;sub esp,4 mov dword[esp],2

 push 2 ;sub esp,4 mov dword[esp],3

 call pluss

 add esp,8 ;”eemaldan” argumendid ‘2’ ja ‘3’ magasinist

*call pluss* on lahti kirjutatult selline:

 push eip ;naasmisaadress (käsu *add esp,8* oma)

jmp pluss ;juhtimine etiketile ‘pluss’

* Alamprogrammi 2 esimest käsku on

push ebp

mov ebp,esp

Registris EBP on mooduli privaatse magasiniruumi baasi (algusaadressi) viit; selle privaatruumi originaalnimetus on ‘freim’ (*frame*). Nende kahe käsuga salvestatakse aktiivseks saanud mooduli väljakutsuja freimi baas-aadress magasini (lahtikirjutatult: sub esp,4 mov[esp],ebp) ning väl-jakutsutud mooduli freimi baasaadressiks registris ebp saab viit magasini tipule (mov ebp,esp). Freimi baas on omamoodi “püsipunkt”:

[ebp] = freimi baas;

[ebp+4] = naasmisaadress;

[ebp+8] = 1. argument;

[ebp+12]= 2. argument

jne.

* Käsurea-parameetrid edastatakse juhtprogrammile (*main-*moodulile) sa-mamoodi nagu seda teevad tavamoodulid. *C-*tekstis on sel juhul

Int main(int argc,char \*\*argv)

ning *NASM*is

[ebp+8] = argc;

[ebp+12]=argv-viit parameetrite vektorile.

* *Cdecl-*kokkuleppe kohaselt “pušitakse” tegelikud parameetrid magasini järjekorras “paremalt vasakule” ─ kui peame silmas nende järjekorda *C*-tekstis, eelmise näite ekvivalent *C-*s on *pluss(2,3).* Nii on argumendid magasinis “ülalt alla” samas järjestuses nagu *C-*s “vasakult paremale”.
* Erinevalt *C-*programmist pole *NASM*is kohta lokaalsetele muutujatele ning kui neid vaja on, siis saab neile ruumi võtta magasinist, oma freimist ning neid saab adresseerida freimi baasi suhtes. Näiteks, kui *C* alam-programmis on lokaalsed muutujad *int a,b;* siis saame *NASM*is kirjutada teksti alguse nii:

push ebp

mov ebp,esp

sub esp,8 ;[ebp-4]=a ja [ebp-8]=b

Kui tahame oma lokaalseid muutujaid *NASM-*programmis “nimepidi kut-suda”, siis võime moodulit (näiteks *P*, nonde a ja b-ga) alustada nii:

P:

%define a dword[ebp-4]

%define b dword[ebp-8][[20]](#footnote-20)

push ebp

…

Enne alamprogrammist väljumist peame magasini oma töömuutujatest puhastama, meie näite puhul käsuga

add esp,8

* Alamprogrammist väljumise standardsed käsud on

pop ebp ;taastame väljakutsuja freimi baasi

ret ;sisuliselt: pop eip

Ent kuivõrd eip-registrit kasutajaprogrammi jaoks pole, siis võime ret-käsku imiteerida näit. nii:

mov ecx,[esp]

add esp,4

jmp ecx

* Väljakutsutud moodul võib oma vajadustele vastavalt kasutada kõiki üld-registreid, ent *cdecl-*kokkuleppe järgi on hea toon säilitada väljakutsuja jaoks registrite ebx, esi ja edi seis. Kui me neid registreid alamprogrammis kasutame siis tuleb nad enne kasutamist (tavaliselt alamprogrammi alguses) magasini hoiule panna ja enne väljumist taastada.

Näiteks:

ap:

push ebp

mov ebp,esp

sub esp,8 ;ruum kahele lokaalsele muutujale

push ebx

push esi

push edi

…

pop edi

pop esi

pop ebx

add esp,8 ;lokaalsete muutujate ruumi vabastamine

pop ebp

ret

* Siinkohal on sobiv aeg hoiatuseks: tsükliloendaja (et kasutada tsüklidirek-tiivi kujul *loop märgend*)tuleb kanda ecx-registrisse, ja kui tsükli sees on pöördumine mõne *crt-*mooduli[[21]](#footnote-21) poole, siis tuleb olla valmis selleks, et see moodul rikub ära ecx-registri ja koos sellega ka meie tsükli. Selle välti-miseks tuleks kirjutada näiteks nii:

ring:

 push ecx

 …

pop ecx

loop ring

* Rõhutame: väljakutsutud moodul (*callee*) peab enne juhtimise tagasta-mist väljakutsujale (*caller*) “puhastama” kogu oma freimi. Kui me prog-rammeerides jätame magasini midagi ülearust, siis pole *ret-*käsu täitmise ajal magasini tipus naasmisaadressi ̶ seal on midagi muud ̶ ning prog-ramm lõpetab avariiliselt.

# **Operandid**

Assemblerdirektiivi operandideks on kas üks või kaks registrit, (indekseeritud) mälu või vahetu operand.

Üldine reegel on, et *X* tähistab registrit (ecx, edi) või mäluaadressi (viita, nt. *n*) ning *[X]* tähistab registris või mäluaadressil olevat *väärtust.* Allpool toome näi-teid just selleks otstarbeks kirjutatud programmist ja ta lahendamiste tulemus-test.

;var.asm :: katsed. 20.11.17

global \_main

extern \_printf

section .data

 a db '123',0

 n dd 37

 pa db '%p %s',10,0 ;viit väljale ja 'väärtus'

 pn db '%p %d',10,0 ;sama

section .code

\_main:

 push ebp

 mov ebp,esp

 push a

 push a

 push pa

 call \_printf

 add esp,12

 push dword[n] ;dword: magasinielemendi 'laius',

 ;[n]: n väärtus

 push n ;4-baidine viit väljale n

 push pn

 call \_printf

 add esp,12

 pop ebp

 ret

 

Joonis x.y. Viidad ja väärtused.

Uute näidete toomiseks lisasime kord-korralt meie programmi uusi lõike; järgmi-ses programmilõigus kantakse registrisse eax stringi *a* aadress ja trükitakse see välja:

pea db '%p',10,0 ;viit väljale

...

 mov eax,a

 push eax

 push pea

 call \_printf

 add esp,8

Tulemus on joonisel x.y.



Joonis x.y. Registris on viit.

Edasi, kirjutame registrisse eax neljabaidise arvu aadressilt *n* ja trükime välja:

 ped db '%d',10,0 ;arv väljalt

 ...

 mov eax,dword[n] ;4 baiti aadressilt n

 push eax

 push ped

 call \_printf

 add esp,8



 Joonis x.y. Registris on väärtus.

Järgnevalt kanname registrisse eax neljabaidise väärtuse väljalt *n* ning omistame selle vektori *v* vasakpoolseimale elemendile:

 mov eax,dword[n] ;4 baiti aadressilt n

 mov [v],eax ;arv aadressile v[0]

 push dword[v]

 push ped

 call \_printf

 add esp,8



Joonis x.y. Kirjutamine mällu.

Mainime, et võime kirjutada ka

 mov eax,[n] ;4 baiti aadressilt n

 mov [v],eax ;arv aadressile v[0]

ent mitte „push [v]“ ̶ aadressilt magasini panemisel tuleb näidata operandi „laius“. Saame veateate:



Joonis x.y. *Push* mälust nõuab operandi pikkust.

Niisiis, kirjutada tuleb „push dword[v]“. Samamoodi nagu mäluaadresside puhul toimib sulupaar „[ ]“ ka operandina kasutatava registri puhul. Järgnevas programmilõigus kantakse viit väljale *v* registrisse eax ning arvu kandmiseks tollelt väljalt magasini tuleb lisada pikkusatribuut *dword*:

 mov eax,v

 push dword[eax]

Sulud eax ümber tähendavad, et operand on registris oleval aadressil, aga mitte registris olev aadress ise. Katse „push [eax]“ lõpeb sama õnnetult nagu eelmisel pildil nägime.

Järgmises programmilõigus kirjutatakse vektori *v* aadress registrisse, modifit-seeritakse aadressi viitamaks vektori teisele elemendile (direktiivis „add eax,4“ on 4 *vahetu operand*), kirjutatakse sinna arv 77, lisatakse see magasini (ja trükitakse välja):

 mov eax,v

 add eax,4

 mov dword[eax],77 ;mov [eax],77 annab tuttava vea

 push dword[eax]



Joonis x.y. Aadressi modifitseerimine vahetu operandi abil.

Edasi, trükime välja vektori *a* esimese sümboli („1“). Selleks tuleb ta kanda registrisse (näiteks eax) ühebaidise pikkusega ̶ kasutame atribuuti *byte*  *̶* ning registrist kasutame 8 madalamat bitti so. registrit *al.*

 pec db '%c',10,0 ;sümbol registrist

 ...

 mov al,byte[a]

 push eax

 push pec



Joonis x.y. Trükitud sümbol.

Märkigem, et süntaksivea saame, püüdes baidi väärtust kanda 32-bitisesse registrisse eax:

mov eax,byte[a]



Joonis x.y. Operandide ebaklapp.

Lisame *.data-*sektsiooni read

 pecx db 'x[0]=%c',10,0 ;symbol m2lust

 x resb 3 ;char x[3]

ja programmi *var.asm* lõigu

 mov al,byte[a]

 mov byte[x],al

 push dword[x]

 push pecx

 call \_printf

 add esp,8

Tulemus on joonisel x.y.



Joonis x.y. Mälubaidi trükk.

Järgmine näide demonstreerib register-register aritmeetikat (ja *ASCII*-koodide liitmist):

 kood98 db 'kood(a)=49,kood 98=%c',10,0

 ...

 mov al,byte[a]

 mov ah,byte[x]

 add al,ah

 push eax

 push kood98

 call \_printf

 add esp,8



Joonis x.y. Koodide liitmine registrites.

Niisiis, programmi objektide aadresse tähistatakse nimedega (näit. *v resd 3* või *ring:* ). Nime esialgseks väärtuseks on ta suhtaadress sektsioonis; translaator peab nimede tabelit, kus need on kirjas. Intuitiivselt on selge, et programselt me ei tohi (ega saagi) nime väärtust muuta (nimede tabelis üle kirjutada). Lisame oma programmi kaks rida:

 mov eax,n ;rea nr. on 127

 mov v,eax ;rida 128

Translaator annabki veateate:



Joonis x.y. Aadressi ülekirjutamise katse.

# Indekseerimine

Vektori elementide adresseerimine toimub vektori aadressile nihke liitmise abil, näiteks kui *C-*programmis kirjutame *a[0]*, *a[1]*, või *a[i]*, siis *NASM*is saame kirjutada

mov ebx,a ;vektori aadress

mov al,byte[ebx] ;a[0]

mov ah,byte[ebx+1] ;a[1]

Tavaliselt tuleb kirjutada tsükkel üle vektori elementide. Kui registrisse ebx on kantud stringi *a* aadress ja ecx-i stringi pikkus 3, siis võime indeksi jaoks kasutada näit. registrit esi ning tsükkel võib välja näha nii:

 mov esi,0

ring:

 mov al,byte[ebx+esi]

 push ecx ;hoiule, printf rikub ära

 push eax

 push pec

 call \_printf

 add esp,8

 inc esi ;add esi,1

 pop ecx

 loop ring

Selle lõigu lahendamise tulemus on joonisel x.y.



joonis x.y. Tsükkel üle stringi.

Niisiis, tsükli indeksit hoiame ja suurendame pärast igat tsüklisammu registris. Baitvektori puhul on samm 1, vektori *word-*formaadi puhul 2 ning *dword*i puhul 4. *NASM* võimaldab kasutada indeksregistri *kordajat*, väärtusega 2, 4 või 8. Meie näiteprogrammis on reserveeritud ruum *C* mõttes *int-*vektorile *v.* Järgmise programmilõiguga omistatakse ta elementidele väärtused 1, 2 ja 3.

 mov ebx,v

 mov ecx,3 ;tsükliloendaja

 mov esi,0 ;tsükliindeks

ring1:

 mov dword[ebx+esi\*4],esi

 push ecx ;hoiule, printf rikub ära

 push dword[ebx+esi\*4]

 push ped

 call \_printf

 add esp,8

 inc esi ;add esi,1

 pop ecx

 loop ring1

Mõistagi võiksime kirjutada ka *mov-* ja *push-*käskudes *dword[ebx+esi]* ja *inc esi* asemel *add esi,4.*



Joonis x.y. Tsükkel üle *int-*vektori.

# **Kompilaator, komplekteerija ja paigaldaja**

Vaatleme siin põgusalt etappe, mis läbitakse C-keeles alates tekstide lõpetami-sest. Kasutame näitena otsimis-järjestamiskahendpuuga manipuleerimise C-tekste *dp.c* (põhiprogramm, sisaldab sisendpunkti määravat *main*-moodulit), puu töötlemist võimaldavate funktsioonide jm. moodulite tekste sisaldavat *dp\_moodulid.c* ning viimaste kirjeldusi ning globaalseid andmestruktuure ja muutujaid sisaldav päisfail *dp.h*.

# **Päisfail (*header*, *xxx.h*)**

*C* on minimalistlik keel, millel on vähe reaalset funktsionaalsust: näiteks, puuduvad sisend- ja väljundoperaatorid, stringivahendid, dünaamiline mälu-jaotus, matemaatilised funktsioonid jpm. Ent keel on hõlpsasti laiendatav väliste teekide abil. Need on kompilaatoriga kaasas käivad standardsed vahendid, mille kasutamist võimaldavad makrod kujul

*#include <xxx.h>*

ning programmeerija saab luua enda tarbeks moodulite komplekte, millede kasutamiseks tuleb kirjutada makrod kujul

 *#include “xxx.h“*

Omaloodud „*C* laiendused“ tuleb kirjutada samasse kausta (teek, *folder*), kus on neid kasutavad *C-*programmid. Päisfaili programmeerimisel tuleb järgida mõnin-gaid lihtsaid reegleid[[22]](#footnote-22). Päisfaili tekst tuleb kirjutada *C* preprotsessori-korralduste

#ifndef \_\_XXX\_H\_\_

#define \_\_XXX\_H\_\_

Ja

#endif

vahele – vältimaks päisfaili „topelt-defineerimist“ iga *#include “xxx.h“*-makrot sisaldava *C-*programmi eeltöötluse käigus. Siinkohal on paras paik hoiatuseks: programmeerides tuleb vältida alakriipsuga algavaid nimesid, need on mõeldud kompilaatori ja linkeri sisekasutuseks ning põhjustavad tavaprogrammis hulgi arusaamatuid veateateid.

* Päisfail võib sisaldada ainult *#define*-makrosid, andme- (sh. struktuuride) kirjeldusi ning moodulite eelkirjeldusi. Muutujatele alg- või vaikeväärtusi seal omistada ei tohi: omistamine saab toimuda täitmise käigus, ent päisfaili ei „täideta“, ning selle reegli eiramine on taas (arusaamatute) veateadete põhjuseks. Kui päisfailis on eelkirjeldatud mooduleid (protse-duure või funktsioone), siis tuleb kirjutada nonde *C-*tekste sisaldav fail *xxx.c.* Selles *C-*failis peab olema makro

*#include “xxx.h“*

ja ei tohi olla *main*-moodulit. Käsurealt tuleb see kompileerida kui

*gcc -c xxx.c -o xxx.obj*

Võti *-c* tähendab, et *main-*moodulit pole ning *-o*, et väljund on vahekeelne *objekt-fail.*

Kuivõrd me programmeerime hiljem sama ülesande ka NASMis, siis tundub ots-tarbekas mainitud tekstid siinkohal esitada.

## **Päisfail *dp.h***

//dp päisfail. 20.10.17

//NB! Siin ei tohi olla OMISTAMISI!

#ifndef \_\_DP\_H\_\_

#define \_\_DP\_H\_\_

struct top{

 char key[32];

 struct top \*v;

 struct top \*p;

};

struct top \*juur,\*tipp,\*jooksev,\*parent;

struct top \*mstack[64]; //LIFO

int sp; //magasiniviit

FILE \*mf;

int tipa; //tippude arv

struct top \*\*tv;

int j,r;

char key[32];

void mpush(struct top \*t);

struct top \*mpop();

void wp(struct top \*t);

void tee();

struct top \*rp();

void inovp(struct top \*t);

void inopv(struct top \*t);

void t2v(struct top \*t);

int t\_arv(struct top \*t);

struct top \*new(char \*k);

void pp(struct top \*t);

int binsearch(char \*x,struct top \*v[],int n);

struct top \*vector2AVL(int low,int high);

void deltipp(struct top \*jooksev,struct top \*parent);

int roni();

#endif

Kommenteerime C preprotsessori jaoks kirjutatud makrosid *#ifndef,#define* ja *#endif.* *Include*-makroga määratud teksti kopeerib mainitud preprotsessor transleerimisele tuleva teksti ette ning kuivõrd meie näites on rida

#*include “dp.h“*

nii *dp.c* kui ka *dp\_moodulid.c* tekstides, siis ilma nende päisfaili algusse ja lõppu kirjutatud makrodeta kopeeritaks *dp.h* sisu ehitatavasse programmi topelt[[23]](#footnote-23).

# Moodulite fail *dp\_moodulid.c*

//dp\_moodulid.c :: dynaamiline otsimis- ja j2rjestamiskahendpuu

//globaalsed andmestruktuurid + moodulid. 20.10.17. Ise

#include<stdio.h> //printf(...)

#include<stdlib.h> //system

#include<string.h>

#include "dp.h"

int n=sizeof(struct top);

void mpush(struct top \*t){

 sp--;

 mstack[sp]=t;

}

struct top \*mpop(){

 struct top \*t;

 if(sp==64)return 0; //empty stack

 t=mstack[sp];

 sp++;

 return(t);

}

void wp(struct top \*t){

 fwrite(t,n,1,mf);

 if(t->v)wp(t->v);

 if(t->p)wp(t->p);

}

void tee(){

 int ait;

 struct top \*t;

 ait=sp;

 printf("tee=");

next:

 t=mpop();

 if(t){

 printf("%s ",t->key);

 goto next;

 }

 printf("\n");

 sp=ait;

}

struct top \*rp(){

 struct top \*t;

 t=malloc(n);

 fread(t,n,1,mf);

 if(t->v)t->v=rp();

 if(t->p)t->p=rp();

 return(t);

}

//v->j->p

void inovp(struct top \*t){

 if(t->v)inovp(t->v);

 printf("%s\n",t->key);

 if(t->p)inovp(t->p);

}

//p->j->v

void inopv(struct top \*t){

 if(t->p)inopv(t->p);

 printf("%s\n",t->key);

 if(t->v)inopv(t->v);

}

void t2v(struct top \*t){

 if(t->v)t2v(t->v);

 tv[j]=t; j++;

 if(t->p)t2v(t->p);

}

//siia k6ik kuni main-moodulini...

//j->v->p

int t\_arv(struct top \*t){

 tipa++;

 //printf("ta=%d\n",tipa);

 if(t->v)t\_arv(t->v);

 if(t->p)t\_arv(t->p);

}

struct top \*new(char \*k){

 struct top \*t;

 t=malloc(n);

 memset(t,'\0',n);

 strcpy(t->key,k); //kopeerib stringi

 return(t);

}

void pp(struct top \*t){

 printf("%p)%s v=%p:%s p=%p:%s\n",t,t->key,t->v,t->v->key,t->p,t->p->key);

 if(t->v)pp(t->v);

 if(t->p)pp(t->p);

}

//Kernighan&Ritchie, lk.134

int binsearch(char \*x,struct top \*v[],int n){

 int low,high,mid;

 low=0;

 high=n-1;

 while(low<=high){

 mid=(low+high)/2;

 if(strcmp(x,v[mid]->key)<0) high=mid-1;

 else if(strcmp(x,v[mid]->key)>0) low=mid+1;

 else{

 printf("v[%d]->key=%s\n",mid,x);

 return 0;

 }

 }

 printf("%s pole vektoris\n",x);

 return -1;

}

struct top \*vector2AVL(int low,int high){

 struct top \*t=NULL;

 int mid;

 if(low<=high){

 mid=(low+high)/2;

 t=tv[mid];

 t->v=vector2AVL(low,mid-1);

 t->p=vector2AVL(mid+1,high);

 }

 return t;

}

void deltipp(struct top \*jooksev,struct top \*parent){

 struct top \*b,\*ait;

 if(jooksev->v==NULL)goto parem;

 b=ait=jooksev->v;

palla:

 if(b->p==NULL)goto paika;

 b=b->p;

 goto palla;

paika:

 b->p=jooksev->p;

 goto juurpaika;

parem:

 ait=jooksev->p;

juurpaika:

 if(parent!=NULL)goto polejuur;

 juur=ait;

 goto vaba;

polejuur:

 if(parent->v!=jooksev) goto palluv;

 parent->v=ait;

 goto vaba;

palluv:

 parent->p=ait;

vaba:

 free(jooksev);

}

//j->v->p Otsib, lisab, kustutab. Globaalsed parameetrid.

int roni(){

//juur

 mpush(jooksev);

 r=strcmp(key,jooksev->key);

 if(r==0){

 printf("%s v=%s p=%s\n",key,jooksev->v->key,jooksev->p->key);

 printf("kas kustutan? j/e ");

 gets(key);

 if(key[0]=='e')return 0;

 tee();

 jooksev=mpop();

 parent=mpop();

 deltipp(jooksev,parent);

 return 0;

 }

//vasak/parem

 if(r<0){

 if(jooksev->v!=NULL){

 jooksev=jooksev->v;

 if(roni()==0)return 0;

 }

 jooksev->v=tipp; //uus tipp puusse

 return 0;

 }

 if(r>0){

 if(jooksev->p!=NULL){

 jooksev=jooksev->p;

 if(roni()==0)return 0;;

 }

 jooksev->p=tipp; //uus tipp puusse

 return 0;

 }

}

//tipuvektori tegemine, sealt kahendotsimine, AVL-puu tegemine

void tipuvektor(){

 tipa=0;

 t\_arv(juur);

 printf("tipa=%d\n",tipa);

 tv=malloc(tipa\*sizeof(struct top \*));

 j=0;

//"tree to vector"

 t2v(juur);

 printf("v6tmed vektorist:\n");

 for(r=0;r<tipa;r++)printf("%s ",tv[r]->key);

 printf("\n");

//kahendotsimine

 printf("kahendotsimine tippude vektorist:\n");

ring:

 printf("key=");

 gets(key);

 if(strlen(key)==0)goto avl;

 binsearch(key,tv,tipa);

 goto ring;

avl:

 juur=vector2AVL(0,tipa-1);

 wp(juur);

}

Juhime tähelepanu seigale, et selles C-tekstis *pole main-*moodulit, see võib olla (ja peab olema) ainult „põhiprogrammis“[[24]](#footnote-24). Ja meie päisfailile viitame makroga

#include "dp.h"

erinevalt süsteemsetest teekidest, kus päisfaili nimi on sulu-paari < > vahel.

# Põhiprogramm *dp.c*

//dp.c :: dynaamiline otsimis- ja j2rjestamiskahendpuu. 24.10.17. Ise

#include<stdio.h> //printf(...)

#include<stdlib.h> //system

#include<string.h>

#include "dp.h"

int main(int argc,char \*\*argv){

 char nimi[80];

 //k2surida OK?

 if(argc<2){

 printf("puu nime pole\n");

 return 1;

 }

 printf("argc=%d argv[1]=%s\n",argc,argv[1]); //OK k2surida

//Kui puu juba on kettal: loe ja n2ita

 mf=fopen(argv[1],"rb");

 if(mf==NULL)goto dialoog;

 juur=rp();

 printf("puu kettalt eesjrk-s:\n");

 pp(juur);

 printf("kasvav jrk:\n");

 inovp(juur);

//otsimine, lisamine, kustutamine

dialoog:

 printf("key=");

 gets(key);

 if(strlen(key)==0) goto ots;

 sp=63; //viit magasini p6hja

 tipp=new(key);

 if(juur==NULL){

 juur=tipp;

 goto dialoog;

 }

 jooksev=juur;

 roni();

 goto dialoog;

//seansi l6petamine

ots:

//kustutan 'vana puu'

 fclose(mf);

 remove(argv[1]);

//(modifitseeritud) puu kettale

 mf=fopen(argv[1],"wb");

 wp(juur);

 fclose(mf);

//teeme avl-puu, see kettale xxx.avl

 strcpy(nimi,argv[1]);

 strcat(nimi,".avl");

 mf=fopen(nimi,"wb");

 tipuvektor();

 printf("AVL-puu eesjrk-s:\n");

 pp(juur);

 printf("kahanev jrk:\n");

 inopv(juur);

 fclose(mf);

 system("pause");

}

# *dp.exe* ehitamine käsurealt

Vaatame ekraanitõmmist joonisel x.y:



Joonis x.y. *dp.exe* ehitamine käsurealt.

Esmalt kompileeritakse C teksti *dp\_moodulid.c* järgi objektfail *dp\_moodulid.obj*, kusjuures võti *-c* osutab, et C-tekstis pole sisendpunkti määravat *main-*moodulit.

Teine *gcc* poole pöördumine kompileerib *dp.c*, teeb sellestki objektfaili (mida me käsureal ei näita) ning *gcc komplekteerija* (*linker*) teeb neist kahest objekt-failist kettale *exe-*faili *dp.exe.*

Sellelaadseteks töödeks on mõttekas teha nn pakkfail (*batch file*) nimelaiendiga *xxxbat*, kusjuures nimi *xxx* ei tohi kattuda samas teegis olevate *exe-*failide nimedega. Käsurealt käivitatakse – kui programmi nimi on esitatud laiendita – esmajoones *exe-*fail ja kui seda teegis pole, siis *bat-*fail. Meie ülesande jaoks sobiks kirjutada näiteks *P.bat*:

P.bat:

gcc -c dp\_moodulid.c -o dp\_moodulid.obj

gcc dp\_moodulid.obj dp.c -o dp.exe



Joonis x.y2. Pakkfaili *P.bat* kasutamine.

# dp.exe ehitamine Dev-C++ abil

*Gnu* kompilaatorite kollektsiooni pealisehitus *Dev-C++* pakub sama töö jaoks oma võimalusi. Meie ülesande jaoks sobiva tulemuse saab nii:

1. Tühja kausta, näit. *proj* kanda failid *dp.c*, *dp\_moodulid.c* ja *dp.h.*
2. Avada *Dev-C++* ning valida *new* à *project.* Algab dialoog (vt. joonis x.1):

Valida tuleb ülalt-vasakult variant *Basic*, seejärel viiest võimalusest parem-poolseim, *Empty Project* ning alt-paremalt, kirjutada projekti nimi (meil *dp*) ja – *OK.*

Joonis x.1. Projekti sätted.

1. Rippmenüüst *Project* valida *Add To Project*; failisüsteemis tuleb minna oma kausta (meil *proj*) ning valida sealolevad kolm oma faili. *Dev-C++* vasakusse aknasse kuvatakse projekti puu (ning lisatud võib esikohale olla võõras näit. *untitled-*fail– see tuleb kustutada: (hiirega parem-klikk+“Delete). Edasine on harjumuspärane: *Execute* à *Rebuild All* (Vt.joonis x.2.)[[25]](#footnote-25)



Joonis x.2. Valmisehitatud projekt *Dev-C++* aknas.

 Projekti modifitseerimiseks tuleb vastavas kaustas topeltklikkida *puu.dev* – avaneb *Dev-C++* aken (vt. joonis 2). Kui vahepeal oli parandatud ainult komponentfaile, siis piisab toiminguist *Execute 🡪 Rebuld All*. Uusi faile saab lisada *Project 🡪 Add To Project* abil.

Projekti silumise (testimine ja parandamine) ajal on mugav *Dev-C++* aknas avada järjest kõik projekti lülitatud programmid, nende vahel on hõlpus liikuda **akna päi-**

****

Joonis x.3. Failivahetus.

**se viimases reas kuvatavate nimede abil; joonisel 3 on aktiivne neist keskmine, ’*dp.h*’. Saame avada ja parandada seda vajavaid faile, salvestada, ja samas aknas**

**projekti uuesti ’ehitada’.**



Joonis x.4. *dp.exe* lahendamisaegne ekraanitõmmis.

# Kasutatud materjalid

[geeks] <http://www.geeksforgeeks.org/memory-layout-of-c-program/> 30.10.17

[idsia] <http://people.idsia.ch/~juergen/bauer.html> 9.11.17

[push] <http://www.felixcloutier.com/x86/PUSH.html> 22.11.17

[wbss] <https://en.wikipedia.org/wiki/.bss>, 29.10.17

[x86asm] <http://cs.lmu.edu/~ray/notes/x86assembly/> 5.11.17

[Wang] Executable File Format, spring 2016, <http://www.cs.virginia.edu/~ww6r/CS4630/lectures/Executable_File_Format.pdf> 5.11.17

[wcall] <https://en.wikipedia.org/wiki/X86_calling_conventions> 8.11.17

[Yale] x86 Assembly Guide, <http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html> 6.11.17

1. *destination*, *source* ‒ kirjutamiskoht ning koht, kust kirjutatakse. [↑](#footnote-ref-1)
2. Nihe on ± suhtaadress tegeliku mäluaadressi suhtes. [↑](#footnote-ref-2)
3. *mov-*näite masinkoodi-formaat on valitud seetõttu, et siin kasutatakse kõiki võimalikke variante; muud käsud kasutavad neid valdavalt vastavalt oma spetsiifikale, näit. aritmeetika- või laadimiskäsud võivad kasutada vahe-tuid operande, ent suunamiskäsud mitte, jne. Ent formaat on ühtne ja üldine, ehkki tavaliselt mõnda komponenti ei kasutata. [↑](#footnote-ref-3)
4. Näiteks kood 100010 on *mov eax*: registri määravad vasakpoolseimad bitid „10“ ja instruktsioon *mov* on bitid „0010“. [↑](#footnote-ref-4)
5. *R/M* näitab, kas teine operand on register (*R*) või mälu (*M*, *memory*). [↑](#footnote-ref-5)
6. Nihe on absoluutsele mäluaadressile lisatav suhtaadress – näitamaks operandi tegelikku aadressi. [↑](#footnote-ref-6)
7. *Nihke* (*displacement)* pikkus on 1 või 2 baiti. [↑](#footnote-ref-7)
8. Ainult nihe on *vahetu* (*immidiate*) *operand.* [↑](#footnote-ref-8)
9. *rd=register double*, 32-bitine register. [↑](#footnote-ref-9)
10. Executable and Linkable Format [↑](#footnote-ref-10)
11. Common Object File Format. Sisuliselt on see Windowsi opsüsteemi puhul sama mis Win32. [↑](#footnote-ref-11)
12. Mainigem, et nende formaatide ülesehitus on üldvaates sarnane kaitserežiimis lahendatavale programmile tehtava mälujaotusega (vt. joon. X.1). [↑](#footnote-ref-12)
13. Artikli autor (ise kirjutab „kompileerija“) on *Narendra Kangralkar.* [↑](#footnote-ref-13)
14. Kui mingeid algväärtustatud andmeid ei tohi programmi täitmise ajal muuta, siis tuleb nende jaoks teha konstantide sektsioon, „*section .const*“ süsteemse atribuudiga *RO*. [↑](#footnote-ref-14)
15. Aparatuurne sellepärast, et magasini järge peetakse *esp-*registris ning magasini lisamise ja magasinist eemaldamise jaoks on masinkoodi käsud (assembleris *push* ja *pop*). [↑](#footnote-ref-15)
16. Vajadusel saab „muutmiskeeluga muutujate“ jaoks teha eraldi *section .const*. Märkigem, et nii saab programmeerja kaitsta ennast iseenda vigade eest. [↑](#footnote-ref-16)
17. *BSS* *( Block Started by Symbol*) on termin, mille võtsid 1950.-te keskel kasutusele *United Aircraft Corporation*is arvuti *IBM704* assembleri teinud *Roy Nutt*, b*Walter Ramshaw* jt. [wbss]. [↑](#footnote-ref-17)
18. Selle sektsiooni teine nimevariant on *.code* ̶ mis sisuliselt tundub kohasemgi. [↑](#footnote-ref-18)
19. Muudest tuntumatest variantidest mainigem järgmisi: *stdcall*, *fastcall* ja *thiscall.*  [↑](#footnote-ref-19)
20. Omistamine *a=a+b* on nüüd assembleris järgmine:

mov eax,a ;mov eax,dword[ebp-4]

add eax,b ;add eax,dword[ebp-8] [↑](#footnote-ref-20)
21. *crt* on lühend nimest *C RunTime* – standardmoodulite teek, mis kasutaja jaoks asendab 16-bitise protsessori aegseid funktsioonide teeke *DOS* ja *BIOS.* [↑](#footnote-ref-21)
22. Vt. näit. **C Header File Guidelines, David Kieras, EECS Dept., University of Michigan, December 19, 2012** [↑](#footnote-ref-22)
23. *#ifndef = „if not defined“.* Pärast esimest kopeerimist on järgmistel kordadel selle tingimuse väärtus *väär.* [↑](#footnote-ref-23)
24. *Main-*moodul määrab käivitatava *exe-*faili sisendpunkti. [↑](#footnote-ref-24)
25. Mõnevõrra arusaamatul põhjusel annab uue projekti esmakordne „ehitamine“ (*Rebuild All*) mõnikord palju veateateid, ent neist saab lahti, kui valime *File 🡪 CloseProject* ning kordame „*Rebuild All*“. Aga lõpuks tehakse (meie näite jaoks) *dp.exe.* [↑](#footnote-ref-25)