PROGRAMMEERIMINE ASSEMBLERIS

Ain Isotamm

“Programmeerimine assembleris”
Ain Isotamm
Tartu, 2020

Toimetaja: Anne Villems
Kujundus: Ellen Vene

Väljaandja: Heisenberg Ehrmantraut
ISBN 978-9916-4-0689-2

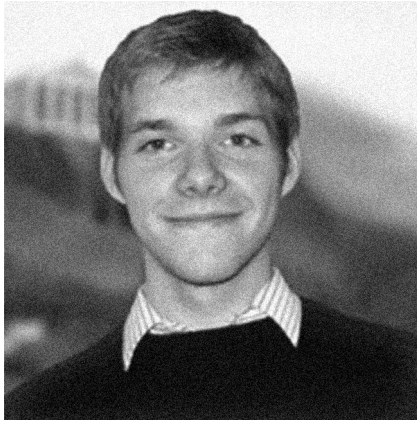
SISUKORD

SAATEKS

See raamat siin on minu neljas TÜ Arvutiteaduse instituudi egiidi all ilmuv. Eelnesid „Programmeerimiskeeled“, „Programmeerimine C-keeles algoritmide ja andmestruktuuride näidetel“ ja „Translaatorite tegemise süsteem“. Vene ajal olin eeskätt programmeerija TRÜ Arvutuskeskuses, kes selle kõrvalt tegeles õppetööga (majanduskuberneetika ja statistika kateedris), uuel Eesti ajal on programmeerimine aegamööda taandunud õppetööks vajaliku kirjutamisele.

Minu keeled olid Malgol (Minsk-22, TPI arvutuskeskus), Razdan-3 masinkood (Eesti Raadio arvutuskeskuses, Leo Võhandu meeskonnas), Minsk-32 assembler, seejärel EC-1060 makro-assembler TRÜ arvutuskeskuses. 1980-ndate lõpus tulid lauaarvutid ning programmeerisin mõnda aega Forthis, 1994-st alates tulid C tellimustööd Regiole ja selle keele juurde jäingi.

Tartu Ülikooli Arvutiteaduse instituudis oli peamiseks õpetatavaks programmeerimiskeeleks alul Java ja hiljem Python, nende kõrval muudki kõrgtaseme keeled, Haskell ja C++, näiteks. Madala taseme keelte nišš oli tühi. Jüri Kiho instituudi tolleaegse juhatajana oli nõus, et teen algust C fakultatiivkursusega (2007). Füüsikute programmijuhi Kalev Tarkpea ettepanekul tegime selle ainekursuse pooleks: kaks kuud C ja kaks kuud Inteli assembler, esimesel aastal (2011) *MASM-32* ja hiljem *NASM*. Assemblerit õpetas mitu aastat Jorma Rebane – alustades oli ta II kursuse informaatikatudeng, kes võttis 2010.a. C



Jorma Rebane

kursust ja jäi meelde tavapäratu huvi ning teadmistega C tausta – assembleri – valdkonnas.

Aga miks ma lugesin üles omad raamatud:

- „Programmeerimiskeeltes“ [Isotamm, PKd] on väga lihtne sissejuhatus teemasse „masinkood ja assembler“, mida söandan soovitada lugemiseks enne käesoleva raamatu läbitöötamist, ning süsteemprogrammeerimise sissejuhatus (mõiste, Forth ja C).
- „Programmeerimine C-keeles Algoritmide ja andmestruktuuride näidetel“ [Isotamm, C] peaks olema Kernighani ja Ritchie raamatu [K&R] järel „kohustuslik kirjandus“ läbitöötamiseks enne assembleri kallale asumist: käesoleva raamatu algoritmide tutvustamine käib eeskätt C-keeles ja harva verbaalselt.
- „Translaatorite tegemise süsteem“ [Isotamm, TTS] võib meie raamatu kontekstis olla C-programmeerimise vaates suhteliselt suure moodulite süsteemi realiseerimise näide, aga primitiivse Algol-tüüpi keele kompilaatori (trigol → exe-fail) vaheetapp on assemblerkoodi genereerimine (16-bitine Borland TurboAssembler).

Niisiis, need van(em)ad raamatud võivad hõlbustada käesoleva raamatu lugemist, aga C-teadmised – kusiganes nad ka saadud on[1] – on arusaamiseks vajalikud.

„Akadeemilistes ringkondades“[2] levinud arvamus oli C-keelele kaua ebasoodus. Autoriteetne Niklaus Wirth[3] kirjutas, et „Tegelikult esitab C assembleri koodi, mis on peidetud ilmetu süntaksi varju ning mis on täis tipitud igasugu salamärke“ [Wirth]. Talle sekundeeris (irooniliselt) Eric Lebherz, et C „kombineerib assemblerkeele kogu elegantsi ja võimsuse assemblerkeele loetavuse ja hallatavusega“ [Lebherz].

Jättes kõrvale Lebherzi iroonia, on C kõigist kõrgtaseme (masinast sõltu-

1 Parim C-raamat on võrgus allalaetav: Brian W. Kernighan, Dennis M. Ritchie, The C Programming Language, Second Edition (ANSI C), Prentice Hall Software Series.

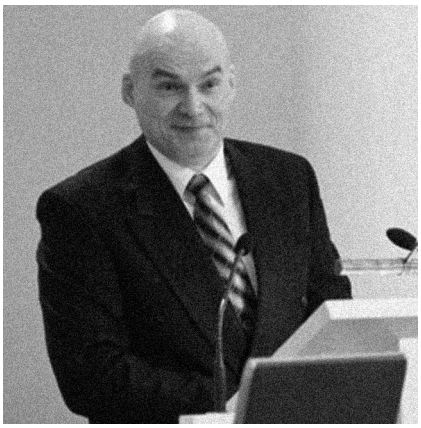
2 Peame siin silmas valdkonna juhtivaid ülikoole ja aktsepteeritavaid publikatsioone juhtivates teadusajakirjades.

3 Šveits, keeled Pascal, PL360; Modula jpt. Hoolimata Wirthi kriitikast on paljud hilisemad keeled võtnud eeskujuks C kirjapildi.

matutest) keeltest kõige lähemal assemblerkeeltele (säilitades oma masinsõltumatuse). C-l on tarkvara programmeerimises eriline koht:

- ta on konkurentsitult populaarseim süsteemprogrammeerimise keel (operatsioonisüsteemide loomine, translaatorite tegemine jmt);
- kompilaatorite kirjutamise kaasaegne trend on „keel X → C → lõpuks .exe-fail“ (vt. näit. [github]).
- C standardprogrammide teeke kasutavad tänapäeval paljud (enamik?) mikroprotsessorite jaoks realiseeritud programmeerimiskeeli (tavaliselt kogumina crt library, crt = C RunTime. Windowsi operatsioonisüsteemis on nad failis MSVCRT.DLL)[1].

Ja assembleri roll C kompilaatorites on olla objektkeel – see, millesse tõlgitakse lähteprogramm; edasine jääb assembleri (kui translaatori) hooleks. Niisiis, ahel on „keel X → C → assembler → *.exe*-fail“.



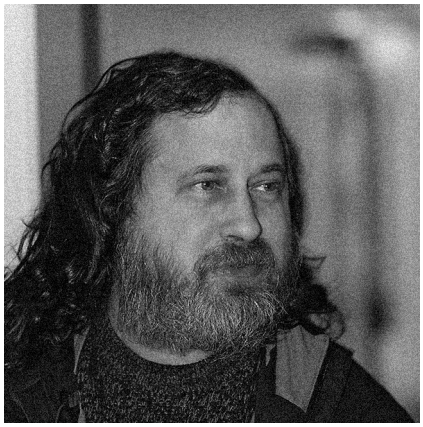
Kalev Tarkpea

Selle raamatu kirjutamise algpõhjus oli füüsiku Kalev Tarkpea ettepanek — tutvustada arvutitehnika tudengitele C kõrval Inteli assemblerit. Ehkki olin kaua aega tagasi pikalt programmeerinud assembleris (Minsk-32, EC-1060), polnud mul aimugi mikroprotsessori assemblerist ning tuli valida, milline vabavaraline (ülikoolil oli siis veel vähem raha kui nüüd) oleks sobiv, ja võtsin Microsofti paketi *MASM*-32 (sellel on graafiline Windowsi-liides, umbes nagu *gcc* pealisehitus *Dev-C++*), ent üsna tüütud nõudmised.[2] Esimesel õpetamise aastal kasutasimegi seda paketti, ent siis leidis Jorma üles *NASM*i ja see oli hea leid.

NASM-programmide komplekteerija (linker) on *gcc* (*Gnu Compilers Collection*) – millega saab käsurealt transleerida ja komplekteerida ka C-programme. Siinkohal paar lauset Gnu'st.

1 16-bitise protsessori ajal olid nood funktsioonid koondatud DOSi ja BIOSi funktsioonide teekidesse.

2 Näiteks, C eeskujul tuli näidata kasutatava crt-funktsiooni teek (näit. stdlib.h) ja seejärel tuli seda veelkord üle deklareerida), lisaks tegi *MASM* projekti jm. tülikat.



Richard Stallman

Tolle firma asutas hea haridusega (Harvardi ülikool – bakalaureus – ja *MIT – Massachusetts Institute of Technology*), mõlemad suur-Bostoni Cambridge osalinnas, vahemaa ca 1 miil) vabavara „käilakuju“ – ka üks esimestest häkkeritest – Richard Stallman (1953). Gnu projektiga alustas Stallman 1983. aastal (rohkem loe [Stallman]).

Õpperaamatutest soovitan lugeda Paul A. Carteri oma [Carter][1]. Nii see raamat, Kernighani ja Ritchie C-raamat [K&R] kui ka minu raamatud (sh. käesolev) on lihtsalt allalaaditavad minu koduleheküljelt *http://kodu.ut.ee/~isotamm/* Tartu Ülikooli serveris.

Ja veel üks soovitus – lugeja võiks liituda kahe portaaliga: *Code Project*[2] ning *The Crazy Programmer*[3], kus leiab aeg-ajalt midagi huvitavat ka assembleri kohta.

Käesoleva raamatu kirjutamise mõte oli aidata asjasthuvitatul[4] omandada assembleris programmeerimise kunsti[5]. Enamik assembleri-raamatuid (mida on lihtne „guugeldades“ leida) alustavad (pisut vabandavas toonis) jutuga, miks ikkagi tänapäeval võiks assemblerit õppida (ajakriitilised moodulid, mikrokontrollerite programmeerimine jmt); minu arvates võib aga põhjus olla eluterve uudishimu rahuldamine: kuidas asjad masinas *tegelikult* käivad.

1 Algajat võivad mõneti segada tema kasutatud makrod, mis kohati varjavad „päris-*NASM*i“.

2 *https://www.codeproject.com/*

3 *https://www.thecrazyprogrammer.com/*

4 Just „huvitatuile“, neile, kes ise tahavad õppida assembleris programmeerima. Minu kogemus TTÜ Tartu kolledžis näitab, et suure kohustusliku C+assembleri masskursuse tegemine kukub läbi – ilma omapoolse motivatsioonita on see aine liiga raske. Kusjuures, esimesed kaks aastat (2016 ja 2017) olid kolledžis väikeste rühmadega minu jaoks puhas rõõm.

5 D. Knuthi meenutades, „The *Art* of Computer Programming“.

1 ARVUTI JA MASINKOOD

1.1. MIKROARVUTI

Personaalarvuti (*Personal Computer*, PC) arhitektuur ja toimimismehhanismid on kõrgkoolide (meil TTÜ ja TÜ füüsikaosakonna) õppekavade temaatika. Meie püüame käesolevas sissejuhatuses anda lihtsa (ja lihtsustatud) rakendusprogrammeerijavaate masinale – et teaksime, mida me juhime ja mis jääb meie eest paratamatult varjatuks ka meie raamatus. Situatsioon on umbes sama nagu autoinseneride ja autojuhtide õppevahendite suunitluses, hea juht peaks teadma mõndagi oma masinast, aga tänase auto eri sõlmede konstruktorite valdkond on ikkagi midagi muud kui see, mida vajab hea autojuht heaks juhtimiseks. Niisiis, järgnev pole mõeldud riistvaraprofessionaalidele, vaid toda riistvara kasutatavatele programmeerijatele[1].

Arvutid – vähemalt pärast John v. Neumanni[2] printsiipide omaksvõtmist – koosnevad masinatena **seadmetest**:

- Juhtimisseade (*Control Unit*), mis annab järgemööda (kui eelmine käsk seda järjekorda ei muuda) ette täitmisele tulevaid masinkoodi-käske: dekomponeerib masinkoodi (käsk, registrid, mäluaadressid, vahetud operandid jmt). Käsu kood näitab, mis on komponendid ning milline on käsu pikkus (mis määrab järgmise käsu aadressi). Ja käsu kood näitab, milline seade käsku täitma hakkab.
- Aritmeetika-loogikaseade (*Arithmetical-Logical Unit*), mis sooritab koodiga määratud täisarvudevahelise aritmeetikatehte või „loogika“ (arvude võrdlemine, bitikaupa operatsioonid) tehte.

1 Võhandu ja raadio; meie: masinorienteerit asm! 3. taseme keeles programmeerimiseks ei peagi teadma

2 Väga lühidalt – arvuti töötab kahendsüsteemis ja nii programm kui ka töödeldavad andmed on arvuti ühtses mälus.

- Registrid – spetsiifilised mäluväljad, milledele kirjutamine ja milledelt lugemine võtab võimalikest kõige vähem aega.

Need seadmed koos moodustavad **keskprotsessori**[1] (*CPU*, *Central Processor Unit*). Lisaks moodustavad masina:

- Ajaseade (*Time Unit*), mis peab arvestust suuresti erinevate „aegade“ üle: kalendriaeg, mida peab ülal sisseehitatud autonoomse patarei toitel olev kell, protsessori töötaktide loendaja, protsessoriaeg, protsessiaeg.
- Mäluseade (*Memory Unit*), mis on ühine nimetaja erinevatele „allseadmetele“:

  → *ROM* (*Read Only Memory*): „ainult lugemiseks“, kirjutamine on rakendusprog-rammide jaoks võimatu; seal on tavaliselt algpaigalduse (*boot*) käsud. Varem olid seal ka standardsed sisend- ja väljundfunktsioonid (*BIOS – Basic Input/Output System*). *ROM*-mälu võib olla autonoomsel toitel.

  → *RAM* (*Read-Access-Memory*) – seal on nii programmid, aparatuurne magasin kui ka programmide andmed. Seejuures on ka siin kirjutamiskaitsega (nagu *ROM*) piirkonnad, näiteks see, kus on täidetav programm.

  → *Cache* – *RAM*ist kiirem vahemälu[2] „ettepumbatud“, *RAM*ist pärit käskude jaoks, kiirendamaks protsessori tööd. Rakendusprogrammidel puudub sellele mälule juurdepääs.
- Välisseadmed: ekraan, klaviatuur[3], kõvaketas[4], lisamälu(d), skanner, printer, hiir, võrguühendus(ed) jne. Nende ja keskprotsessori infovahetus toimub eriliideste – draiverite (*driver*) abil.
- Tänapäeval võime „masina“ komponentidele lisada emaplaadil paikneva kiibi, mille nimi on harjumuspäraselt *BIOS* (*Basic Input/Output System*) ja nüüd pigem *UEFI* (*Unified Extended Firmware Interface*). Mikroarvuti käivitamisel käivitub esimesena *BIOS* ja aktiveerib arvuti

1 Miks keskprotsessor: vastavalt vajadusele jagab ta ülesandeid teistele, nn. kaasprotsessoritele.

2 Vahemälu maht võib varieeruda tüübist olenevalt piirides 32 KB kuni 32 MB [cache].

3 Nende kahe ühine nimetus on „konsool“.

4 Nimetagem seda harjumuspäraselt nii – ehkki insenerlik lahendus ei pruugi enam „tiirlev ketas“ olla.

riistvara, misjärel paneb ta käima algpaigaldusprogrammi (*boot*), mis kutsub välja operatsioonisüsteemi (näit. Win7 või Ubuntu) [BIOS].

Siinkohal tuleb vist meenutada üht arvutiloo seika. Esimesed masinad konstrueeriti arvutusmatemaatikute jaoks nende tööde hõlbustamiseks ning kulus üsna palju aega kuni lisandusid tekstitöötlus ning süsteemprogrammeerimise ülesanded – eeskätt translaatorite ja operatsioonisüsteemide kirjutamine. Mikroarvutid seevastu alustasid nii, et arvutus-matemaatika ujupunkt-tehteid esialgu prioriteetseteks ei peetud (nende ülesannete jaoks olid oluliselt võimsamad kesk- ja suurarvutid) ning võimsuse (mälu mahu ja protsessori kiiruse) kiire kasv võimaldas minna teadusarvutuste turule. Heaks lahenduseks osutus mikroprotsessorite tegijaile ujupunktarvutuste jaoks kaasprotsessori loomine (Inteli oma on tuntud kui x87). Sel on oma käsustik ja registrid ning kaasprot- sessor käivitatakse, kui keskprotsessori käsuregistris on ujupunktprotsessori käsk. Selle protsessori programmeerimist tutvustame ja katsetame peatükis 13.

Ujupunkt-kaasprotsessor pole ainus omataoline. Tsiteerigem allikat [blogspot]: „operatsioonid, mida võivad täita kaasprotsessorid on ujukomaarvutused, graafikatöö, signaalitöötlus, kodeerimine/dekodeerimine, krüpteerimine. Kaasprotsessorid aitavad vabastada põhiprotsessori spetsiifilistest toimingutest ja tõsta nii süsteemi jõudlust... Näiteks võimaldab spetsiaalsel graafikakaardil olev protsessor vabastada põhiprotsessori Blu-Ray video dekodeerimisega seotud koormusest ja põhiprotsessor saab samal ajal pühenduda teistele operatsioonisüsteemi hooldustegevustele või muude kasutajarakenduste käivitamisele, mis töötavad koos filmi kuvamisega.“ Graafikaprotsessori programmeerimine tundub nii keerulise kui ka huvitavana, ent – paraku – sellesuunalised katsetused ei mahu meie raamatu temaatikasse, jääme oma raamidesse: rakendusprogrammeerimine IA-32 vahendite kasutamine *NASM*-keeles ja *gcc* keskkonnas.

Niisiis, kogu masin töötab kui masinkoodi interpretaator. Interpreteeritav programm koosneb käskudest, mille operandid on registrid ja *RAM*-piirkonna mäluaadressid (sh. magasini omad) ning üldjuhul võib operandiks olla ka käsku kirjutatud konstant (vahetu operand). Mikroprotsessorid on üheaadres-

silised: käsus osaleb operandina ülimalt üks mäluaadress (sh. erikohtlemisega magasiniaadress) – sellega on kõik lihtne, silmas tuleb pidada vaid operandi pikkust baitides ning operandiväljale salvestatud/salvestatava muutuja tüüpi. Ent samas on peaaegu kõik operatsioonid binaarsed, käsus osaleb kaks operandi: register-register, register-mälu, mälu-register (pluss mõningad vahetu operandiga võimalused); **mälu-mälu tehteid pole**. Registrite kui operandidega on asi iseenesest ka lihtne, aga nõuab nii arusaamist kui ka harjumist.

1.2. REGISTRID

1.2.1. IA-32 ÜLDREGISTRID

Meie raamatu kontekstis (*IA-32*[1], 32-bitine *NASM*) on olulised 4 struktuurset üldregistrit *eax, ebx, ecx* ja *edx*, kaks indeks- (või baas-)registrit *esi* ning *edi*, kaks aparatuurset magasini ülal pidavat registrit *esp* ja *ebp*, pluss rakendusprogrammeerijale kättesaamatu *eip* (*instruction pointer*, järgmise käsu aadressi hoidmiseks) ning rakendusprogrammeerija jaoks ilmutatud kujul vähefunktsionaalne signaalide („lippude") register *eflags*. Lipuregistrit kasutavad vaikimisi kõik tingimusliku suunamise käsud (null? väiksem? miinus? jne) ning vektoritega opereerivad käsud – seal tuleb lipuregistrisse fikseerida vektori töötlemise suund: vasakult paremale või vastupidi.

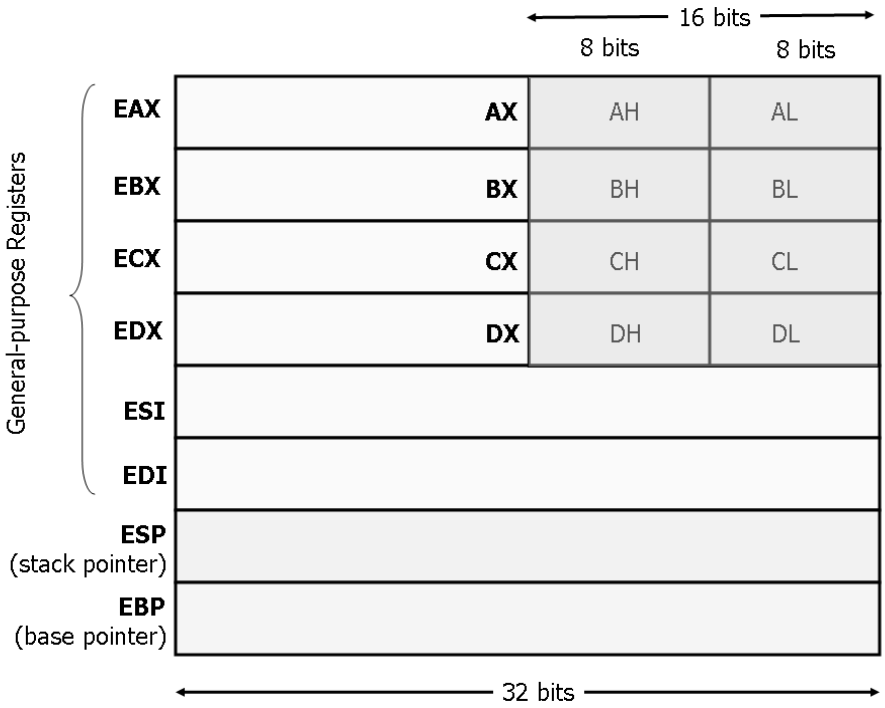
64-bitise protsessori registrite ülevaade on lisas 2. Üldregistrite struktuuri peaks selgitama joonis 1.2.1.a.

- AL/AH/AX/EAX: Akumulaator, tehete operandi ja funktsiooni resultaadi tagastamise rollis;
- BL/BH/BX/EBX: Baas, struktuurse mäluvälja algusaadress;
- 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

1 *IA-32: "Intel Architecture, 32-bit"*, 32-bitine Inteli arhitektuur.

“tavatöödeks”, EAX kuni EDX on kasutamiseks tõepoolest vabad.



Joonis 1.2.1.a. Üldregistrid [Yale]

Registrite ESI ja EDI eriotstarve on olla indeksite hoidmiseks, kui programmeeritakse tsükleid üle vektorite. Kui tsüklis osaleb neid kaks, siis üks on tavaliselt “ressursi” ja teine “resultaadi” rollis. Nende registrite sihtotstarbeline kasutamine on määratud omaette käskudegrupiga, muidu aga võib neid registreid kasutada enam-vähem vabalt.

Registrid ESP (magasini tipu viida jaoks) ja EBP (aktiivse mooduli “freimi” baasi fikseerimiseks magasinis) on kasutatavad ainult sihtotstarbeliselt. Magasini (*stack*) tutvustame hiljem.

Assemblerprogrammid on soovitatav vormistatada moodulitena – arvestusega, et nende poole pöördub ülemise taseme programm (väljakutsuja, *caller*) ja kirjutatav moodul on “väljakutsutav” (*callee*). Registrite EAX, ECX ja EDX säilimise eest – kui pöördutakse alamprogrammi poole – peab hoolitsema *caller* ja ülejäänute eest *callee*. See tähendab, et kui alamprogramm kasutab mõnda neist “ülejäänud” üldotstarbelisest registrist (EBX, ESI või EDI), siis peab ta nende seisu enne väljumist taastama.

Joonisel pole kaht registrit.

- EFLAGS on ühebitiste signaalide jaoks. Enamus neist on protsessori omakasutuses või dokumenteerimata, rakendusprogrammeerijale olulised lipud on need, mida heisatakse käskude resultaadina. Ainuke programselt muudetav lipp on *DF* (*destination flag*), mille väärtus '0' näitab, et vektorit töödeldakse vasakult paremale, ja '1' vastupidises suunas. Ülejäänuid saab rakendusprogrammides kasutada tingimustele orienteeritud direktiivides, näiteks *ZF* (*zero*, null): 1, kui tehte resultaat = 0 – käsud "mine, kui *ZF*=0" või "mine, kui *ZF* 0";
- EIP: *Instruction pointer*. Järgmisena täidetava käsu aadress. Kaitserežiimis[1] pole see register kasutajaprogrammidele kättesaadav.

Märkigem, et masinkoodikäsud 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. Mida suurem on "ringi" number, seda rohkem võimalusi on varjatud. Tavaliselt kasutatakse ring1-privileege draiverite programmeerimiseks, ring2 on sama tavaliselt täpsemalt piiritlemata ning ring3 on rakendusprogrammeerijate päralt. Võimaluste peitmine johtub julgeolekukaalutlustest: tuumaprogrammid ei tohi arvutit "kinni jooksutada" ja rakendusprogrammidel on see silumise etapil tavaline, et nende töö päädib avariiga, ent arvuti töötab normaalselt edasi.

1.2.2. MUUDEST REGISTRITEST[2]

Lisaks üldregistritele on protsessoris *segmentregistrid*, mille kasutamine oli 16-bitise arhitektuuri ajal programmeerijale vältimatult vajalik[3]. Tolleaegne mälujaotus järgis "reaalrežiimi" – operatiivmälu jagunes 64-kilobaidisteks

1 8- ja 16-bitised personaalarvutid töötasid „reaalrežiimis" (*real mode*): kogu ühemegabaidine op-mälu oli kättesaadav (640KB lihtsamalt ja 360KB mõningase vaevaga) rakendusprogrammidele. See polnud kuigi turvaline ning alates 32-bitisest protsessorist töötavad rakendusprogrammid „kaitserežiimis" (*protected mode*) – kasutajale on kättesaadav ainult talle (tavaliselt virtuaalselt) eraldatud 4-gigabaidine (2^{32}) piirkond *RAM*-is.

2 Pikemalt vt. Lisa 2 „x64 lühiülevaade". x64 tähistab Inteli 64-bitist arhitektuuri; samaväärne on *IA-64*.

3 Segmentregistrid on ka praegu programmeerijale lugemiseks kättesaadavad.

16-bitise aadressiga adresseeritavateks plokkideks, aadressruum oli aga 20-bitine: "segmendi" baasaadress (20 bitti) ja 16-bitine suhtaadress segmendis[1]. Segmentide baasaadresse hoitakse registrites:

- CS: koodisegmendi (c*ode Segment*) register,
- DS: andmesegmendi (*data Segment)* register,
- SS: magasinisegmendi (*stack Segment*) register ja
- ES: lisasegmendi (*extra Segment*) register. Viimast saab kasutada vastavalt vajadustele; nood neli olid kasutusel alates 16-bitise arhitektuuri evitamisest. Mõnevõrra hiljem lisati veel kaks segmentregistrit, FS ja GS, nende nimed võeti lihtsalt tähestiku järgi järgmised.

Segmentregistreid kasutab protsessor ka 64-bitises masinas[2], aga programmeerija ei pea nendega enam ise manipuleerima – alates 32-bitise arhitektuuri ja „kaitserežiimi" evitamisest.

64-bitisele masinale on lisatud 8 üldotstarbelist struktuurset 64-bitilist registrit: r8..r15 – 64-bitised ja r8d .. r15d – 32-bitised r8..r15 "madalamad järgud"; "d"-variandis pole paraku nende registrite 32 "vasakpoolsemat" bitti kasutatavad.

Registrite-teema lõpetuseks naaseme veel kord „ringide" juurde. Ring3 hõlmab rakendusprogrammeerijatele kättesaadavaid vahendeid (ressursse, sh. kasutatavad registrid ja masinkoodi käsud). Mida väiksem on „ringi" number, seda vähem on kitsendusi ning nois priviligeeritud ringides on kasutatavad kolm komplekti eriotstarbelisi registreid (vt. näit. [Chourdakis]):

- 4 juhtregistrit (*Control Registers*) CR0...CR3;
- 8 silumisregistrit (*Debug Registers*) DR0...DR7 ja
- 4 testimisregistrit[3] (*Test Registers*) TR0...TR3.

1.2.3. INTEL X86 (IA-32) MÄLUJAOTUS

Siin jaotises esitame artikli[4] „*Memory Layout of C Programs*" [geeks]

1 Segmendi pikkus oli 16-bitises variandis kuni 65 536 (2^{16}) baiti.

2 Protsessori „bitilisuse" suurenemisega ei kaasnenud segmentregistrite nimedele prefiksi lisamine, so. näit. andmesegmendi register on läbivalt DS ja mitte EDS või RDS.

3 [Chourdakis] märgib, et need pole enam kasutusel. Mõeldud olid nad protsessori enda testimiseks.

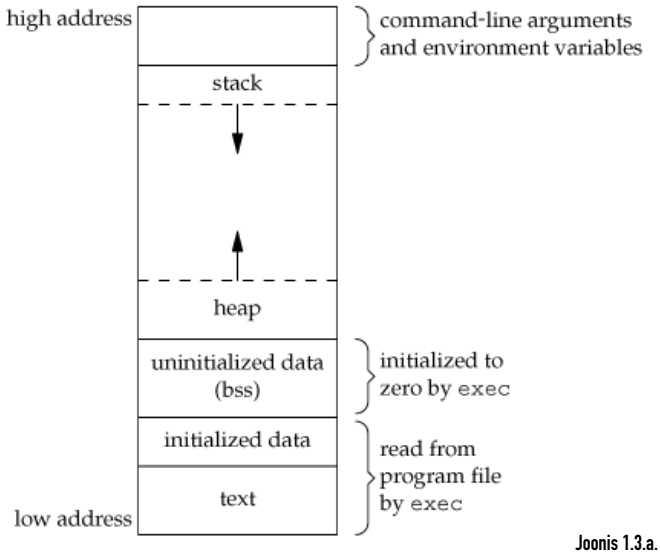
4 Artikli autor (ise kirjutab „kompileerija") on Narendra Kangralkar.

refereeringu. Ehkki tolle pealkiri viitab üheselt C-programmi lahendusaegsele mälujaotusele, on pilt samasugune ka *NASM*i puhul; sisuliselt on C mälupilt dikteeritud *NASM*i omast, viimase määrab aga x86-le orienteeritud Windowsi kaitserežiimile (*protected mode*) tuginev mälujaotus.

Niisiis, tavapärane lahendusaegne operatiivmälu on jaotatud järgmisteks osadeks (segmendid, sektsioonid – tuletagem meelde segmentregistreid – neis on hoiul segmentide baasaadressid):

1. „*text*“ – täidetav programm, so. masinkoodi käskude jada (*NASM*is *section .text*), juurdepääs on ainult lugemiseks (*Read Only, RO*);
2. väärtustatud (*initialized*) andmed (*NASM*is *section .data*) Juurdepääs nii lugemiseks kui ka kirjutamiseks (*RW*)[1];
3. algväärtustamata (*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 *NASM*is funktsiooni *malloc* abil. Juurdepääs on *RW*.

Programmi lahendusaegse mälupidi esitame originaalis (vt. joonis 1.3.a).



Joonis 1.3.a. Lahendamisaegne mälujaotus [geeks].

1 Kui mingeid algväärtustatud andmeid ei tohi programmi täitmise ajal muuta, siis tuleb nende jaoks teha konstantide sektsioon, „*section .const*“ süsteemse atribuudiga *RO*.

Joonisel esitatu tõlge ja kommentaarid: pildil on väikseim mäluaadress all ja suurim üleval. 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.

Järgmine piirkond on aparatuurse magasini[1] jaoks. Magasin „kasvab“ aadresside vähenemise suunas. Teoreetiliselt on võimalik, et „vastukasvav“ kuhi ja aadresside mõttes „vastukahanev“ magasin saavad kokku; sel juhul võivad uuemad lahendused kasutada virtuaalmälu, kui aga mitte, siis tagastab *malloc* NULL-viida (vaba mälu pole), *push* aga annab signaali *stack overflow*.

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'
```

Ning samas piirkonnas on ka keskkonnamuutujad. Nende hulka kuulub ka *tee* – meie raamatu kontekstis trajektoorid C-ketta juurkataloogist failideni *nasm.exe* ja *gcc.exe*.

Ja lõpuks, eelmises peatükis mainitud segmentregistrid (*CS, SS* jt.) hoiavad kaitsepiirkonna sihtotstarbeliste mäluväljade alg- („baas“-)aadresside (*code* segment, *stack* segment) väärtusi.

Ja et aduda, mis meie programmide täitmisel *tegelikult* toimub, ei pääse me tolle interpreteeritava masinkoodi konspektiivsest tutvustamisest. Seda teeme järgmises jaotises.

1 Aparatuurne sellepärast, et magasini järge peetakse *esp*-registris, freimi baasi *ebp*-registris ning magasini lisamise ja magasinist eemaldamise jaoks on masinkoodi käsud (assembleris *push* ja *pop*). Ja magasiniga manipuleerivad ka direktiivid *call* (mine alamprogrammi) ning *ret* (välju alamprogrammist). Magasin *ise* on tavaline *RAM*-mälu piirkond.

2 INTEL x86 MASINKOOD[1]

Mikromasinate kood on üheaadresseline: masinkoodi formaat näeb ette kas ühe mäluaadressi, või pole seda üldse. Ja masina „aadressilisuse“ määrab ära mäluaadresside maksimumarv koodis.

Mikroprotsessorite ehk suurim tootja Intel on suhteliselt lühikese ajaga jõudnud 8-bitisest mudelist 64-bitise baasmudelini, valides jätkusuutliku tee: iga järgmine protsessor on säilitanud võimalikult palju eelmistest variantidest. See on peamine põhjus, miks masinkoodi baasformaadile on lisatud kuni neli ühebaidilist prefiksit – kolme neist kasutatakse, kui käsk toimib 16-bitises või 64-bitises režiimis ja sellega kehtestatakse (*override*) muu kui 8- või 32-bitine variant[2]. Asi on selles, et baasformaadis on registri jaoks kolm bitti ning režiimi (8 bitti või pikem variant) määrab 1 bitt – seega valida saab kahe režiimi vahel. Et *ASCII*-stringid on olulised, siis lühike variant on 8-bitine; 16-bitise arhitektuuri ajal oli loomulik „pikk variant“ kahebaidine (näit. *AX*) ning üleminekul 32-bitisele masinale oli valida, kas registri 000 pikk variant on jätkuvalt *AX* või on see *EAX*. Loomulikult valiti viimane.

64-bitisele arhitektuurile minek valikuprobleeme (nähtavasti) ei tekitanud. Kaitserežiimi (*protected mode*) puhul opereerivad programmid 32-bitises (4GB) aadressruumis ning 64-bitine protsessor seda ei muutnud. Ehk ainus programmeerijale nähtav lisavõimalus on registrite r8d..r15d lisandumine ning „mittenähtav“ on vastava prefiksibaidi lisamine masinkoodi nende registrite kasutamise puhul.

1 Selle peatüki kirjutamisel on tuginetud põhiliselt Chemnitzi Tehnikaülikooli veebimaterjalidele.

2 Siit nõuanne: kui programm peab olema võimalikult lühike ja/või töötama võimalikult kiiresti, siis tuleb kasutada kas 8- või 32-bitiseid registreid ja mäluvälju, vältimaks prefikseid.

2.1. FORMAAT[1]

Intel x86 käsu pikkus on 1 kuni 15 baiti[2] ja selle põhimõtteline formaat on järgmine:

0 kuni 4 baiti **prefiksite**[3] jaoks – 1 bait igale (vt.[CIS-77]):

- käsu prefiks (*instruction prefix*) *lock*, *rep* või *repne* on kasutatavad, kui mitu protsessi jagavad omavahel mälu ning vastava prefiksiga saab üks protsessidest oma töö lõpetada teisi eemal hoides. Selle baidi võimalikud väärtused on f0h[4] – *lock*, f2h – *repne* ja f3h – *rep* või *repe*. Viimastega saab kaitsta tsükleid üle stringide. Assembleris tuleb vastav prefiks (*lock*, *rep*) programmeerijal mnemo-koodi kirjutada – näit. *repe movs*;
- aadressi prefiks (*address-size prefix*) 67h.
- operandi prefiks (*operand-size prefix*) 66h neid kahte kasutatakse, kui vaikimisi-kehtivate 8- või 32-bitiste operandide asemel kasutatakse 16- või 64-bitiseid. Assembler-translaator lisab prefiksi 66h või 67h ise;
- segmendi prefiks (*segment override*) võimaldab vaikimisi-kehtiva mälusegmendi registri asemel kasutada mõnda muud, näit. koodisegmenti (CS=2eh), magasini-segmenti (SS=36h), andmesegmenti (DS=3eh) jne. Ilmselt pole see programmeerija teema.

Järgneb 1 või rohkem nn „standardkomponenti“ (vt. [Chemnitz]):

1. **Käsu kood** (*OpCode*) 1 või 2 baiti. Kui nummerdame baidi bitte vasakult paremale (7, 6, .. ,0), siis bittidel 7..2 on kood, bitt kohal 1 on d (*destination*) – andmete liigutamise suund[5]. Registrist mällu suuna puhul d=0 ja mälust registrisse – 1. Bitt kohal 0 on s (*size*): 0, kui operandid on 8-bitised ja 1,

1 See jaotis tuleb lugejal vajadusel mitu korda läbi lugeda ja ka näidete juurest ilmselt siia tagasi pöörduda, sest ilma käsu formaati mõistmata pole võimalik näiteid jälgida ega ise masinkoodis programme kirjutada. (Toimeteja)

2 15-baidine piirmäär tähendab seda, et ühes käsus pole kunagi kasutatud korraga kõiki fakultatiivseid võimalusi.

3 Prefiks pole käsu normaalne komponent – ta „kirjutab üle“ normaalse käsu välju asendamaks protsessori tehtavat tavainterpretatsiooni prefiksi poolt määratuga.

4 „h“ on 16-ndarvu tunnus, näit. 17=11h. (Loetavamalt, $17_{10} = 11_{16}$)

5 Andmeid liigutatakse lähtekohast (source) sihtkohta (destination). Võimalikud liikumised on registrist registrisse, registrist mällu või mälust registrisse. Varianti „mälust mällu“ Intel (paraku) ei toeta.

kui 32-bitised (16- või 64-bitise variandi jaoks on prefiks 0fh – sel juhul ongi käsu kood kahebaidine. Selle prefiksi parempoolne naaber on alati koodibait.). Ja veel – see prefiks ei kuulu ülalmainitud „nelja võimaliku prefiksi“ hulka;

2. **0 või 1 bait nimega *Mod-REG-R/M*.** Sellega kirjeldatakse eeskätt binaarse tehte teist operandi. Bittidel 7 ja 6 on ***MOD***. Vaatame võimalikke variante:

   00 – kaudadresseerimine registri abil[1] (*register indirect addressing mode*) või ilma nihketa[2] *SIB* (*SIB with no displacement*)[3] ksj. *R/M*=100 või ainult nihke abil adresseerimine (*displacement only addressing mode*), *R/M*=101;

   01 – ühebaidine nihe (*displacement*);

   10 – neljabaidine nihe;

   11 – register

   Bittidel 5-3 (*REG*) on register (teine operand):

   000 – al või eax;

   001 – cl või ecx;

   010 -- dl või edx;

   011 -- bl või ebx;

   100 -- ah või esp;

   101 -- ch või ebp;

   110 -- dh või esi;

   111 -- bh või edi.

Bittidel 2—0 (*R/M*) näidatakse käsu teist operandi (register või mälu)[4] pluss võimalikku nihet baasaadressi suhtes. Semantika sõltub *MOD*-bittidest; allikast [Chemnitz] pärineb järgmine tabel[5]:

1 Otseadresseerimise puhul on aadress operandi vahetu aadress, kaudadresseerimise puhul on aadress „viit aadressile, kus on operandi aadress“.

2 Nihke väärtus on täisarv, mis liidetakse struktuurse mäluvälja (näit. vektori) baasaadressile võimaldamaks juurdepääsu struktuuri elemendile. Nihe võib olla konstandi väärtus või muutuja (näit. vektori indeksi) väärtus.

3 *SIB (Scaled Indexed Adressing Mode)* – mälu skaleeritud indekseerimine. „Skaleerimine“ tähendab vektori indeksi sammu määramist. Bait-vektori indeks muutub sammuga 1 (2^0), kahebaidiste elamentide puhul on samm 2 (2^1), neljabaidiste elementide puhul 4 (2^2) ja 8-baidistele 8 (2^3).

4 Suuna 'register → mälu' või 'mälu → register' määrab käsu koodi bitt d. Kui d=0, siis on register lähtekoht ja kui d=1, siis sihtkoht.

5 Tabelis tähendab disp8 ühe- ja disp32 neljabaidilist nihkevälja.

```
MOD R/M Addressing Mode
=== === ================================
00 000 [ eax ]
01 000 [ eax + disp8 ]                (1)
10 000 [ eax + disp32 ]
11 000 register  ( al / ax / eax )   (2)
00 001 [ ecx ]
01 001 [ ecx + disp8 ]
10 001 [ ecx + disp32 ]
11 001 register  ( cl / cx / ecx )
00 010 [ edx ]
01 010 [ edx + disp8 ]
10 010 [ edx + disp32 ]
11 010 register  ( dl / dx / edx )
00 011 [ ebx ]
01 011 [ ebx + disp8 ]
10 011 [ ebx + disp32 ]
11 011 register  ( bl / bx / ebx )
00 100 SIB  Mode                     (3)
01 100 SIB  +  disp8  Mode
10 100 SIB  +  disp32  Mode
11 100 register  ( ah / sp / esp )
00 101 32-bit Displacement-Only Mode (4)
01 101 [ ebp + disp8 ]
10 101 [ ebp + disp32 ]
11 101 register  ( ch / bp / ebp )
00 110 [ esi ]
01 110 [ esi + disp8 ]
10 110 [ esi + disp32 ]
11 110 register  ( dh / si / esi )
00 111 [ edi ]
01 111 [ edi + disp8 ]
10 111 [ edi + disp32 ]
11.11 register  ( bh / di / edi )
```

3. **0 või 1 bait nimega *SIB*** *(Scaled Indexed Adressing Mode)* – mälu skaleeritud indekseerimine. „Skaleerimine“ tähendab vektori indeksi sammu määramist. Bait-vektori indeks muutub sammuga 1 (2^0), kahebaidiste elamentide puhul on samm 2 (2^1), neljabaidiste elementide puhul 4 (2^2) ja 8-baidistele 8 (2^3). *SIB*-baidi bitid jagunevad järgmiselt:
   7 ja 6: skaala (kahe astendaja – 00, 01, 10 või 11);
   5 – 3: indeksregister ja
   2 – 0: vektori baasi register.
4. **0 või 4 baiti nihke** (*displacement*) jaoks juhul, kui käsus on registris baasaadress ning nihe on täisarv, millega modifitseeritakse baasiga määratud aadressi. Register määrab nihke diapasooni, näit. kui register on *AL*, on nihe ühebaidine, *EBX* puhul aga neljabaidine[1].
5. **0 või 4 baiti vahetu (*immediate*) operandi** jaoks. See saab olla käsu teine operand ning esimeseks operandiks oleva registri bittide arv määrab vahetu operandi baitide arvu käsus.

2.2. ARVUDE SALVESTAMINE

Nihke ja vahetu operandi käsukoodis kujutamise eripära viib „sujuvalt“ sellele, kuidas x86[2] mäluseadmes arvusid hoiab. Baidi kujutamine on ootuspärane: arv on 8-l bitil just nii, nagu me eeldame. Sinna mahuvad arvud 0...255 (2^8-1) ning 1 on mälus bitijadana 00000001 ja 254 – 11111110. Kahe- ja enamabaidiseid arve hoitakse baitide pöördjärjestuses, näiteks arv 258 on „normaalselt“ (kui loetavaks tegemiseks paneme baitide vahele tühiku) 00000001 00000010, ent pöördjärjestuses 00000010 00000001. Miks nii arve hoitakse – ühest vastust ei tea (arendus 8-bitisest protsessorist suutlikuma suunas?), ent enda jaoks oleks mugav asja seletada „india arvutustega“ – näiteks, paberi ja pastakaga liidame 1789+23 nii:

$$\begin{array}{r} 1789 \\ +\quad 23 \\ \hline 1812 \end{array}$$

1 Suunamiskäskude (sh. *call*) puhul on nihe lähte- ja siht(suht)aadresside vahe.

2 Nii teeb enamik mikroarvutite tootjaid.

Eks ole, liidame paremalt vasakule, 9+3=12 – „2 kirja, 1 meelde“, 8+2=10 – „1 oli meeles – 1 kirja ja 1 meelde“ – ning 7 asemele kirjutame 8 – „1 oli meeles“.

Kui kanname oma mugava algoritmi masinale üle, siis on lihtne alustada liitmist ühelistest jne.

Ja nood „ühelised“ on rajastatud ja „kohakuti“, arvuväljade (võimalik, et need on erinevate pikkustega) algustes, kusjuures meie „meeldejätmise“ jaoks on lipuregistris (*FLAGS*) *carry flag* (*CF*)[1]. Seejuures – kui resultaat-arvuväli saab enne täis kui viimane ülekanne õnnestus, lülitab *CF* sisse ületäitumislipu *OF* (*overflow flag*).

Ingliskeelses erialakirjanduses on nende variantide nimed *big endian* (enne tuhanded, siis sajad, kümned ja ühed) ja little endian (ühed, kümned, sajad jne), kusjuures nende mõistete seletus on ehtinglaslik: Gulliveri reisides on maa, mis oli kodusõjas või selle äärel, ja põhjus oli selles, kuidas alustada keedetud muna koorimist – kas tömbist otsast (*big end*) või teravamast (*little end*). Ei mäleta, kas Gulliver jõudis võitjate selgumise ära oodata, aga Intel valis poole: *little endian*.

Sellest hoolimata, käsud on protsessori jaoks ees meile loetava(ma)s „*big endian*“-formaadis välja arvatud mitmebaidine nihe ja vahetu operand – nende formaat on „*little endian*“.

2.3. KÄSU DEKODEERIMISE NÄITED

Selles jaotises tuuakse näiteid masinkoodi „dekodeerimisest“. Paratamatult tuleb seejuures seletada, mida „käsk teeb“, so näidata käsu semantikat – aga selleks sobib kõige paremini vaadeldava käsu assemblerkeelne vaste. Aga assemblerkeelt käsitleme (sissejuhatavalt) mõne-võrra hiljem. Loodame siiski, et see raamatu loetavust üleliia ei sega.

Chemnitzi Tehnikaülikooli saidil [Chemnitz] on masinkoodi näidetena toodud liitmiskäsu (*add, ADD*) võimalike variantide koodid. Meie arvates on mõned[2] neist siinkohalgi hea ära tuua. Niisiis:

1 Ülekandelipp.

2 Enne näiteid on seal märkus, et need on lihtsustatud. Asjast ettekujutuse saamist see ei sega, küll

- ***ADD***i opkood on 00000ds, kus d=0, kui liidetakse registrist mällu (mäluaadressil olevat arvu suurendatakse koos salvestamisega) ja d=1, kui registris olevat arvu suurendatakse. Bait *MOD-REG-R/M* võib „saatja“ ümber mängida. Kui s=0, siis on operandid ühebaidised, muidu (ilma prefiksi sekkumiseta) neljabaidised.
- ***ADD CL,AL***: registris *CL* oleva baidi väärtust suurendatakse registris *AL* oleva baidi väärtuse võrra. Üsna hea metakeel selle väljendamiseks on *CL*=(*CL*)+(*AL*). Sulgudeta pannakse kirja „aadress“, sihtkoht, ja sulud näitavad, et sulgudes olevalt aadressilt saadakse bitid („väärtus“). Jälgitavuse huvides eraldame bitijada(de)s koodikomponendid punktidega. Kood on 000000.0.0.11.000.001 (kood d s mod reg r/m): Kood on 6 „nulli“; d=s=0. Et s=0, on operandid 8-bitised. Bitt d=0 – liidetakse *REG* ja *R/M*-iga näidatud väljadel olevad väärtused. Mängu tuleb *MOD*-komponent 11: teine operand on register ja selle registri määrab baidi *MOD-REG-RM*-komponent *REG*=000: et s=0, siis teine operand on *AL*. Ning et *R/M*=001, siis „saaja“ rollis on register *CL*. (*CL*=001). Käsk 16-ndkoodis on **00 c1**.
- ***ADD ECX,EAX*** : eelmise näite metakeelt kasutades *ECX*=(*ECX*-)+(*EAX*). Käsk 16-ndkoodis on **01 c1** – erinevus on koodi s-bitis. Et see on 1, siis operandid on 32-bitised. Aga bitikaupa: kood on 000000.0.1.11.000.001.
- ***ADD EDX,* <nihe>**: meenutuseks – „nihe“ on *displacement instruction*. Käsk[1] (ikka 16-ndkoodis) on **03 1d ww xx yy zz**: 000000.1.1.00.011.101 <nihe>, kus ww, xx, yy ja zz on 4-baidise nihke komponendid (max $=2^{32}$-1). Näiteks, kui nihe=4, siis kahendkoodis on see komponent 0100.0000.0000.000[2]. Harutame selle käsu lahti: koodi bitt d=1 näitab, et „saaja“ on register ja „saatja“ määrab *R/M*-komponent. s=1 näitab, et operandid on 32-bitised. Kombinatsioon *MOD*=00 ja *R/M*=101

aga võib tekitada küsimusi, kui hakkame oma transleerimislistingut uurima. Näiteks pole *ADD*i opkood mitte alati 000000ds. Me ei tea, millist Inteli pidevalt arendatud versiooni näidete kirjutamisel kasutati. Näiteiks oleme valinud ainult need, mille kehtivust „täna“ kinnitab tegelik transleerimislisting. Tänakehtivaid seise näitame tabelis 2.3.2.

1 [wx86] annab koodiks 81 c2.

ADD Add (1) `r/m += r/imm`; (2) `r += m/imm`; 0x00...0x05, 0x80/0...0x83/0

2 *Little endian*

määrab, et teine aadress on ainult „nihe“[1]. Ja *Mod-Reg-R/M*- komponent *REG* näitab, et „saaja“ on register *EDX*. Näitekäsk: ***add edx,4***.

- ***ADD EDI,[EBX]***: *EDI=(EDI)+(EBX)* .Kood on 03 3b: 000000.1.1.00.111.011. Et d=1, siis liidetakse *REG (EDI)* ja *R/M*-komponendiga määratud [*EBX*], ksj „saaja“ on *EDI*. *MOD*=00 näitab, et nihet pole.
- ***ADD EAX, [ESI + disp8]***: *EAX=(EAX)+(ESI+nihe8)*, kus *disp8*, *nihe8* tähendavad ühebaidilist nihet. Kood on **03 46 xx**: 000000.1.1.01.000.110. xxxxxxxx.

Vaatamaks, kuidas NASM Chemnitzi näiteid transleerib, kirjutasime vastava „pseudoprogrammi“, mille ainus eesmärk oligi objektfaili listingu saamine. Selleks sobis järgmine käsurida:

```
nasm -f win32 chemnitz.asm -o chemnitz.obj -l chemnitz.txt
```

Tulemus on järgmine:

```
1                                   ;chemnitz.asm
2
3                                   section .text
4 00000000 00C1                        add cl,al ;00 c1
5 00000002 01C1                        add ecx,eax ;01 c1
6 00000004 81C200000100                 add edx,65536  ;03
1d ww xx yy zz
7 0000000A 033B                       add edi,[ebx] ;03 3b
8 0000000C 034604                    add eax,[esi+4] ;03 46
```

Loodetavasti andis ülaltoodu võtme x86 masinkoodi dekomponeerimiseks – hoolimata tõigast, et tegu on lihtsustatud „õpikunäidetega“. Järgmises tabelis on toodud liitmiskäsu tegelikud formaatid [x86IS]:

1 Nihe (*displacement*) täpsustab operandi aadressi, vahetu (*immediate*) operand on käsku sisse kirjutatud.

kood	“mnemoonika”	tegevus
04 ib	ADD AL,imm8	AL+=vahetu8
05 iw	ADD AX,imm16	AX+vahetu16
05 id	ADD EAX,imm32	EAX+=vahetu32
80 /0 ib	ADD r/m8,imm8	r/m8+=vahetu8
81 /0 iw	ADD r/m16,imm16	r/m16+=vahetu16
81 /0 id	ADD r/m32,imm32	r/m32+=vahetu32
83 /0 ib	ADD r/m16,imm8	r/m16+=laiendatud vahetu8[1]
83 /0 ib	ADD r/m32,imm8	r/m32+=laiendatud vahetu8
00 /r	ADD r/m8,r8	r/m8+=r8
01 /r	ADD r/m16,r16	r/m16+=r16
01 /r	ADD r/m32,r32	r/m32+=r32
02 /r	ADD r8,r/m8	r8+=r/m8
03 /r	ADD r16,r/m16	r16+=r/m16
03 /r	ADD r32,r/m32	r32+=r/m32

Ülaltoodud tabelis on käsu koodi kõrval lühendid *ib, iw* ja *id* – need näitavad teise operandi pikkust, /0 on koodi täiend ning */r* näitab, et teine operand on kas register või mäluväli. Mujal tabelis tähistab *r/m* kas registrit või mälu ning sellele järgnev arv tolle operandi bittide arvu.

Seda tabelit testisime pseudoprogrammiga test.asm ja tulemused on järgmised:

```
1                                   ;test.asm [x86IS]
2                                   section .bss
3 00000000 <res 00000001>              M8  resb 1
4 00000001 <res 00000002>              M16 resw 1
5 00000003 <res 00000004>              M32 resd 1
6                                   section .text
7 00000000 0407                     add al,7
8 00000002 6683C007                 add ax,7
9 00000006 83C007                   add eax,7
10 00000009 8005[00000000]07         add byte[M8],7
11 00000010 668305[01000000]07       add word[M16],7
12 00000018 8305[03000000]07         add dword[M32],7
13 0000001F 0025[00000000]           add byte[M8],ah
```

1 Bittide arv võrdsustatakse “saaja” pikkusega.

```
14 00000025 660105[01000000]        add word[M16],ax
15 0000002C 0105[03000000]          add dword[M32],eax
16 00000032 0225[00000000]          add ah,byte[M8]
17 00000038 660305[01000000]        add ax,word[M16]
18 0000003F 0305[03000000]          add eax,dword[M32]
```

Pöörakem tähelepanu tabeli ridadele 8, 11, 14 ja 17, neis on kasutusel 16-bitise operandi prefiks.

Veel ühe pseudoprogrammi (r.asm – „r“ nagu registri nime prefiks) kirjutasime näitlikustamaks 64-bitise protsessori käske. Käsurida:

```
nasm -f win64 r.asm -o r.obj -l r.txt
1                                   ;r.asm
2                                   section .bss
3 00000000 <res 00000001>              M8  resb 1
4 00000001 <res 00000002>              M16 resw 1
5 00000003 <res 00000004>              M32 resd 1
6 00000007 <res 00000008>              M64 resq 1
7
8                                   section .text
9
10 00000000 0407                     add al,7
11 00000002 6683C007                 add ax,7
12 00000006 83C007                   add eax,7
13 00000009 4883C007                 add rax,7
14 0000000D 4983C007                 add r8,7
15 00000011 4183C007                 add r8d,7
16 00000015 800425[00000000]07       add byte[M8],7
17 0000001D 66830425[01000000]-      add word[M16],7
18 00000025 07
19 00000026 830425[03000000]07       add dword[M32],7
20 0000002E 48830425[07000000]-      add qword[M64],7
21 00000036 07
22 00000037 002425[00000000]         add byte[M8],ah
23 0000003E 66010425[01000000]       add word[M16],ax
24 00000046 010425[03000000]         add dword[M32],eax
25 0000004D 48010425[07000000]       add qword[M64],rax
```

```
26 00000055 4C013C25[07000000]      add qword[M64],r15
27 0000005D 022425[00000000]        add ah,byte[M8]
28 00000064 66030425[01000000]      add ax,word[M16]
29 0000006C 030425[03000000]        add eax,dword[M32]
30 00000073 48030425[07000000]      add rax,qword[M64]
```

Näeme, et prefiks on lisatud käskudele ridadel 11, 13, 14, 15, 17, 20, 23, 25, 26, 28 ja 30.

3 ASSEMBLER

3.1. ASSEMBLERKEEL

Vististi kõige üldisem programmeerimiskeelte klassifikatsioon jaotab need keeled kahte suurde klassi: masinast sõltuvad keeled ja masinast sõltumatud keeled. Esimesed on orienteeritud konkreetsetele protsessoritele, neist omakorda on madalaima, 0-taseme keel masinkood – ainus keel, mida protsessor suudab interpreteerida[1].

Masinkoodis on täiesti võimalik programmeerida, suhteliselt lihtne oli see IBM/360/370 arvutitel (vt. näit. [Isotamm, PKd]), kus kood on ühebaidine, 15 üldregistri numbrit (1...f) mahuvad loetavalt poolbaiti ning mäluväljade suhtaadressid on omaette täisbaitidel. Ent nagu eelmises peatükis nägime, tuleb x86 masinkoodi kirjutada sisuliselt bitikaupa – mis on „suhteliselt keeruline".

Ent sõltumata protsessorist teevad masinkoodis programmeerimise ning programmide lugemise keeruliseks järgmised seigad:

- Pole kommentaare.
- Arvulised käsukoodid ei jää kuigi hästi meelde ega pole seetõttu ka kuigi hästi loetavad.
- Mälujaotusega peab tegelema programmeerija: puuduvad programmi objekte tähistavad märgendid (etiketid), nende asemel tuleb käskudesse kirjutada nonde objektide suhtaadressid.
- Käskude lisamine või eemaldamine muudab programmi objektide suhtaadresse ning nende uued väärtused tuleb neid objekte kasutavates käskudes käsitsi parandada.

1 On lahendusi, kus on veel madalam (-1?)-tase, kus saab programmeerida masinkoodi dekomponeerimist ning kirjutada nonde madalaima taseme komponentide baasil uusi käske. Seda võimaldas näit. Armeenia miniarvuti Nairi-2.

Ja niipea, kui arvutitele lisati teksti sisestamise/väljastamise võimekus (seni piirdusid võimalused ainult märgiga arvude ja kümnendpunktiga), loodi kõikjal esimesed assemblerid, mis võimaldasid kirjutada masinkoodiga võrreldes kõrgema (esimese) taseme keeles ning tõlkisid kirjutatud programmi masinkoodi. Mainigem, et vastava translaatori kirjutamine on küllaltki lihtne.[1]

Üldiselt on assemblerkeel üksüheses vastavuses masinkoodiga, ent – nagu eelmise peatüki lõpus nägime – mitte tingimata.

Assembler on süsteem, mis koosneb sisendkeelest (assemblerkeel) ja translaatorist masinkoodi (assembler-translaator). Etteruttavalt, edasises me kasutame mõistet *assembler* nii ühes kui ka teises tähenduses.

Kuivõrd oma assembleri tegid peaaegu kõik mingi arvutitüübi masinkoodi-programmeerijad, siis kujunes üpriski kirju pilt. Ühest tolleaegsest trükisest jäi meelde „assemblerite välimääraja“ – kuidas aru saada, kas tegu on assembleri või millegi muuga. Sisetundega: koeratõuge on väga palju ja nad erinevad „seinast seina“ (taskukoerast bernhardiinini), aga nii meie kui ka koerad ise ei eksi, kui ütleme, et see siin *on* koer, aga see seal (hunt või rebane) *pole*.

Assemblerkeele peamised tunnused on järgmised:

- Programmi tekst kirjutatakse ridahaaval, igal real on üks *direktiiv* – üheks masinkoodi-käsuks teisendatav eeskiri;
- Iga rida järgib kindlat formaati: fakultatiivne etiketiväli[2], käsu mnemokood, operandid (need võivad ka puududa) ning nende järgi võib kirjutada kommentaari.
- Võimalus kirjutada reale ainult kommentaar.
- *Mnemokood* on käsukoodi sisuline nimi (*add* vs. 03).
- Etikettide (märgendite) kasutamine programmi objektide tähistamiseks – see tagab „auto-maatse“ mälujaotuse.
- Registrid kirjutatakse direktiivi nende nimesid (*eax, r15*) kasutades.
- Direktiivide (assemblerprogrammi käskude) lisamine/kustutamine on

1 Siiski, assemblerist polnud huvitatud *kõik* arvutuskeskused: kui pakkusin (vist 1973) Eesti Raadio arvutuskeskusele, et teen nende Razdan-3 jaoks assembleri, siis nad ei tahtnud – neil oli kasutada mahukas ja hästidokumenteeritud masinkoodis kirjutatud moodulite teek, milledest sai nagu legoklotsidest kokku laduda suvalise uue suure programmi. Ilma vajaduseta pealisehituse järele.

2 Kõrgema taseme keeltes kutsutakse etikette märgenditeks.

probleemivaba.

Nagu teiste protsessorite jaoks on ka x86 jaoks tehtud üsna palju assemblereid. Netiallikas [Assemblers] mainib mõningaid neist:

- *MASM*-32 – *Microsoft Macro Assembler*. Graafilise liidesega (nagu C jaoks Dev-C++) assembler, mis pole eriti kasutajasõbralik, nõudes eriti crt-moodulite[1] kasutatavaks tegemiseks liigset informatsiooni.
- *FASM* – „*Flat Assembler*“. Kasutatav mitme erineva op-süsteemiga.
- *BBC „Basic for Windows“*
- Linuxi assembler
- *YASM*
- *NASM* – „*Netwide Assembler*“. Objektiivsetel (vabavaraline) ja subjektiivsetel (lihtne ja loogiline) põhjustel kasutame selles raamatus just seda assemblerit.
- *GNU* assembler *GAS*. See firma – *Gnu* – on meile tuttav ja oluline oma *gcc*-ga (*Gnu Compilers Collection*), seda nii C-programmide käsurea-kompilaatorina (mille graafiline liides on Dev-C++) kui ka eraldi transleerimise resultaatidest – *.obj*-failidest – *.exe*-faili komplekteerijana (*linker*). Etteruttavalt, *gcc* abil komplekteerime ka oma assembler-programmid.

3.2. KOMPILAATOR, KOMPLEKTEERIJA JA PAIGALDAJA

Programmeerimiskeelte realiseerimisel on kaks võimalust: tehakse kas interpretaator, mis programmi analüüsi puu (tavaliselt tsüklilise) läbimise käigus seda programmi täidab või kompilaator, mis tolle puu ühekordsel läbimisel genereerib 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

1 „*C RunTime*“ – meile tuttavad C standardfunktsioonid on (vähemalt Windowsi puhul) teegis *crt.lib*.

(*linker*) ja paigaldaja (*loader*) poolt modifitseerida (vt. näit. [Isotamm, TTS]). Niisiis, programmeerides kompileeritavas kõrgtaseme keeles on programmi realiseerimise (*.exe*-faili tegemise) tavaline vaheetapp masinkoodi kompileerimine assemblerist. Ja assembleri esmane väljund on vahekeelne objektfail.

3.2.1. GNU C REALISEERIMINE

„Programmeerimiskeele realiseerimise“ all mõistetakse tema jaoks translaatori kirjutamist ning üldjuhul läbivad kõrgtaseme (masinast sõltumatute) keelte mikroprotsessoritele orienteeritud translaatorid neli faasi[1]:

1. Preprotsessimine. Töö käib C-programmi teksti tasemel: töödeldakse makrosid[2] (süsteemsed makrokäsud, nagu *include* või *define* asendatakse makrolaienditega – needki on tekstid). Ning sel etapil eemaldatakse tekstist kommentaarid.
2. Preprotsessori resultaadist genereeritakse *Gnu* assemblerkeele *GAS* tekst (selle saab „kinni püüda“, näited järgnevad pisut hiljem) nimelaiendiga „s“.
3. Assembleri tekstist transleeritakse vahekood – objektfail nimelaiendiga “*.o*“ või „*.obj*“.
4. *Gcc* linker viib töö objektfailiga lõpule, kirjutades kettale *.exe*-faili.

Selle protsessi näitlikustamiseks toome triviaalseima C-programmi („*hello world*“) *tere.c* transleerimise etapiviisilise käigu. Kompileerimiseks piisab käsureast

```
>gcc tere.c -o tere.exe
```

Faasikaupa saame jälgida resultaadini jõudmist nii:

```
>gcc -S tere.c
```

gcc genereerib *gnu* assembleri (*GAS*) teksti *tere.s* (vt. pilti.)[3]

Ehkki *„tere.s“*- tekst on korrektne assemblertekst (mis järgib *cdecl-konventsiooni* – sellest hiljem), on see teine assembler, *GAS*, ja mitte *NASM* ega aita kuigi palju meie raamatu materjali paremini mõista. *GAS*-sisendkeeles

1 *Gnu-C* realiseerimise kohta vt. näit. [Chen].

2 Makrodest vt. näit. [Isotamm, PK].

3 *GAS* sisendkeelena on lihtsam kui siinnäidatu, siinne on „sisekasutuseks“. Siiski, võiks tähele panna *GAS*i iseärasusi – näiteks, direktiivides on „saaja“ (*destination*) ja „saatja“ (*source*) vahetuses – kui võrdleme *NASM*iga. Ning operandidel on prefiksid – registritel ’%’ ja teistel ’$’.

näeb meie (inglise keeles tervitav) lihtprogramm nii välja [GAS]:

```
        .file   "hello.c"
        .def    ___main; .scl 2; .type 32; .endef
        .text
LC0:
        .ascii "Hello, world!\12\0"
.globl _main
        .def _main; .scl 2; .type 32; .endef
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        call    __alloca
        call    ___main
        movl    $LC0, (%esp)
        call    _printf
        movl    $0, %eax
        leave
        ret
        .def    _printf; .scl 2; .type 2; .endef
```

```
Administrator: cmd - Shortcut

C:\0_asm\tere>type tere.c
#include<stdio.h>
   char t[]="tere kevad!";
int main(){
    printf("%s\n",t);
}

C:\0_asm\tere>type t.bat
gcc -S tere.c
gcc -c tere.s -o tere.o
gcc tere.o -o tere.exe
tere

C:\0_asm\tere>t

C:\0_asm\tere>gcc -S tere.c

C:\0_asm\tere>gcc -c tere.s -o tere.o

C:\0_asm\tere>gcc tere.o -o tere.exe

C:\0_asm\tere>tere
tere kevad!

C:\0_asm\tere>dir
 Volume in drive C has no label.
 Volume Serial Number is 8055-F841

 Directory of C:\0_asm\tere

12.04.2019  21:16    <DIR>          .
12.04.2019  21:16    <DIR>          ..
12.04.2019  21:11                70 t.bat
12.04.2019  21:15                86 tere.c
12.04.2019  21:16            40 746 tere.exe
12.04.2019  21:16               752 tere.o
12.04.2019  21:16               572 tere.s
               5 File(s)         42 226 bytes
               2 Dir(s)  85 167 513 600 bytes free
```

Joonis 3.2.1.1. gnu C-programmi transleerimise ositamine.

```
Administrator: cmd - Shortcut
               2 Dir(s)  85 167 513 600 bytes free

C:\0_asm\tere>type tere.s
        .file   "tere.c"
        .globl  _t
        .data
        .align 4
_t:
        .ascii "tere kevad!\0"
        .def    ___main;        .scl    2;      .type   32;     .endef
        .text
        .globl  _main
        .def    _main;  .scl    2;      .type   32;     .endef
_main:
LFB10:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $16, %esp
        call    ___main
        movl    $_t, (%esp)
        call    _puts
        movl    $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
LFE10:
        .ident  "GCC: (MinGW.org GCC-6.3.0-1) 6.3.0"
        .def    _puts;  .scl    2;      .type   32;     .endef

C:\0_asm\tere>_
```

Joonis 3.2.1.b. *tere.s* listing *gnu*-assembleris *GAS* (vahekeelne tekst).

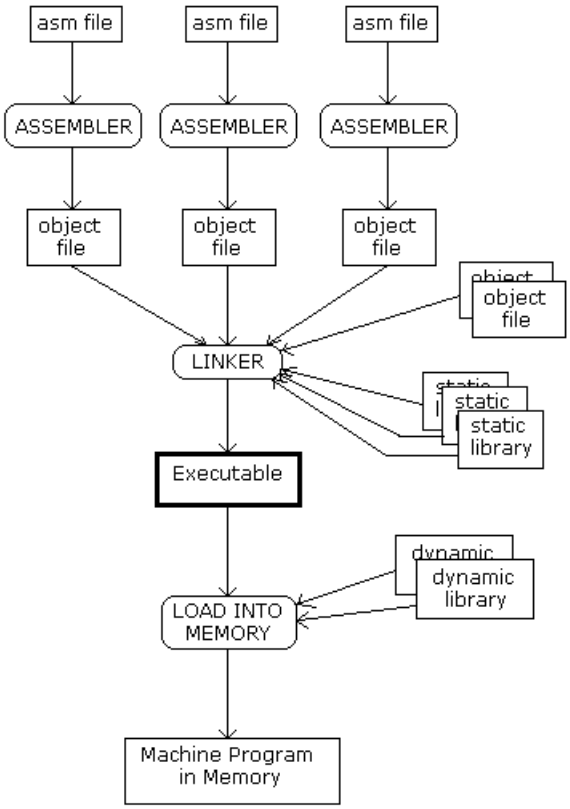
3.2.2. ASSEMBLERPROGRAMMI KOMPILEERIMINE

Meie raamatu assembler-keel on *NASM*[1], mille komplekteerijana kasutatakse *gcc*-d. Viimane toetab mitut erinevat objektfaili-formaati, sh. Windowsi operatsioonisüsteemile orienteeritud formaate win32 või win64, Linuxi elf32[2] või *UNIX*i coff[3]. Joonisel 3.2.1.a (allikas: [x86asm]) on esitatud assembleri, linkeri ja paigaldaja üldine skeem.

1 Miks just see – mõistagi, see on „subjektiivne“. *MASM*-32 tundus tüütu, *NASM* mitte, ja see otsustaski valiku (ksj, *NASM*i leidis meie tandemi jaoks Jorma). Borlandi *TASM* (*Turbo Assembler*) oli normaalne, ent paraku 16-bitine.

2 *Executable and Linkable Format*

3 *Common Object File Format*. Sisuliselt on see Windowsi opsüsteemi puhul sama mis Win32.



Joonis 3.2.1.a. ***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 nende tutvustamine ei mahu nende kaante vahele[1]. Huvilistele soovitame näiteks Wei Wangi netimanuaali [Wang].

1 Mainigem, et nende formaatide ülesehitus on **üldvaates** sarnane kaitserežiimis lahendatavale programmile tehtava mälujaotusega (vt. joon. X.1).

4 NASM

4.1. SAAMISLOOST

Oma koduleheküljel [Tatham] kirjutab Simon Tatham, et keegi kaastudengitest kaebles kunagi jälle tema kuuldes, et pole viisakat ja vabavaralist Inteli assemblerit, ning olles noor ja idealistlik tudeng[1], võttis ta asja ette, saades nii *NASM*i esmaarendajaks[2]. *NASM*i koduleht [NASM] märgib tema kõrval teisena Julian Halli.

S. Tatham lahkus projektist ajapuudusel pärast ülikooli lõpetamist. Kodulehe andmeil on jätkamas viiemeheline rühm H. Peter Anvini juhtimisel (ka J. Hall pole enam meeskonnas).

Muide, Simon Tatham on ka failitranspordi programmi PuTTY autor {Tatham].

Allikas [wnasm] iseloomustab NASMi järgmiselt: ta suudab genereerida erineva formaadiga objektfaile – *Win, COFF, OMF, a.out, ELF , Mach-O*.

Väljundformaatide mitmekesisus võimaldab *NASM*i kasutada kõigis x86 jaoks kirjutatud operatsioonisüsteemide keskkondades. *NASM*i abil saab genereeri-

Simon Tatham (s.3.05.1977)

1 Cambridge'i ülikoolis (UK).

2 Paraku ei õnnestunud teada saada, miks sai nimeks *Netwide Assembler*. Siinkirjutaja ei näe ei neti, ei wide'i.

da ka lame-kahendfaile[1] (nimelaiendiga *.bin*) – see formaat sobib algpaigaldusprogrammide (*boot loaders*) ning *ROM*-piirkonna programmide (näit. *BIOS*i – *Basic Input-Output System*) kirjutamiseks.

NASM-programmid tuleks kirjutada *Cdecl*-kokkuleppeid järgides. Kuivõrd *crt*-funktsioonid[2] (millede hulka kuuluvad ka kõik C standardfunktsioonid) on programmeeritud *Cdecl*-kokkulepetest kinni pidades ning et *NASM* võimaldab neid kasutada, tuleb sellega arvestada: *crt*-alamprogrammi parameetrid edastatakse magasinis ning pärast *crt*- alamprogrammist naasmist peab väljakutsuja nad – parameetrid – sealt eemaldama.

Seejuures – NASM ise ei ole orienteeritud *Cdecl*-ile ega ka ei kontrolli sellest kinnipidamist; põhjus on *crt*-programmide möödapääsmatus kasutamises.

„Oma“ alamprogrammide kirjutamiseks võib kasutada loomulikult ka muid väljakutsevariante[3] kui *Cdecl*, aga programmi kirjutamise rutiini huvides võiks *Cdecli* läbivalt järgida. Kasvõi selleks, et kui meie kirjutatud funktsioonid lähevad laiemalt kasutusse, siis pole pöördumisprobleeme.

4.2. KESKKOND

Siinkohal on vististi sobiv tutvustada x86 assembleri suhteid kahe funktsioonide hulgaga – *BIOS*-funktsioonide ja *MS-DOS*[4]-funktsioonidega. Ütleme kohe, et alates 32-bitisest aritektuurist pole esimesed enam programselt vahetult kättesaadavad ning osa teistest – käsurea-direktiivid – on (käsuga *system*) .

BIOS (*Basic Input/Output System*) oli 16-bitise protsessori ja 1 MB mälu ajal funktsioonide kogum operatiivmälus *RO*-kaitsega (ainult lugemiseks) ja seal olid programmid nii riistvara käivitamiseks kui ka kasutamiseks – viimased

1 *Flat binary file* – [CD] järgi on lamefail samatüübiliste kirjete kogum, kus ei kasutata viitu: kirjed koosnevad lihtväljadest. *CRT* (C *RunTime*) funktsioon *fopen* kasutab seda tüüpi failide lugemiseks režiimi „*rb*“ ja kirjutamiseks „*wb*“. Tavaliselt interpreteeritakse kahendfaili baite 8-bitiste täisarvude ja mitte *ASCII*-sümbolite koodidena.

2 *Crt* on lühend teegi C *Run-Time Library* nimest; sellest teegist vt. lähemalt näit. [Crt L].

3 Me imiteerime neid variante „oma“ *NASM*i abil lisas 2.

4 *Microsoft Disk Operation System*

funktsioonidena, mille poole sai assemblerprogrammist pöörduda katkestus-direktiivi ***int*** (*interrupt*) abil[1]. Näiteks, kui kirjutasime registrisse *AH* väärtuse 0 ja andsime käsu ***int 16h*** (klaviatuuri sisend), siis loeti klaviatuurilt sümbol registrisse AH. Seejuures registri *AL* väärtus näitas, kas loeti *ASCII*-sümbol või *scan*-kood (näit. *Ctrl+c*).

Alates 32-bitisest arhitektuurist pole *BIOS*-katkestused enam assembler-programmides toetatud (võimalikud) – peaaegu kõik rakendusprogrammile vajaliku annavad *crt*-funktsioonid. *BIOS* ise on *ROM*-mälust liikunud omaette kiipi arvuti emaplaadil.

Sootuks teine lugu on *MS-DOS*i funktsioonidega. Assembleris sai ka neid välja kutsuda katkestuse (*interrupt 21*) abil. Alates 32-bitisest arhitektuurist pole selleks enam otsest vajadust. Mõned op-süsteemid – näit. Linux – on millegipärast selle katkestuse-võimaluse[2] säilitanud.

4.3. PROGRAMMI ÜLESEHITUS

*NASM*is 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 *NASM*i kommentaari mõjuulatus on nagu C'//'omal; kommentaar algab semikooloniga ';'.
- C-s programmeerides tuleb seejärel kirjutada makrod, tavaliselt ja vältimatult vajalike moodulite kirjeldusteekide deklaratsioonid *#include<...h>* ja – kui see on otstarbekas, ka kasutaja-makromäärangud *#define*. *NASM*is pole vaja (ega võimalikki) deklareerida standardteeke (*%include* on teksti kopeerimiseks kasuta-

1 Vt. näit.[int21].

2 Katkestusi (nii riist- kui ka tarkvara omi) püüab kinni op-süsteem. Reageerimise variandid:
- Katkestust eiratakase.
- Lõpetatakse kohe käimasoleva protsessi täitmine ja teenindatakse katkestuse põhjustajat.
- Uus protsess võetakse vahele ja pärast seda jätkatakse vanaga.
- Uus protsess pannakse ootele ja võetakse ette pärast käimasoleva lõppu.

tav – sellest hiljem), ent *#define* võimalusi pakub *%define*.

- C *#include*-teekide moodulid tuleb *NASM*is ükshaaval deklareerida neid välisnimedeks (*extern*) kuulutades, lisades nimele prefiksi '_'. Näiteks, kui kasutame C *stdio*-teegist moodulit printf, siis *NASM*is tuleb kirjutada *extern _printf*. Oma programmi *main*-moodul (*exe*-faili sisendpunkt) tuleb samas deklareerida kui *global _main*. Miks: *Linker* otsib (ja leiab) ainult selliseid nimesid – prefiksiga „_".
- C-tekstis võib makrodele järgneda (ja mooduli(te) teksti(de)le eelneda) osa, kus kirjeldatakse globaalseid muutujaid ja andmestruktuure ning võib eeldeklareerida mooduleid. *NASM*is moodulite eelkirjeldusi pole (nende järjekord tekstis on suvaline), küll aga saab kirjutada üks või kaks „sektsiooni": üks globaalsete konstantide ja eelväärtustatud muutujate jaoks – see on *section .data*[1] – ja teine globaalsetele muutujatele mälu reserveerimiseks – *section .bss*[2].
- *.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*;

1 Vajadusel saab „muutmiskeeluga muutujate" jaoks teha eraldi *section .const*. Märkigem, et nii saab programmeerja kaitsta ennast iseenda vigade eest.

2 *BSS* (*Block Started by Symbol*) on termin, mille võtsid 1950.-te keskel kasutusele *United Aircraft Corporation*is arvuti IBM704 assembleri teinud Roy Nutt, Walter Ramshaw jt. [wbss].

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

'*times n*' on fakultatiivne võimalus algväärtsustatud vektori deklareerimiseks. 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 kuueteistkümnendarvuna (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];*

  Struktuurid tuleb kirjeldada ka selles sektsioonis. Näiteks, C-keeles võime otsimis-järjestamiskahendpuu tippu kirjeldada nii:

```
struct tipp{
   char key[32];
   struct tipp *v;
   struct tipp *p;
};
```

  ja struktuuri mahu baitides saame *sizeof(struct tipp)* abil.

  *NASM*is tuleb kirjutada:

```
struc tipp
   .key resb 32
   .v   resd 1
   .p   resd 1
endstruc
```

  ja C *sizeofi* asemel tuleb kirjutada *tipp_size.*

- Sektsioonis .*text*[1] on assembleri direktiivid, mis „tõlgitakse" masinkoodi käskudeks. Intel x86 on üheaadressiline masin: käsukoodi

1 Selle sektsiooni teine nimevariant on .*code*– ent „koodi" all mõistavad „vana kooli" programmeerijad siiski masinkoodi. Assembleri direktiivid ongi sisuliselt *ASCII*-tekst.

formaat on kas:

Kood register register (käsus on 0 aadressi) või

Kood register mäluaadress (käsus on 1 aadress) või

Kood mäluaadress register (käsus on 1 aadress) või

Kood mäluaadress vahetu operand (käsus on 1 aadress) või

Kood register vahetu_operand (0 aadressi) või

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

Kood mäluaadress (näit. *loop* <*etikett*>, käsus on 1 aadress) või

Kood näit. *ret*.

Käskudes adresseeritav mäluväli on kas *.data*- või *.bss*-sektsioonis, suunamiskäskude puhul on mäluaadressi rollis selle käsu aadress, kuhu antakse juhtimine. Assembleris kasutatakse mäluaadressidena *etikette* (C-keeles märgendeid). *NASM*is 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 sellel 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*.

Ja *.bss*-sektsioonis võime reserveerida välja 8 baidile (*qword*), aga kirjutada sinna lühemaid asju – mida just, see tuleb näidata pikkuseatribuudiga (*byte* – bait, *word* –2 baiti, *dword* – 4 baiti).

Pikkusatribuut on oluline, kui me lükkame midagi magasini: 32-bitises

režiimis on magasini „laius“ 4 baiti[1] ning sinna saab panna ainult nii „laiu“ asju – viitu ning *dword*-arvusid või 32-bitiseid registreid. Aga, kui me tahame viia magasini registrite AL või AX sisu, siis tuleb „pushida“ terve register EAX.

1 Või neljakordne baitide arv – näit., kui mudel on Win64.

5 MAGASIN (STACK)

5.1. APARATUURNE MAGASIN

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

Friedrich Ludwig "Fritz" Bauer (1924 - 2015)

Programmeerimises on magasini tekitamine ja kasutamine vana võte, ent sootuks uue rolli andis *LIFO*-tüüpi magasinile selle protsessoripoolne toetus – *aparatuurne magasin*[1]. Mälus on ta tavalise piirkonnana (omaette segmendis), aparatuurseks teeb magasini protsessori ning masinkoodipoolne toetus. Üld- registrite hulgas on *esp* (viit magasini tipule) ja *ebp* (freimi baasi hoidmiseks) ja käsud *push*, *pop*, *call* ning *ret*.

Aparatuurse magasini „laius“ (elemendi pikkus) on määratud protsessori „bittide arvuga“, 32-bitise mudeli puhul on see 4 baiti[2]. Vaatleme allpool selle

1 Arvatavasti oli esimene aparatuurse magasini evitaja miniarvuti PDP-11. (vt. näit. [Isotamm, PK]). Inseneridena mainitakse kaht nime: Edson de Castro ja Harold McFarland. [PDP].

2 See on nii, kui formaat on win32. Kui töörežiim on 64-bitine (win64), siis on „laius“ ka 8 baiti, näiteks kui alamprogrammile antakse parameetrina ette *rbx* (*push rbx*), siis pärast alamprogrammi tööd tuleb

magasini kasutamist.

- 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 allapoole
  mov [esp],ebp      ;registri ebp sisu → magasin
  ```

  pop ebx:

  ```
  mov [ebx],esp      ;magasini tipust viit esp registrisse ebx
  add esp,4          ;"kustutan" magasini tipmise elemendi
  ```

  Ja samuti nagu *push* ja *pop* nihutavad magasiniviita *esp*, saab seda teha vajadusel ka programmis, direktiividega *sub* ja *add*: nende argument on nihke suurus baitides[1]; *sub* nihutab viita allapoole (freimi maht kasvab) ja *add* ülespoole (freimi maht kahaneb):

- 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 baasaadress magasini (lahtikirjutatult: sub esp,4 mov[esp],ebp) ning väljakutsutud mooduli freimi baasaadressiks

täita direktiiv *add rsp,8*.

1 4 või 8 baiti

registris ebp saab viit magasini tipule (mov ebp,esp). Freimi baas on omamoodi "püsipunkt":

[ebp] = freimi baas;

[ebp+4] = naasmisaadress;

Kui alamprogrammi kõik parameetrid või osa neist on pandud magasini, siis neist esimese aadress on

[ebp+8] = 1. argument;

[ebp+12]= 2. argument

jne.

- **Käsurea-parameetrid** edastatakse juhtprogrammile (*main*-moodulile) samamoodi 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.

- 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 kutsuda", siis võime moodulit (näiteks P, nonde a ja b-ga) alustada nii:

  P:

  %define a dword[ebp-4]

  %define b dword[ebp-8][1]

  push ebp

  ...

  Enne alamprogrammist väljumist peame magasini oma töömuutujat-

1 Omistamine a=a+b on nüüd assembleris järgmine:
mov eax,a ;mov eax,dword[ebp-4]
add eax,b ;add eax,dword[ebp-8]

est puhastama, meie näite puhul käsuga
add esp,8

- Alamprogrammist väljumise standardsed käsud on
  pop ebp ;taastame väljakutsuja freimi baasi
  ret ;sisuliselt: pop eip
- Väljakutsutud moodul võib oma vajadustele vastavalt kasutada kõiki üldregistreid, ent tavaliselt 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 tsükli-direktiivi kujul *loop* märgend) tuleb kanda ecx-registrisse, ja kui tsükli sees on pöördumine mõne *crt*-mooduli poole, siis tuleb olla valmis selleks, et see moodul rikub ära ecx-registri ja koos sellega ka meie tsükli. Selle vältimiseks tuleks kirjutada näiteks nii:
  ```
  ring:
     push ecx
     …
  pop ecx
  loop ring
  ```

- Rõhutame: väljakutsutud moodul (*callee*) peab enne juhtimise tagastamist väljakutsujale (*caller*) "puhastama" kogu oma freimi. Kui me programmeerides jätame magasini midagi ülearust, siis pole *ret*-käsu täitmise ajal magasini tipus naasmisaadressi – seal on midagi muud – ning programm lõpetab avariiliselt.

5.2. INTEL X86 KUTSEVARIANDID

Assemblerid on põhimõtteliselt orienteeritud toetama moodulprogrammeerimist. 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. iseennast. Assemblerid arvestavad neid seiku ning moodulite poole pöördumiseks ning nendega info vahetamiseks on mitmeid mooduseid.

Kutse (*call*) on „Arvutikasutaja sõnastiku" [AKS] järgi „juhtimise üleandmine ühest programmimoodulist teise, koos naasmisjuhistega". *Kutsevariandid* on i.k. *calling conventions*. Nende paljus on seletatav sellega, et mikroprotsessorite esimesed tootjad ei teinud ei op-süsteeme ega ka kõrgtaseme keelte (näit. C) translaatoreid, seega – ei kehtestanud standardeid.

Selles jaotises püüame anda lühiülevaate erinevatest variantidest, mida on selle mikroprotsessori jaoks programmeerimiskeelte realiseerijad (translaatorite kirjutajad) valinud. Tugineme eeskätt allikale [wcc], kus nood variandid jagatakse kahte klassi, ksj. on olulised mõisted *caller* (pöörduja, väljakutsuja, ülemus jmt.) ja *callee* (alamprogramm). Jaotuse aluseks on, „kes" koristab magasinist alamprogrammi parameetrid, kas alamprogrammi pöörduja või alamprogramm ning kuidas (kus) edastatatakse alamprogrammile tema parametrid.

Intel x86 dikteerib oma käskude süsteemiga tegelikult üsna paljut:

- Kogu käskude süsteem toetab moodulprogrameerimist: pikkade liigendamate programmi-de asemel soositakse ühte ülesannet täitvate alamprogrammide (sh. funktsioonide) – moodulite – programmeerimist ning nendevahelist infovahetust.

- Programmeerimine on magasinikeskne ja toetatud registritega *esp* ja *ebp* ning käskudega *push*, *pop*, *call* ja *ret*.
- Programm peab säilitama väljakutsuja freimi baasi ning tegema oma freimi.
- Üldjuhul on alamprogrammi parameetrid magasinis (kui neid on kuni neli, edastatakse mõne kutsevariandi puhul need üldregistrites ja kui neid on rohkem, siis ülejäänud ikkagi magasinis) – väljakutsuja freimis.
- Oma freimi on võimalik reserveerida tööväli lokaalsete (töö)muutujate jaoks.
- Nii parameetrite kui ka töömuutujate adresseerimine toimub freimi baasi suhtes.
- Freim peab alamprogrammi töö lõpetamise ajaks olema puhas: töö alul oli magasini tipus naasmisaadress ja see peab magasini tipus olema ka väljumise hetkel *ret*-käsu täitmiseks.

5.2.1. CDECL

Cdecl (C declarations) on meie jaoks eelistatud variant, kuivõrd just nii on häälestatud meile hädavajalikud C „teegiprogrammid“ – need, mida C-programmides saame kasutada *include*-teekide (*#include<stdio.h>* jne) abil ja *NASM*-programmides. Kokkulepped:

- Parameetrid[1] pannakse „paremalt vasakule“ magasini: *push p5, push p4,...push p1;*
- Alamprogramm saab nad kätte freimi baasi abil, p1 aadress on *dword[ebp+8]* ja p5 oma *dword[ebp+24]*
- Väljakutsuja puhastab magasini alamprogrammi parameetritest, nihutades pärast alam-programmist väljumist *esp*-viita *n* baidi võrra ülespoole (magasini baasi poole); *n=parm x 4* (*parm* on magasini pandud neljabaidiste parameetrite arv).
- Kui alamprogramm on funktsioon, siis selle väärtuse (*int*-arvu või viida) tagastab *alamprogramm eax-registris*. Väljakutsutav programm peab salvestama ja enne väljumist taastama väljakutsuja freimi

1 C-programmis näit. ap(;p1, p2, p3, p4, p5)

baasi, tegema oma freimi ning – kui ta neid kasutab – siis säilitama üldregistrite *ebx, esi* ja *edi* sisud. Lokaalsetele muutujatele saab ruumi reserveerida omas freimis ning sisendparameetrite ja lokaalsete muutujate adresseerimiseks kasutatakse suhtaadresse freimi baasi suhtes. Üldregistrite *ecx* ja *edx* säilitamine alamprogrammi väljakutse puhul on on väljakutsuja enda mure.

- Halb praktika on direktiivide *ENTER* ja *LEAVE* kasutamine. Näiteks, *ENTER 16* on:

```
push ebp
      mov  ebp,esp
      sub  esp,16     ;tööväli 4 lokaalse muutuja jaoks
```

Programmi lõppu kirjutatav *LEAVE* teeb kaks asja:

```
mov  esp,ebp
pop  ebp
```

Sel moel pole vaja alamprogrammi parameetreid elimineerida kohe pärast juhtimise tagasi saamist – seda teeb *LEAVE* – ja programmeerijal pole pikema teksti korral enam ei ülevaadet magasini seisust ega ka kontrolli selle üle. Siinkirjutaja arvates pole see enam *Cdecl*.

ret-direktiiv toimib sisuliselt kui *pop eip*, aga käsuviida-register *eip* pole *ring3*-programmeerijaile kättesaadav. *ret*-i võime näiteks asendada käskudega

```
pop  ecx
jmp  ecx
```

Toome lihtsa näite.

```
;summa.asm :: z=x+y. 9.04.19. Mina Ise
global _main
extern _printf
section .data
   x dd 5
   y dd 77

   pf db “summa=%d”,10,0
   tf db “sum(x,y)=%d”,10,0
section .bss
   z resd 1
section .code
```

```
_main:
   push ebp
   mov  ebp,esp

   mov  eax,[x]  ;x on aadress, [x] on arv aadressilt x
   add  eax,[y]
   mov  [z],eax
   push dword[z]
    push pf
   call _printf
   add  esp,8
;---------------------------------------
;int sum(int x,int y){
;   return(x+y);
;}
   push dword[y]
   push dword[x]
   call sum  ;res on eax-s
   add  esp,8
   push eax
   push tf
   call _printf
   add  esp,8
   pop  ebp
   ret
;---------------------------------------
sum:
   push ebp
   mov  ebp,esp
   mov  eax,dword[ebp+8]
   add  eax,dword[ebp+12]
   pop  ebp
   ret
```

Joonis 5.2.1. *Cdecl*: summa

5.2.2. SYSCALL

syscall sarnaneb üldjoontes *cdecl*-ile. Lisakokkulepe on, et registris *eax*[1] edastatakse alamprogrammile ta parameetrite arv. See variant on väga hea juhul, kui alamprogramm on orienteeritud määramata parameetrite arvule. Vt. C-teeki *stdarg.h* raamatutest [K&R] lk. 254 või [Isotamm, C] lk.123. Selliste moodulite näideteks sobivad *printf* ja *fscanf*.

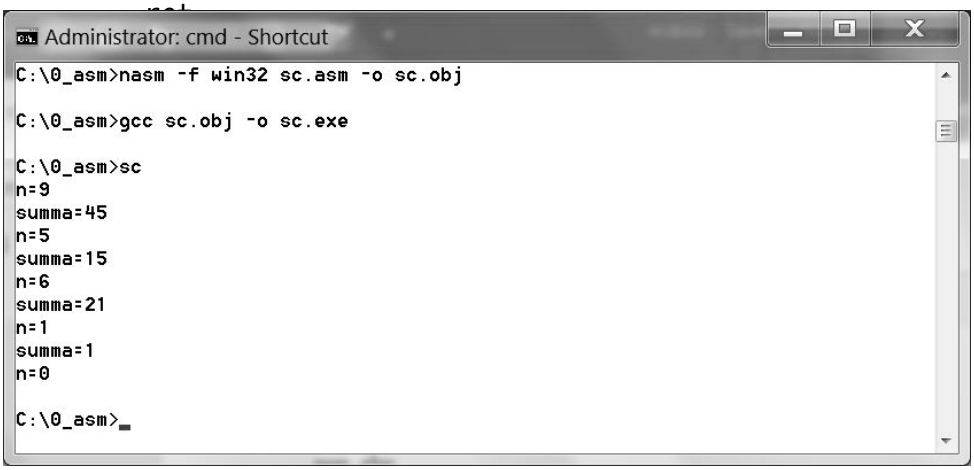
Toome siinkohal näiteks programmi, mis leiab *n* esimese naturaalarvu summa.

```
;sc.asm :: syscall, stdarg.h. 6.05.19. Mina Ise

global _main
extern _printf
extern _gets
extern _atoi

section .data
   tf db “summa=%d”,10,0
   kysi db ‘n=’,0

section .bss
   a resb 7
   n resd 1
  z resd 1

section .code
_main:
```

1 Kui täpne olla, siis registris *al* edastatakse enna alamprogrammi pöördumist magasini pandud neljabaidiste elementide arv.

```
    push ebp
    mov  ebp,esp
    push esi
uus:
    push kysi
    call _printf
    add  esp,4
    push a
    call _gets
    add  esp,4
    push a
    call _atoi
    add  esp,4
    cmp  eax,0
    je   aut
    mov  dword[n],eax
    mov  ecx,eax
    mov  esi,1
p:
    push esi
    inc  esi
    loop p
    mov  eax,dword[n]  ;parv
    call sum  ;res on eax-s
    mov  dword[z],eax
    mov  eax,dword[n]
    shl  eax,2  ;parv*4
    add  esp,eax
    push dword[z]
    push tf
    call _printf
    add  esp,8
    jmp  uus
aut:
    mov  eax,0
    pop  esi
    pop  ebp
```

```
   ret
;---------------------------------------
sum:
   push ebp
   mov  ebp,esp
   push esi
   mov  esi,8 ;1. para
   mov  ecx,eax  ;parv
   xor  eax,eax
ring:
   add  eax,dword[ebp+esi]
   add  esi,4
   loop ring
   pop  esi
   pop  ebp
   ret
```



Joonis 5.2.2. Syscall: *n* esimese naturaalarvu summa.

5.2.3. MUID VARIANTE

Kahele ülaltutvustatud variandile sarnaneb **optlink** selle poolest, et magasini puhastab väljakutsuja, ent parameetreid edastatakse „vasakult paremale“, kolm esimest registrites *eax*, *edx* ja *ecx* ning ülejäänud magasinis. Näiteks alamprogrammi *ap*(a,b,c,d,e) välja kutsumiseks tuleb kirjutada

```
   mov  eax,a
   mov  edx,b
```

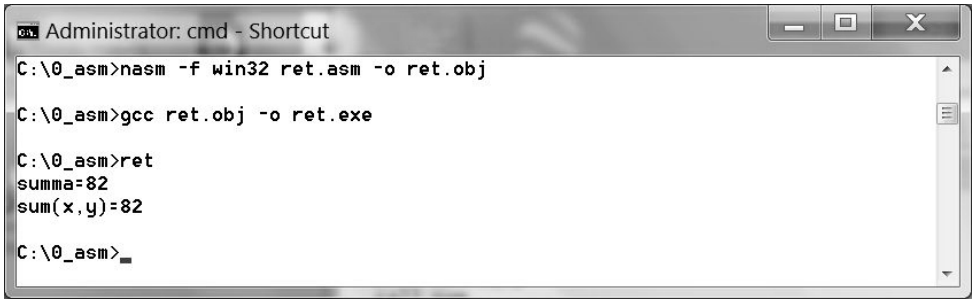
```
   mov  ecx,c
   push d
   push e
   call   ap
   add   esp,8
```

Järgmiste variantide puhul puhastab magasini alamprogramm, kasutades *ret*-instruktsiooni fakultatiivset 16-bitist parameetrit „baitide arv *n*“. Näiteks *ret 8* nihutab magasini tipu viita (*esp*) 8 baidi võrra. Tuntumatest variantidest mainigem järgmisi:

stdcall sarnaneb *cdecl*-ile, erinevus on magasini puhastamise võttes. See on Microsofti standard, mida kasutab Win32 *API* (*Application Program Interface*). Järgmine programm on kirjutatud *stdcall*i võtmes:

```
;ret.asm :: z=x+y. 9.04.19. Mina Ise
global _main
extern _printf

section .data
   x dd 5
   y dd 77
   pf db “summa=%d”,10,0
   tf db “sum(x,y)=%d”,10,0

section .bss
   z resd 1

section .code
_main:
   push ebp
   mov  ebp,esp
   push dword[y]
   push dword[x]
   call sum
   push eax
   push pf
   call _printf
   add  esp,8
```

```
   push dword[y]
   push dword[x]
   call sum2
   push eax
   push tf
   call _printf
   add  esp,8

   pop  ebp
;”ret” asemel suuname ise naasmisaadressile
   pop  ecx
   jmp  ecx
;parameetriga legaalne ‘ret’
sum:
   push ebp
   mov  ebp,esp
   mov  eax,dword[ebp+8]
   add  eax,dword[ebp+12]
   pop  ebp
   ret  8
;’ret 8’ imitatsioon
sum2:
   push ebp
   mov  ebp,esp
   mov  eax,dword[ebp+8]
   add  eax,dword[ebp+12]
   pop  ebp
   pop  ecx   ;naasmisaadress => ecx
   add  esp,8 ;magasini puhastamine
   push ecx   ;naasmisaadress magasini tippu
   ret        ;”pop eip”
```



Joonis 5.2.3. *Stdcall*: alamprogramm puhastab magasini.

Microsoft *fastcall* nõuab, et esimesed kaks (vasakult paremale) parameetrit edastatakse registrites *ecx* ja *edx* ning ülejäänud paremalt vasakule magasinis; need eemaldab sealt alamprogramm.

Segavariant ***thiscall***, mida kasutab näit. Microsoft Visual C++

5.3. REGISTRITE KOKKULEPPED

Üldregistritega käituvad tuntumad pöördumisvariandid ühtemoodi. Säilitada tuleb registrite *ebp, ebx, esi* ja *edi* pöördumisaegne seis. Registris *eax* tagastatakse reeglina funktsiooni väärtus ning üldjuhul „rikuvad“ alamprogrammid ära registrid *ecx* ja *edx*.

Registri *ecx* „kaitsetus“ on tülikas juhul, kui *loop*-direktiivi abil tehtud tsüklis pöördutakse alamprogrammi(de) poole. *Loop* kasutab tsükliloendajana *ecx*-registrit, mille alamprogramm ära rikub. Sel juhul tuleb tsükli alguses panna *ecx* magasini ja enne *loop*-direktiivi taastada:

```
...
   mov  ecx,dword[n]
ring:
   push ecx
  ...
   call ap
   ...
   pop  ecx
   loop ring
```

6 OPERANDID

6.1. VÕIMALUSED. METAKEEL

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 näiteid just selleks otstarbeks kirjutatud programmist ja ta lahendamiste tulemustest.

```
;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
```

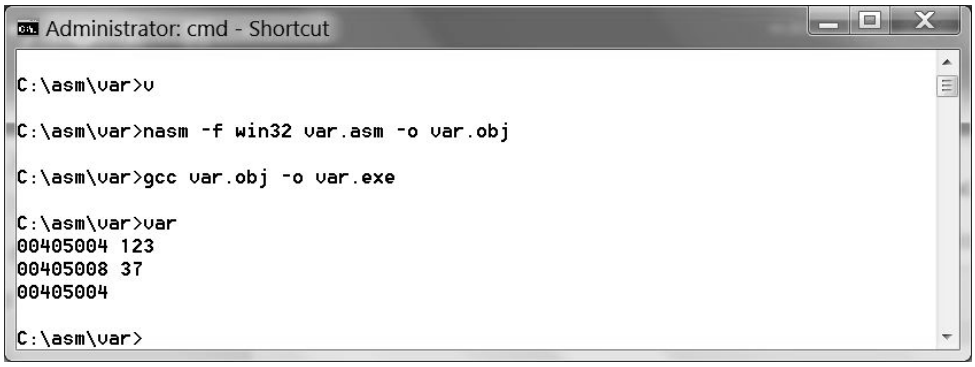
```
        push n          ;4-baidine viit väljale n
        push pn
        call _printf
        add  esp,12
      pop  ebp
        ret
```

Joonis 6.1.a. Viidad ja väärtused.

Uute näidete toomiseks lisasime kord-korralt meie programmi uusi lõike; järgmises 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 6.1.b.



Joonis 6.1.b. 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 6.1.c. 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 6.1.d. 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:

```
Administrator: cmd - Shortcut
37

C:\asm\var>v

C:\asm\var>nasm -f win32 var.asm -o var.obj
var.asm:48: error: operation size not specified
```

Joonis 6.1.e. 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, modifitseeritakse aadressi viitamaks vektori teisele elemendile (direktiivis „add eax,4“ on 4 *vahetu operand*), kirjutatakse sinna arv 77, lisatakse see magasini (ja trükitakse välja):

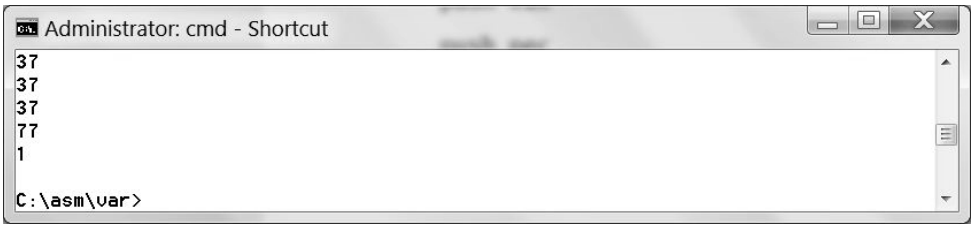
```
    mov   eax,v
    add   eax,4
    mov   dword[eax],77   ;mov [eax],77 annab tuttava vea
    push  dword[eax]
```

```
Administrator: cmd - Shortcut
00405004 123
00405008 37
00405004
37
37
37
77
```

Joonis 6.1.f. 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 6.1.g. Trükitud sümbol.

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

```
mov  eax,byte[a]
```

Joonis 6.1.h. 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

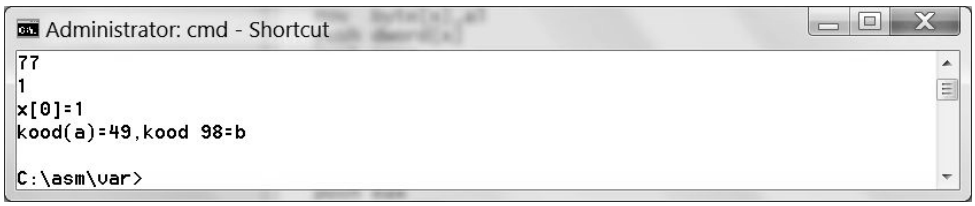
```
    mov  al,byte[a]
mov  byte[x],al
push dword[x]
push pecx
call _printf
add  esp,8
```

Tulemus on joonisel 6.i.

Joonis 6.1.i. 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
```

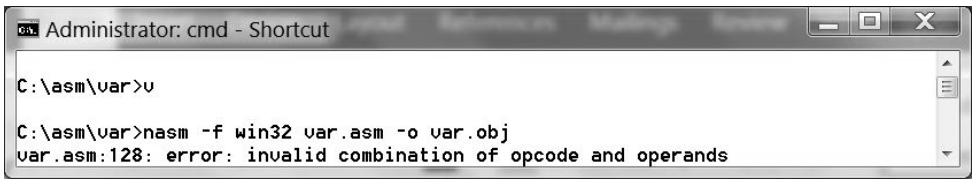


Joonis 6.1.j. Koodide liitmine registrites.

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 6.1.k. Aadressi üleskirjutamise katse.

6.2. KORRUTAMINE JA JAGAMINE

Lõpetame selle peatüki iseenesest lihtsate, ent tavaliselt tarbetult keeruliselt kirjeldatud ja seleta-tud korrutamise ja jagamisega. „Segaseks“ teeb need tehted tõik, et 32-bitiste operandide puhul kasutatakse kahte registrit ning et seda tähistatakse kui EDX:EAX. James T. Streib [Streib] seletab need asjad ära.

6.2.1. KORRUTAMINE

Märgita täisarvude jaoks on korrutamiskäsk *mul* ja märgiga – *imul*. Mõlema variandi puhul on esimene operand (korrutatav) registris eax, teine operand (korrutaja) on kas registris või mälus. Vahetu operand on keelatud (*error: invalid combination of opcode and operands*). Vahetu operand on kasutatav, kui käsku *imul*[1] kirjutada kolm operandi: *eax* resultaadi jaoks, korrutatav (register või mälu) ning korrutaja vahetu operandina. Ülalpool märkisime, et korrutis on registrites EDX:EAX. Seletatakse seda nii, et korrutis võib olla suurem kui 32 bitile mahub ning madalamad järgud on *eax*-s ja kõrgemad *edx*-s. Mis on üsnagi mõttetu, seda korrutist ei saa 32-bitise režiimi korral kuidagi kasutada[2]. Kui korrutis on normaalse (kuni 32-bitise) pikkusega, siis märgiga korrutamine *imul* kirjutab registrisse *edx* tulemuse märgi: positiivse resultaadi puhul on kõik *edx*i bitid nullid ja negatiivse puhul ühed (väärtus „-1“). See veidrus on seletatav käsitluse ühtsuse säilitamisega: 16-bitiste operandide (*ax* ja muu) korrutis paigutatakse 32-bitises masinas registrisse *eax* ja kaheksabitiste operandide (*al* ja teise 8-bitine operandi) korrutis kanti registrisse *ax*. Niisiis, pidagem meeles: korrutamine rikub registri *edx* sisu. Korrutamise testprogramm:

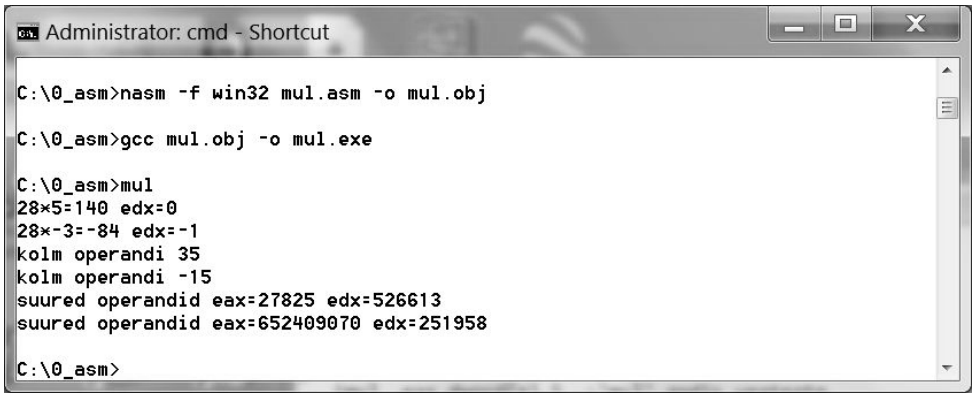
```
;mul.asm :: korrutamine. 19.05.19.
global _main
extern _printf
section .data
   x dd 28
   y dd 5
```

1 Märgita korrutamisel mul kolme-operandi-formaati pole.

2 Muide, see on varjatud ohukoht: kasutuskõlblik korrutis peab mahtuma neljale baidile või peab korrutamine andma ületäitumise ent masinas kantakse mittemahtuvad bitid edx-i. Seda teades saab ületäitumist ise kontrollida.

```
   z dd -3
   rp db ‘28*5=%d edx=%d’,10,0
   rp2 db ‘28*-3=%d edx=%d’,10,0
   rp3 db ‘kolm operandi %d’,10,0
   rp4 db ‘suured operandid eax=%d edx=%d’,10,0
section .bss
   res  resd 1
section .text
_main:
   push ebp
   mov  ebp,esp
   mov  eax,[x]
   mov  ecx,[y]
   mul  ecx
   push edx  ;mida korrutamine sinna kirjutas?
   push eax
   push rp
   call _printf
   add  esp,12
;-----------------------------------------------
   mov  edx,0
   mov  eax,[x]
   imul  dword[z]
   push edx  ;mis seal on?
   push eax
   push rp2
   call _printf
   add  esp,12
;-----------------------------------------------
;   mov  eax,[x]
;   mul  2   siit tuli veateade
;-----------------------------------
   mov  ecx,7
   imul  eax,ecx,5  ;’mul’ andis veateate
   push eax
   push rp3
   call _printf
```

```
   add  esp,8
;-------------------------------------------
   imul  eax,dword[z],5  ;'mul' andis veateate
   push eax
   push rp3
   call _printf
   add  esp,8
;----------------------------------------
   mov  ax,12345
   mov  cx,12345
   mul  cx
   push edx  ;"ületäitumine"
   push eax
   push rp4
   call _printf
   add  esp,12
;-----------------------------
   mov  eax,12345678
   mov  ecx,87654321
   mul  ecx
   push edx  ;"ületäitumine"
   push eax
   push rp4
   call _printf
   add  esp,12
;-----------------------------
   pop  ebp
   ret
```



Joonis 6.2.1.a Korrutamisnäited.

6.2.2. JAGAMINE

Jagamistehte puhul on jagatav registrites „EDX:EAX“, mis meie 32-bitise tavarežiimi puhul tähendab, et jagatava kanname registrisse *eax* („madalamad järgud) ja registrisse *edx* (kõrgemad järgud) kanname jagatava märgi 0 positiivse ja -1 negatiivse arvu jaoks[1]. Märgi kandmiseks on mugav kasutada direktiivi *cdq (convert doubleword to quadword).* Resultaat on aga programmeerijasõbralik: jagatise täisosa on registris *eax* ning jääk – *edx*. Seega, C keelest tuttavaid vahendeid *div_t* (struktuurne jagatistüüp väljadega *quot* – täisosa ja rem – jääk), funktsiooni *div* ja jagamistehteid “/“ ning „%“ pole vaja. Üldistatult:

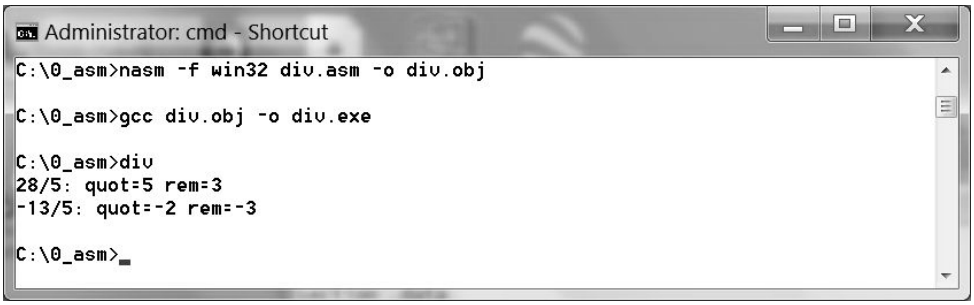
```
mov eax,jagatav
cdq
idiv jagaja
mov  quot,eax
mov  rem,edx
```

Allpool esitame näiteprogrammi *div.asm* teksti ja selle lahendamise pildi.

```
;div.asm :: jagamine. 20.05.19.
global _main
extern _printf
section .data
   x dd 28
   y dd 5
   z dd -13
   rp db ‘28/5: quot=%d rem=%d’,10,0
```

1 Niisiis, 32-bitises masinas on edx jagamisel sarases rollis korrutamisega – jagatava märgi hoidja rollis. Selle registri nullib või täidab ühtedega (mov edx,-1) kas programmeerija -- või käsk cdq.

```
   rp2 db '-13/5: quot=%d rem=%d',10,0
section .text
_main:
   push ebp
   mov  ebp,esp
   mov  eax,[x]
   cdq
   mov  ecx,[y]
   div  ecx
   push edx
   push eax
   push rp
   call _printf
   add  esp,12
;-------------------------------
   mov  eax,[z]
   cdq
   mov  ecx,[y]
   idiv  ecx

   push edx
   push eax
   push rp2
   call _printf
   add  esp,12

   pop  ebp
   ret
```



Joonis 6.2.2.a Jagamine.

6.2.3. KORRUTAMINE JA JAGAMINE "KAHE ASTMEGA"

Meenutuseks, 2^2=4, 2^3=8, 2^4=16 jne. Mugav ja kiire variant on kasutada bitikaupa nihutamist. Kui korrutatav või jagatav arv on *a*, siis näiteks $a \times 8$ väärtus nihutada 3 bitti vasakule ning *a/8* tegemiseks 3 bitti paremale. Üldiselt, kui korrutaja või jagaja on 2^x, siis sobib illustratsiooniks järgmine programmilõik:

```
mov  eax,dword[a]
 mov  ecx, dword[x]
shl  eax,ecx  ;korrutamine arvuga 2 astmes x
mov  eax,dword[b]
shr  eax,ecx  ;jagamine arvuga 2 astmes x
```

6.3. LIHTAVALDIS

Lõpetame selle peatüki programmiga, mis lahendab lihtsaid aritmeetilisi avaldisi. Programm ei kontrolli sisendit ega suuda ise lõpetada – seda tuleb teha *Ctrl+c* abil.

```
;arav.asm :: aritmeetiline avaldis, nt. 7*3. 19.05.19
global _main
extern _printf
extern _scanf
;-----------------------------------------------
section .data
   anna db 'anna aritmeetiline avaldis: ',0
   res db'tulemus=%d',10,0
   scf db '%d%c%d',0
   kf db 'x=%d %c y=%d',10,0
;-----------------------------------------
section .bss
   x resd 1
   y resd 1
   tehe resb 1
;-----------------------------------
section .text
_main:
```

```
   push ebp
   mov  ebp,esp
ring:
   push anna
   call _printf
   add  esp,4
   push y
   push tehe
   push x
   push scf
   call _scanf
   add  esp,16
;--------------------------------
;kontrolltrykk
   push dword[y]
   xor  eax,eax
   mov  al,byte[tehe]
   push eax
   push dword[x]
   push kf
   call _printf
   add  esp,16
;--------------------------------
   mov  al,byte[tehe]
   cmp  al,'+'
   je   liida
   cmp  al,'-'
   je   lahuta
   cmp  al,'*'
   je   korruta
   cmp  al,'/'
   je   jaga
   cmp  al,'%'
   je   jaak
;-------------------------------
liida:
   mov  eax,dword[x]
```

```
   add  eax,dword[y]
   jmp  tryki
   je   jaak
;--------------------------------
lahuta:
   mov  eax,dword[x]
   sub  eax,dword[y]
   jmp  tryki
korruta:
   mov  eax,dword[x]
   mul  dword[y]
   jmp  tryki
;----------------------------------
jaga:
   mov  eax,dword[x]
   cdq
   idiv dword[y]
   jmp tryki
;----------------------------------
jaak:
   mov  eax,dword[x]
   cdq
   idiv dword[y]
   mov  eax,edx
;-----------------------------------
tryki:
   push eax
   push res
   call _printf
   add  esp,8
   jmp  ring
   pop  ebp
   ret
```

```
Administrator: cmd - Shortcut
C:\0_asm>nasm -f win32 arav.asm -o arav.obj

C:\0_asm>gcc arav.obj -o arav.exe

C:\0_asm>arav
anna aritmeetiline avaldis: 2+3
x=2 + y=3
tulemus=5
anna aritmeetiline avaldis: 2-3
x=2 - y=3
tulemus=-1
anna aritmeetiline avaldis: 2×3
x=2 * y=3
tulemus=6
anna aritmeetiline avaldis: 12/5
x=12 / y=5
tulemus=2
anna aritmeetiline avaldis: 12%5
x=12 % y=5
tulemus=2
anna aritmeetiline avaldis: x=12 % y=5
tulemus=2
anna aritmeetiline avaldis: ^C
C:\0_asm>
```

Joonis 6.3.a. Aritmeetika.

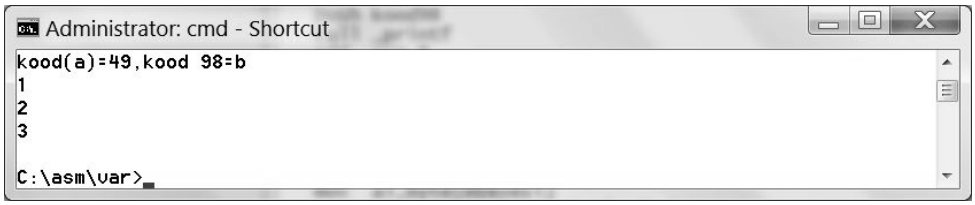
7 INDEKSEERIMINE

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  ;tsükliloendaja taastamine
   loop ring
```



Joonis 7.a. 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[1]. Meie näiteprogrammis on reserveeritud ruum C mõttes *int*-vektorile *v*.

Järgmise programmilõiguga omistatakse selle vektori 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,1*.

Joonis 7.b. Tsükkel üle int-vektori.

1 Masinkoodi-formaadis on kordajale reserveeritid 2 bitti, võimalikud väärtused on 0, 1, 2 ja 3. Täiesti läbinähtavalt kasutatakse neid väärtusi indeksi väärtuse nihutamiseks vasakule (vt. ka meie jaotist 6.2.3).

8 PÕHI- JA ALAMPROGRAMM

Allpool vaatleme põhi- ja alamprogrammi(de) kirjutamise ja transleerimise variante. Lihtsam juht on, kui nii põhi- kui ka alamprogramm(id) on kirjutatud samas keeles (meie raamatu kontekstis kas C- või assemblerkeeles), ent programmeerija jaoks on veel olulisemad C ja assambleri programmide vastastikuste seostamise võimalused: kuidas kasutada C-programmis assembleris kirjutatud mooduleid ja vastupidi – assemblerprogrammis C-mooduleid. Seda kombineerimist on mõnikord nimetatud „ristkasutuseks". Näiteprogrammina kasutame triviaalset kahe arvu summa leidmise ja trükkimise „koodi", kusjuures summeerimine on programmeeritud eraldi funktsioonina.

Siinkohal tutvustame põgusalt programmide järjestikust transleerimist ning komplekteerimist hõlbustavat skriptikeelt pakkfailide[1] moodustamiseks. See keel on iseenesest üsna võimalusterohke (sj. võimalusega parameetreid kasutada, tingimusi kirjutada jmt.)[2], ent meile aitab ta miinimumvahenditest. Kirjutada tuleb tavaline *ASCII*-teksti fail nimelaiendiga *.bat* ja sinna eraldi ridadele need korraldused, mis tuleks *DOS*i (*UNIX*i) aknas ükshaaval sisestada. Pakkfaili käivitamisel täidetakse need korraldused üksteise järel.

Seejuures tuleb silmas pidada, et käsurealt käivitatakse pakkfail nimelaiendit *.bat* arvestamata – just samuti, nagu teeb süsteem *.exe*-failidega. Seejuures, .exe on prioriteetsem kui .bat, ja kui olete teinud failid *minu.bat* ja *minu.exe*, siis pole teil võimalusi *.bat*-faili käivitada.

Ja veel – *.bat*-faili teksti kuvamiseks (näit. parandamiseks) ei sobi tavaline

1 „Pakk" paki mõttes – ühte pakki on pandud järjestikku täidetavad käsurea-korraldused.

2 *Batch*-failist veelgi võimalusterohkem on selle edasiarendus – *Makefile*-komplekt. Huvilised leiavad neist Google'i abil ammendava teabe.

hiireklikk – see käivitab pakkfaili – vaid tuleb kasutada paremat hiireklahvi ning variante *Edit* või *Open with*.

Toome näiteks faili *d.bat* sisu:

```
nasm -f win32 summa.asm -o summa.obj
gcc summa.obj demo.c -o demo.exe
```

Pakkfaili käivitamisel kuvatakse ekraanile järjest kõik selle read; sinna väljastavad ka käivitatud programmid oma teated. Selle väljundi saab „ära keelata“, lisades pakkfaili algusse vastava rea:

```
@echo off
nasm -f win32 summa.asm -o summa.obj
gcc summa.obj demo.c -o demo.exe
```

Ent – arvestades meie raamatu sihtgrupiga – sellist info peitmist pole vaja. Üldse: algaja jaoks on hea, kui tema eest midagi ei peideta (mida teevad näit. direktiivid *ENTER*...*LEAVE* jms.)

8.1. ÜKS C-FAIL

Funktsioon on kirjeldatud põhiprogrammiga samas tekstifailis.

```
//demo.c :: ristkasutus, liht-C
#include<stdio.h>

int summa(int x,int y){
   return(x+y);
}

int main( ){
   int sum;
   sum=summa(2,3);
   printf(“summa on %d\n”,sum);
}
```

```
Administrator: cmd - Shortcut

C:\Praxid\cp17>cd 14_cross\c

C:\Praxid\cp17\14_cross\c>gcc demo.c -o demo.exe

C:\Praxid\cp17\14_cross\c>demo
summa on 5

C:\Praxid\cp17\14_cross\c>
```

Joonis 8.1. Ainult üks C-tekstifail.

8.2. C PÕHI- JA ERALDI ALAMPROGRAMM

Siin kasutab põhiprogramm varemkirjutatud C-funktsiooni *summa*, järgmise tekstiga:

```
//summa.c :: demo.c eraldi transl. alamprog.
int summa(int x,int y){
   return(x+y);
}
```

Põhiprogrammis *demo.c* on selle funktsiooni *summa* kirjeldus.

```
//demo.c :: ristkasutus, liht-C
#include<stdio.h>

int summa(int x,int y);

int main(){
   int sum;
   sum=summa(2,3);
   printf("summa on %d\n",sum);
}
```

```
Administrator: cmd - Shortcut
C:\Praxid\cp17\14_cross\cc>gcc -c summa.c -o summa.obj

C:\Praxid\cp17\14_cross\cc>gcc summa.obj demo.c -o demo.exe

C:\Praxid\cp17\14_cross\cc>demo
summa on 5

C:\Praxid\cp17\14_cross\cc>
```

Joonis 8.2. Eraldi transleeritud C alamprogramm.

Pöörake joonisel tähelepanu lipule *-c* teksti *summa.c* transleerimisel – selle puudumisel antaks veateade, et transleeritavas programmis pole *main*-moodulit ja programmi sisendpunkti ei saa fikseerida.

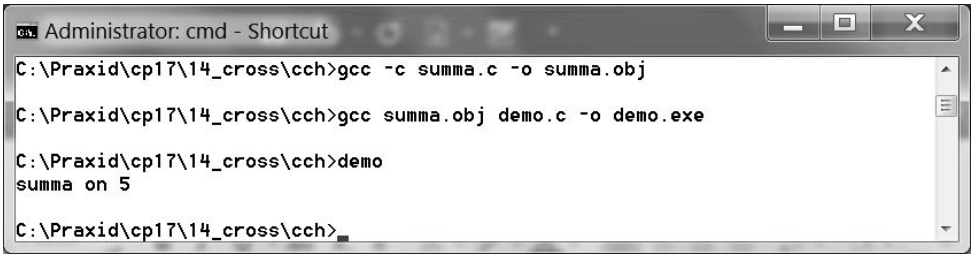
8.3. C PÕHI- JA ERALDI ALAMPROGRAMM + PÄISFAIL

Päisfail s.h sisaldab ainult funktsiooni summa kirjeldust. Juhime tähelepanu päisfaili kirjutamise metakeelele nime s.h esitamisel. Makro *ifndef* tähendus on *if not defined* (kui pole defineeritud) ning *endif* on *ifndefi* lõpetav operaatorsulg. Mõte on selles, et preprotseesor kopeerib defineeritud teksti ainult ühte (esimesena ettejuhtuvasse) makrot *#include* „s.h“ sisaldavasse C-teksti.

```
#ifndef __S_H__
#define __S_H__
int summa(int x,int y);
#endif
```

Põhiprogrammi *demo.c* ja alamprogrammi *summa.c* on järgmised:

```
//summa.c :: demo.c eraldi transl. alamprog.
#include “s.h”

int summa(int x,int y){
   return(x+y);
}
//demo.c :: ristkasutus, liht-C
#include<stdio.h>
#include “s.h”

int main(){
   int sum;
   sum=summa(2,3);
   printf(“summa on %d\n”,sum);
}
```

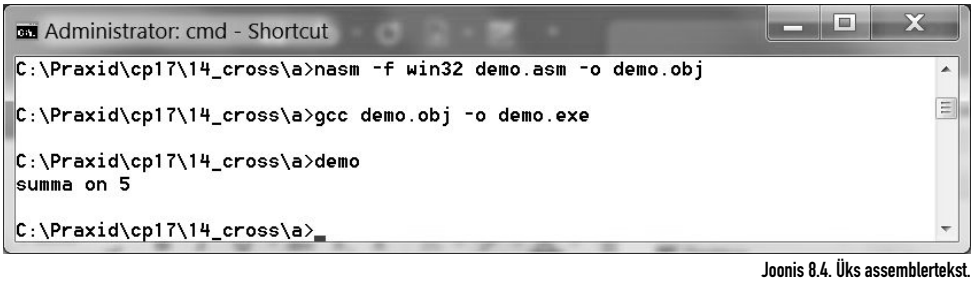


Joonis 8.3. Päisfailiga variant.

8.4. ÜKS ASSEMBLERFAIL

```
;demo.asm :: lihtasm. print(x+y)
global _main
extern _printf

section .const
   pr db 'summa on %d',10,0

section .code
_main:
   push ebp
   mov  ebp,esp

   push 2  ;y
   push 3  ;x
   call summa
   add  esp,8
   push eax  ;summa on eax-s
   push pr
   call _printf
   add  esp,8
   pop  ebp
   ret
;-----------------------------------
summa:
   push ebp
   mov  ebp,esp
   mov  eax,dword[ebp+8] ;x
```

```
   add  eax,dword[ebp+12] ;y
   pop  ebp
   ret
;-------------------------------------
```

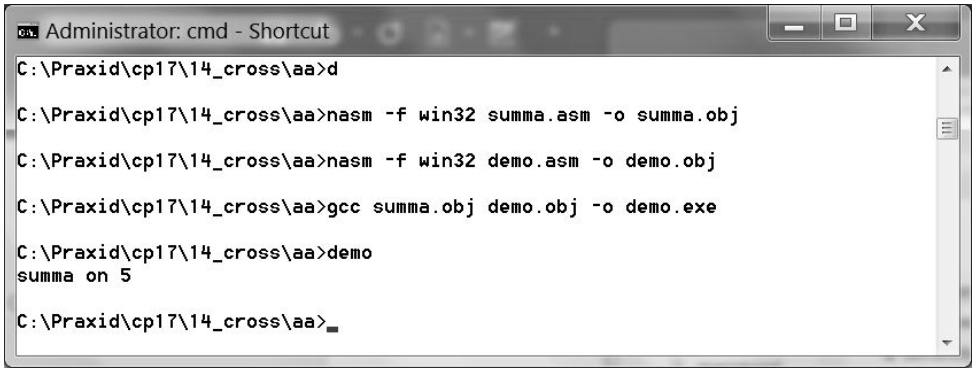


Joonis 8.4. Üks assemblertekst.

8.5. KAKS ASSEMBLERFAILI

```
;summa.asm :: funktsiooni tekst.
global _summa
section .code
_summa:
   push ebp
   mov  ebp,esp
   mov  eax,dword[ebp+8] ;x
   add  eax,dword[ebp+12] ;y
   pop  ebp
   ret
;-------------------------------------
;demo.asm :: z=summa(x,y),  print(z). Summa on eraldi
;transleeritud.
global _main
extern _printf
extern _summa

section .const
   pr db ‘summa on %d’,10,0

section .code
_main:
```

```
        push ebp
        mov  ebp,esp

        push 2  ;y
        push 3  ;x
        call _summa
        add  esp,8
        push eax  ;summa on eax-s
        push pr
        call _printf
        add  esp,8
        pop  ebp
        ret
;-----------------------------------
```



Joonis 8.5. Kaks assemblerfaili.

8.5.1. PÕHIPROGRAMM JA LISATUD TEKST

Makroga *include* saab lisada assemblerteksti mingi teise faili teksti. Meie ülesande jaoks on see lisatekst funktsiooni *summa* oma ning fail on *summa.txt*:

```
section .code
summa:
        push ebp
        mov  ebp,esp
        mov  eax,dword[ebp+8] ;x
        add  eax,dword[ebp+12] ;y
        pop  ebp
        ret
```

```
;-------------------------------------
```

Põhiprogramm demo.asm on järgmine:

```
;demo.asm :: alamprogrammi tekst on lisatud %include
abil
global _main
extern _printf

section .const
   pr db 'summa on %d',10,0

section .code
_main:
   push ebp
   mov  ebp,esp

   push 2  ;y
   push 3  ;x
   call summa
   add  esp,8
   push eax  ;summa on eax-s
   push pr
   call _printf
   add  esp,8

   pop  ebp
   ret
;-----------------------------------
%include 'summa.txt'
```

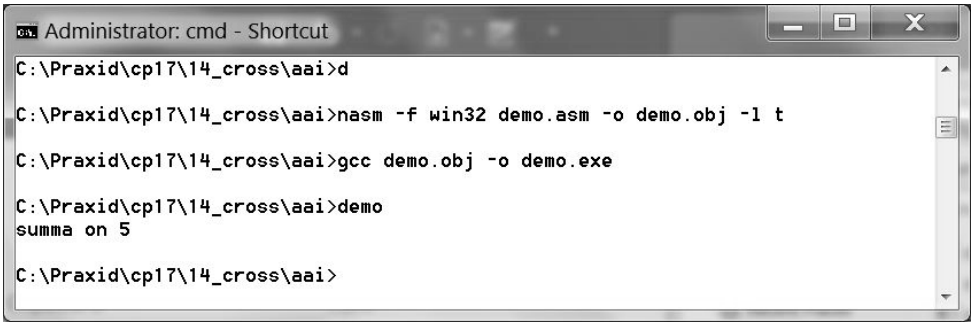
Tulemuse näitamiseks lasime NASMil genereerida vahekeelse objektfaili listingu. Käsurida:

```
Nasm -f win32 demo.asm -o demo.obj -l t
```

Tekstifail *t* on järgmine:

```
1          ;demo.asm :: alamprogrammi tekst on lisatud %include abil
2                                global _main
3                                extern _printf
4
```

```
5                                   section .const
6 00000000 73756D6D61206F6E20-         pr db ‘summa on %d’,10,0
7 00000009 25640A00
8
9                                   section .code
10                                   _main:
11 00000000 55                          push ebp
12 00000001 89E5                        mov  ebp,esp
13
14 00000003 6A02                        push 2  ;y
15 00000005 6A03                        push 3  ;x
16 00000007 E813000000                  call summa
17 0000000C 83C408                      add  esp,8
18 0000000F 50                          push eax  ;summa on eax-s
19 00000010 68[00000000]                push pr
20 00000015 E8(00000000)                call _printf
21 0000001A 83C408                      add  esp,8
22
23 0000001D 5D                          pop  ebp
24 0000001E C3                          ret
26                                   %include ‘summa.txt’
27                                <1> section .code
28                                <1> summa:
29 0000001F 55                    <1> push ebp
30 00000020 89E5                  <1> mov  ebp,esp
31 00000022 8B4508                <1> mov  eax,dword[ebp+8] ;x
32 00000025 03450C                <1>    add  eax,dword[ebp+12] ;y
33 00000028 5D                    <1>    pop  ebp
34 00000029 C3                    <1>    ret
35                                <1> ;-----------------------
```



Joonis 8.5.1. Lisatud alamprogrammi tekst.

8.5.2. ALAMPROGRAMM ON LISATUD MASINKOODIS

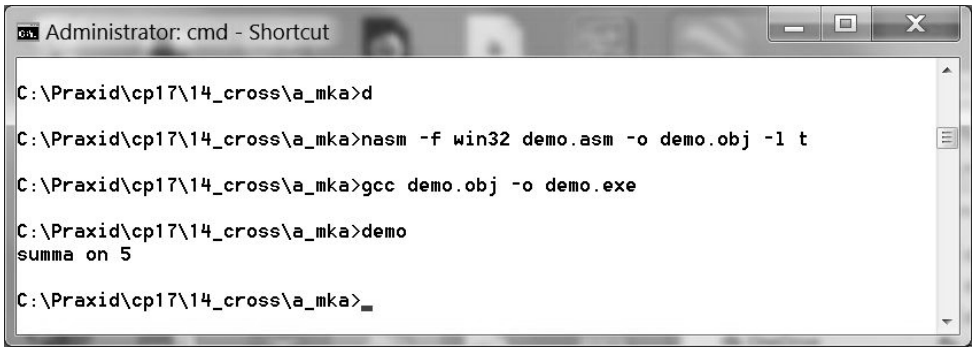
Näitame, kuidas saab *NASM*i abil lisada masinkoodi-teksti. Põhiprogrammi *demo.asm* me enam ei esita; seda saab vaadata objektprogrammi listingus. Niisiis, esmalt fail *pluss.txt*:

```
pluss:
        db 0x55
        db 0x89,0xE5
        db 0x8B,0x45,0x0C
        db 0x03,0x45,0x08
        db 0x5D
        db 0xC3
```

Kommentaariks: käsud on kirjutatud baithaaval – eraldaja on koma – ja kuueteistkümnendkoodis, mida näitab baidi prefiks *0x*. Objektprogrammi listing:

```
1                          ;demo.asm :: lihtasm. print(x+y)
2                                  global _main
3                                  extern _printf
4
5                                  section .const
6 00000000 73756D6D61206F6E20-     pr db ‘summa on %d’,10,0
7 00000009 25640A00
8
9                                  section .code
10                                  _main:
11 00000000 55                         push ebp
12 00000001 89E5                       mov  ebp,esp
13
```

```
14 00000003 6A02                    push 2  ;y
15 00000005 6A03                    push 3  ;x
16 00000007 E813000000              call pluss
17 0000000C 83C408                  add  esp,8
18 0000000F 50                      push eax  ;summa on eax-s
19 00000010 68[00000000]            push pr
20 00000015 E8(00000000)            call _printf
21 0000001A 83C408                  add  esp,8
22
23 0000001D 5D                      pop  ebp
24 0000001E C3                      ret
25                              ;------------------------
26                              %include 'pluss.txt'
27                          <1> pluss:
28 0000001F 55              <1>     db 0x55
29 00000020 89E5            <1>     db 0x89,0xE5
30 00000022 8B450C          <1>     db 0x8B,0x45,0x0C
31 00000025 034508          <1>     db 0x03,0x45,0x08
32 00000028 5D              <1>     db 0x5D
33 00000029 C3              <1>     db 0xC3
```



Joonis 8.5.2.a. Alamprogramm on kuueteistkümnendkoodis.

8.5.3. ALAMPROGRAMM ON LISATUD MASINKOODIS (KAHENDKOOD)

Raamatu alguses, kus tutvustati põgusalt masinkoodi, sai usutavasti selgeks, et 16-ndkoodi kirjutamine on komplitseeritud, kuivõrd käsud on bitikaupa kokku pakitud ning koodi kirjutamine võiks tehtav olla pigem kahendkoodi kasutades. Nii on kirjutatud fail *plussb.txt*:

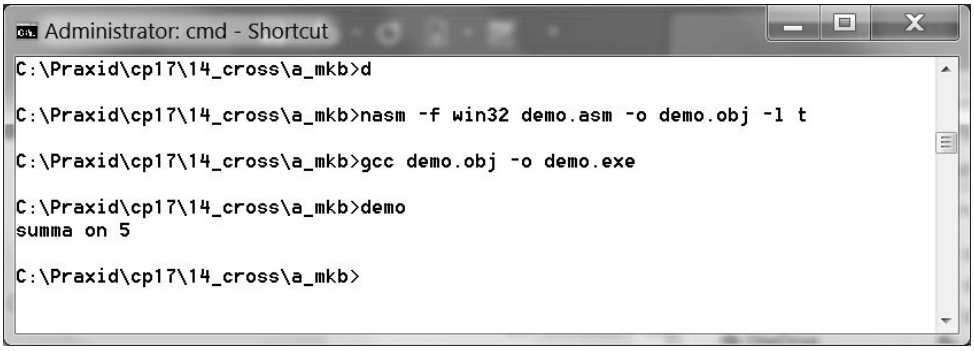
```
pluss:
db 0b01010101
db 0b10001001,0b11100101
db 0b10001011,0b01000101,0b00001100
db 0b00000011,0b01000101,0b00001000
db 0b01011101
db 0b11000011
```

Kood on *NASM*ile esitatud baithaaval, eraldaja on koma ning prefiks *0b* määrab kahendkoodi.

Objektprogrammi listing on failis *t*:

```
1                        ;demo.asm :: lihtasm. print(x+y)
2                                 global _main
3                                 extern _printf
4
5                                 section .const
6 00000000 73756D6D61206F6E20-       pr db ‘summa on %d’,10,0
7 00000009 25640A00
8
9                                 section .code
10                                 _main:
11 00000000 55                        push ebp
12 00000001 89E5                      mov  ebp,esp
13
14 00000003 6A02                      push 2  ;y
15 00000005 6A03                      push 3  ;x
16 00000007 E813000000                call pluss
17 0000000C 83C408                    add  esp,8
18 0000000F 50                        push eax  ;summa on eax-s
19 00000010 68[00000000]              push pr
20 00000015 E8(00000000)              call _printf
21 0000001A 83C408                    add  esp,8
22
23 0000001D 5D                        pop  ebp
24 0000001E C3                        ret
25                             ;-----------------------------------
26                                 %include ‘plussb.txt’
```

```
27                              <1> pluss:
28 0000001F 55                  <1> db 0b01010101
29 00000020 89E5                <1> db 0b10001001,0b11100101
30 00000022 8B450C    <1> db 0b10001011,0b01000101,0b00001100
31 00000025 034508     <1> db 0b00000011,0b01000101,0b00001000
32 00000028 5D                  <1> db 0b01011101
33 00000029 C3                  <1> db 0b11000011
34                              <1>
```



Joonis 8.5.3. Alamprogramm on kahendkoodis.

8.6. C PÕHI- JA ASM-ALAMPROGRAMM

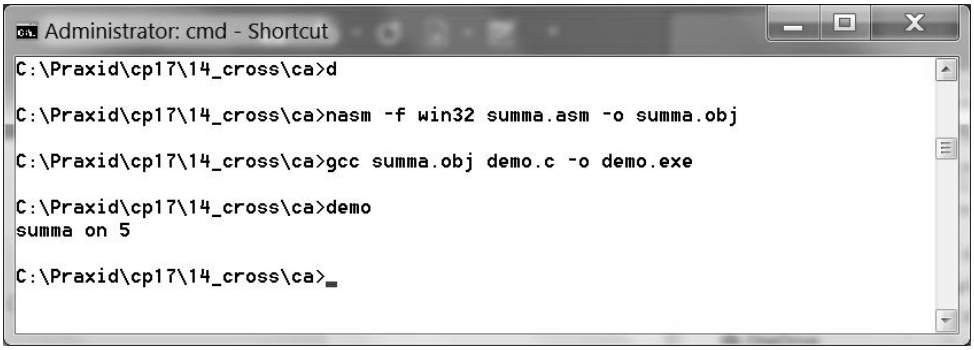
Põhiprogramm tuleb vormistada samuti nagu päisfailita *C+C* puhul ning alamprogramm samamoodi nagu assembler-põhiprogrammi puhul.

```
//demo.c :: ristkasutus, liht-C
#include<stdio.h>

int summa(int x,int y);

int main( ){
   int sum;
   sum=summa(2,3);
   printf("summa on %d\n",sum);
}
```

Assembler:

```
global _summa
```

```
section .code
_summa:
   push ebp
   mov  ebp,esp
   mov  eax,dword[ebp+8] ;x
   add  eax,dword[ebp+12] ;y
   pop  ebp
   ret
;-------------------------------------
```



Joonis 8.6.a. C põhi- ja asm-alamprogramm.

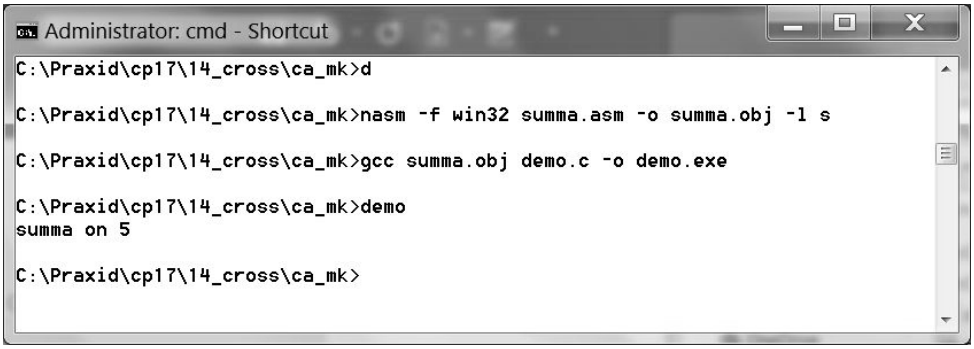
Selles kombinatsioonis võime normaalse *NASM*-alamprogrammi asendada masinkoodis kirjutatuga – faili nimi on nüüd *summa.asm*:

```
global _summa
section .code
_summa:
db 0b01010101
db 0b10001001,0b11100101
db 0b10001011,0b01000101,0b00001100
db 0b00000011,0b01000101,0b00001000
db 0b01011101
db 0b11000011
```

Toome ka *summa.asm*i objektprogrammi listingu:

```
1                                global _summa
2
3                                section .code
4                                _summa:
5 00000000 55                    db 0b01010101
```

```
6 00000001 89E5                    db 0b10001001,0b11100101
7 00000003 8B450C        db 0b10001011,0b01000101,0b00001100
8 00000006 034508        db 0b00000011,0b01000101,0b00001000
9 00000009 5D                      db 0b01011101
10 0000000A C3                     db 0b11000011
```



Joonis 8.6.b. C põhi- ja masinkoodi-alamprogramm.

8.7. ASM PÕHI- JA C-ALAMPROGRAMM

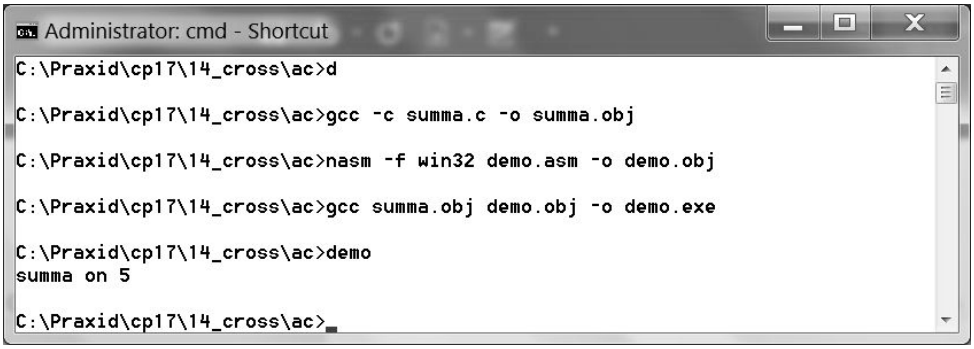
Assemblertekstis deklareeritakse C-keeles kirjutatud funktsioon välisnimena: *extern _summa*.

```
;demo.asm :: lihtasm. print(x+y)
global _main
extern _printf
extern _summa
section .const
   pr db 'summa on %d',10,0
section .code
_main:
   push ebp
   mov  ebp,esp
   push 2  ;y
   push 3  ;x
   call _summa
   add  esp,8
   push eax  ;summa on eax-s
   push pr
```

```
call _printf
add  esp,8
pop  ebp
ret
```

Funktsioon summa on programmeeritud C-keeles:

```
//summa.c :: demo.c eraldi transl. alamprog.
int summa(int x,int y){
   return(x+y);
}
```



Joonis 8.7. Põhiprogramm on assembleris ja alamprogramm C-s.

9 ÜHEMÕÕTMELINE MASSIIV (VEKTOR)

9.1. FAIL

Üldjuhul käsitletakse välismälu-faili kui lihtsat baidijada ning ta sisestatakse mällu just sellisena – ühemõõtmelise massiivi e. vektorina[1]. Sisestamiseks tuleb teha järgmised rutiinsed tööd:

- Avada fail lugemiseks.
- Küsida faili pikkus baitides.
- Küsida faili jaoks kuhjast mälu.
- Lugeda fail mällu.
- Üldjuhul – sulgeda fail.

Nende tööde jaoks on otstarbekas kirjeldada faili parameetrite väli ning kirjutada funktsioon, mis tagastab viida tollele väljale (või tühiviida, kui faili ei õnnestunud avada). Esitame selle funktsiooni teksti esmalt C-failina.

```
//fail.c :: Faili avamine ja lugemine. 12.02.19. Mina
Ise
//struct F *fail(char *nimi)
#include<stdio.h>
#include<stdlib.h>

struct F{
    char *nimi;
    int n;
    char *buf;
    FILE *mf;
```

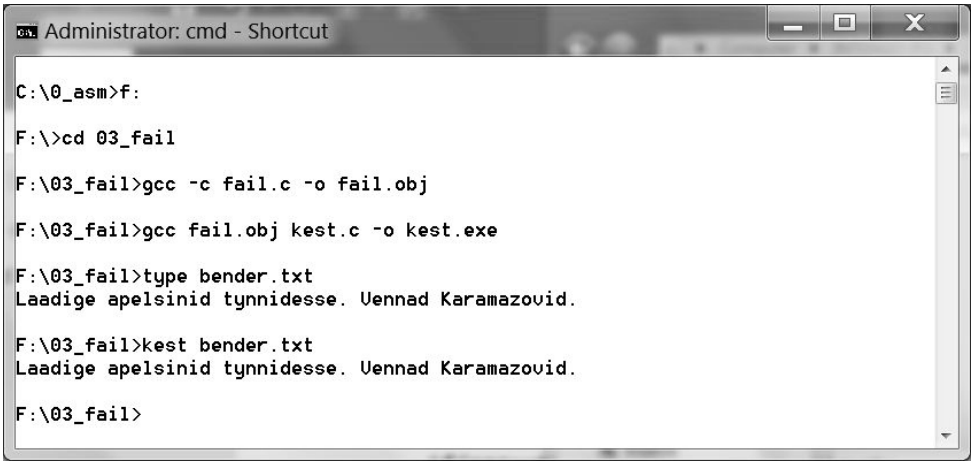
1 Struktureeritud faili sisestamise näite toome graafi käsitlevas peatükis.

```
};

struct F *fail(char *nimi){
    int i,n;
    FILE *mf=NULL;  //faili pide (handle)
    char *text;
    struct F *d; //deskriptor

    mf=fopen(nimi,”rb”);
    if(mf==NULL){
       printf(“faili %s pole\n”,nimi);
       return(NULL);
       }

      fseek(mf,0,SEEK_END);  //positsioneeri  viimasele
baidile
    n=ftell(mf);  //anna positsioon
      fseek(mf,0,SEEK_SET);  //positsioneeri  esimesele
baidile
     text=malloc(n+1);  //lisabait stringi l6putunnuse
jaoks
    fread(text,n,1,mf);
    text[n]=’\0’;  //stringi l6putunnus paika

    d=malloc(sizeof(struct F));
    d->nimi=nimi;
    d->n=n;
    d->buf=text;
    d->mf=mf;
    return(d);
}
```

Selle funktsiooni testimiseks on kirjutatud *kest.c*:

```
//kest.c :: fail(*nimi) tester. 14.02.19. Mina Ise
#include<stdio.h>
#include<stdlib.h>
struct F{
    char *nimi;
```

```
    int n;  //faili pikkus baitides kettal
    char *buf;  //faili aadress m2lus
    FILE *mf;
};
struct F *fail(char *nimi);
int main(int argc,char **argv){
    struct F *rec;
    rec=fail(argv[1]);
    if(rec==NULL)return 2;
    printf("%s",rec->buf);
    fclose(rec->mf);
}
```



Joonis 9.1.a. Funktsiooni *fail* silumine: C-prototüüp.

Kirjutame funktsiooni struct *F *fail(char *nimi)* nüüd assembleris[1]:

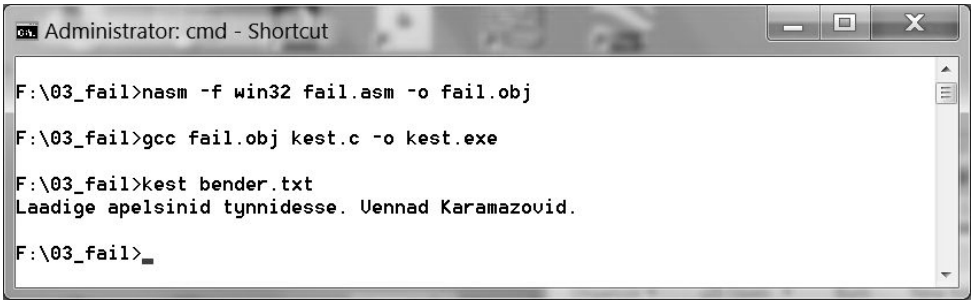
```
;fail.asm :: struct F *fail(char *filename). 16.04.19.
Mina Ise
global _fail
extern _fopen
extern _fclose
extern _fseek
extern _ftell
```

1 Uue asjana kasutatakse siin struktuurse välja kirjeldust *.bss*-sektsioonis. Erinevused C-keelest: andmetüübi nimi on *struc*, väljade nimed algavad punktiga ja kirjelduse lõpetab *endstruc*. Siin on struktuurse tüübi nimi *F* ning selle baitide arvu *n* saime C-s *n=sizeof(struct F)*, *NASM*is aga *F_size*. C-s adresseeritakse struktuuri *F* tüüpi muutuja *rec* alamvälju näit. *rec→n* või *rec→mf*, *NASM*is aga kui *baasaadress+F.n* ja *baasaadress+F.mf*. Nood nimed transleeritakse suhtaadressideks, käesoleval juhul vastavalt 4 ja 12.

```
extern _malloc
extern _fread
extern _printf
section .data
  viga db ‘faili %s pole teegis’,10,0
  mood db ‘rb’,0
section .bss
  struc F  ;struct F{
    .nimi resd 1  ;char *nimi;
    .n resd 1     ;int n;
    .buf resd 1   ;char *buf;
    .mf resd 1    ;FILE *mf;
endstruc         ;};
section .text
_fail:
   push ebp
   mov  ebp,esp
   push ebx
   push esi
   mov  eax,F_size  ;sizeof(struct F)
   push eax
   call _malloc
   add  esp,4
   mov  ebx,eax  ;ebx=parm-v2lja aadress
   mov  eax,dword[ebp+8] ;*filename
   mov  dword[ebx+F.nimi],eax ;parm->nimi=filename
   push mood
   push dword[ebp+8]
   call _fopen
   add  esp,8
   cmp  eax,0
   jne  oki
   push dword[ebp+8]
   push viga
   call _printf
   add  esp,8
   mov  eax,0   ;return(NULL)
```

```
    jmp  out
oki:
    mov  dword[ebx+F.mf],eax
    push 2  ;SEEK_END
    push 0
    push dword[ebx+F.mf]
    call _fseek
    add  esp,12
    push dword[ebx+F.mf]
    call _ftell
    add  esp,4
    mov  dword[ebx+F.n],eax
    push 0  ;SEEK_SET
    push 0
    push dword[ebx+F.mf]
    call _fseek  ;positsioon esimesele baidile
    add  esp,12
    mov  eax,dword[ebx+F.n]
    add  eax,1   ;lisabait stringi lõputunnusele
    push eax
    call _malloc
    add  esp,4
    mov  dword[ebx+F.buf],eax
    push dword[ebx+F.mf]
    push 1
    push dword[ebx+F.n]
    push dword[ebx+F.buf]
    call _fread
    add  esp,16
    mov  esi,dword[ebx+F.n]
    mov  eax,dword[ebx+F.buf]
    mov  byte[eax+esi],0   ;buf[n]='\0'
    push dword[ebx+F.mf]
    call _fclose
    add  esp,4
    mov  eax,ebx
out:
```

```
pop  esi
pop  ebx
pop  ebp
ret
```



Joonis 9.1.b. Funktsiooni *fail* silumine: *NASM*-prototüüp.

9.2. VERNAMI ŠIFFER


G.S. Vernam

USA firma AT&T insener Gilbert S. Vernam (1890-1960) leiutas 1917.a. *välistava* või *(xor)* loogikatehetele põhineva šifreerimismasina, mis krüpteeris ühe telegraafilindi ("dokumendi") teise lindi ("võtme") abil, saades kolmanda, krüpteeritud lindi. Dešifreerimiseks tuli dokumendi rollis kasutada šifreeritud linti koos võtmelindiga ning väljundlint oli identne lähtedokumendiga. Vernam ise ei suutnud tõestada oma koodi täielikku murdmiskindlust; seda tegi 1949. a. Claude Shannon [cryptowiki].

Murdmiskindlaks teeb selle meetodi tõik, et puudub võtme genereerimise algoritm; algoritmiliselt genereeritud võti on alati taasgenereeritav – iseasi, kui keeruliseks see võib osutuda.

See meetod on lihtsalt programmeeritav[1]: šifreerida saab suvalist faili, kasutades võtme rollis teist mistahes tüüpi vähemalt sama pikka faili. Jälgede segamiseks võib võtmefaili kasutada nihkega.

Tuletagem bitikaupa toimivat *xor*-tehet meelde:

1. Dokument (101) *xor* võti (011) → lukus (110)

1 Vt. näit. [Isotamm C], lk. 113 jj.

2. Lukus (110) *xor* võti (011) → dokument (101), aga ka:
3. Lukus (110) *xor* dokument (101) → 011 = võti
4. Dokument (101) *xor* dokument (101) → 000
5. Dokument (101) *xor* 000 → 101

Neist meeldetuletustest võime teha mõned praktilised järeldused:

- Šifreerimiseks (1) ja dešifreerimiseks (2) kasutatakse sama algoritmi (programmi) ja samu faile, eeldusel, et töö lõpus kirjutatakse dokumendifail krüpteerimisresultaadiga üle: esimesel lahendamisel pannakse dokument lukku, teisel tehakse lahti, kolmandal pannakse jälle kinni jne.
- Märkusest (3) näeme, et võtmefail on lihtsalt tuvastatav sel juhul, kui korraga on koodi murdja kätte saanud nii sifreeritud faili kui ka (õnnekombel kätte saadud) dokumendifaili. Sellest on aga vähe kasu, kui ühtegi faili ei kasutata võtme rollis teist korda.
- Šifreerimisprogrammiga saab dokumendi sisuliselt ja taastamatult kustutada (4), kui panna dokumendifail lukku iseenda abil. Kettal on ta näiliselt muutmata kujul, ent sisuks on 0-baidid.
- Kuivõrd *.exe*-faili alguspooles on suhteliselt suured nullidega täidetud alad, siis (5) näitab, et sedatüüpi faili šifreerimine reedab üsna suure osa võtmefailist, ja kui *.exe*-faili kasutada võtmena, siis reedab ta samamoodi osa dokumendi sisust. Seega tuleks – kui *.exe*- faili on vaja kasutada – ta eelnevalt kokku pakkida.

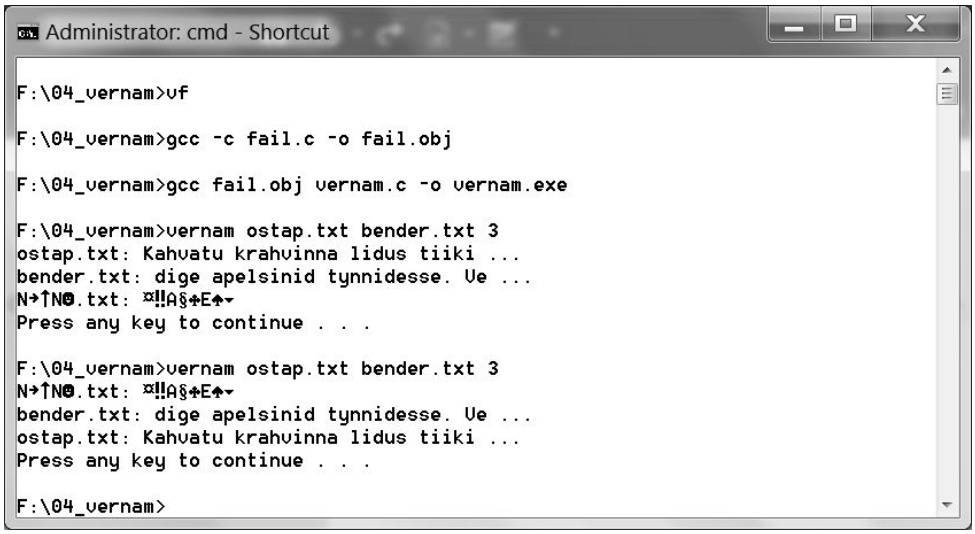
Allpool esitame Vernami meetodi põhiprogrammi esmalt C-keeles (alamprogrammina kasutatakse funktsiooni *fail*), seejärel aga esitame põhiprogrammi ka *NASM*is.

9.2.1. PROGRAMM VERNAM.C

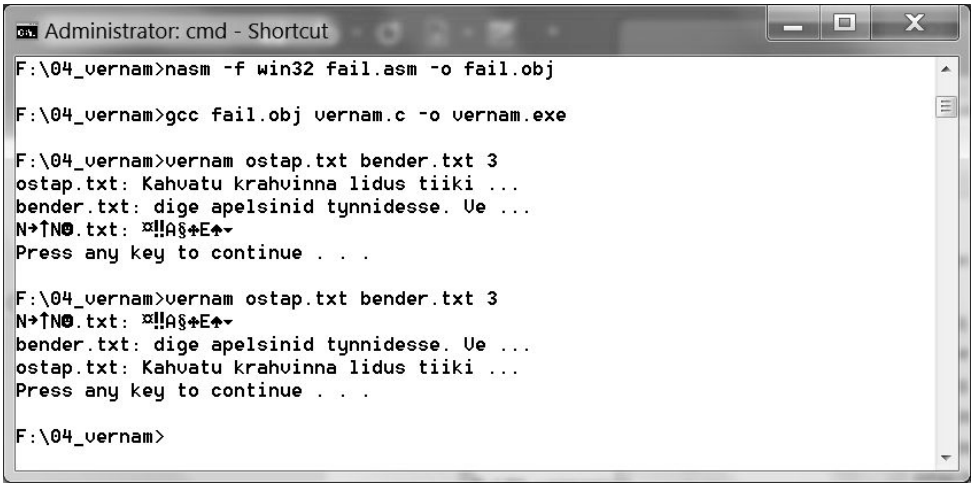
```
//vernam.c :: G.S.Vernam'i shiffer. 25.09.18. Mina Ise
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
```

```
struct F{
   char *nimi;
   int n;
   char *buf;
   FILE *mf;
};
struct F *fail(char *nimi);
//faili prefiksi trykk
int pp(char *fail,char *t,int n){
    char p;
    printf("%s: ",fail);
    if(strlen(t)<=n){
       printf("%s\n",t);
       return 0;
       }
    p=t[n-1];
    t[n-1]='\0';
    printf("%s ...\n",t);
    t[n-1]=p;
}
//>vernam bender.txt vernam.jpg [19]
int main(int argc,char **argv){
    struct F *dok;
    struct F *key;
    int i,nihe=0;
//k2surea formaalne kontroll
    if(argc>=3) goto korras;
    printf(">vernam dok key [nihe]\n");
    return 1;
korras:
    dok=fail(argv[1]);
    if(dok==NULL)return 1;
    pp(dok->nimi,dok->buf,30);
    key=fail(argv[2]);
    if(key==NULL)return 1;
    if(argc==4){
       nihe=atoi(argv[3]);
```

```
            key->buf+=nihe;
            key->n-=nihe;
            }
        pp(key->nimi,key->buf,30);
        if(dok->n > key->n){
        printf("v6ti %d on lyhem kui dok %d\n",key->n,dok->n);
            return 1;
            }
        for(i=0;i<dok->n;i++)dok->buf[i]^=key->buf[i];
        pp(dok->nimi,dok->buf,30);
        fclose(dok->mf);
        dok->mf=fopen(argv[1],"wb");
        fwrite(dok->buf,dok->n,1,dok->mf);
        fclose(dok->mf);
        fclose(key->mf);
        system("pause");
    }
```



Joonis 9.2.1.a. Vernami šiffer: C-lahendus.



Joonis 9.2.1.b. Vernami šiffer: C-põhi- ja NASM-alamprogramm.

9.2.2. FAILI PREFIKSI TRÜKK

Faili prefiksi trükk assembleris:

```
;pp.asm ::  prefiksi trykk void pp(char *nimi,char *t,int
n,int ;p) 25.04.19. Mina Ise
global _pp
extern _printf
section .data
   tf db ‘%s: %s’,’...’,10,0
section .text
_pp:
   push ebp
   mov  ebp,esp
   push ebx
   push esi
   push edi
   mov  eax,dword[ebp+16]  ;”teksti” pikkus
   cmp  eax,dword[ebp+20]  ;prefiksi pikkus
   jg   prefiks
   push dword[ebp+12]
   push dword[ebp+8]
   push tf
   call _printf
   add  esp,12
```

```
   jmp  aut
prefiks:
   mov  edi,dword[ebp+12]
   mov  esi,dword[ebp+20]
   sub  esi,1  ;index
   mov  bl,byte[edi+esi]
   mov  byte[edi+esi],0
   push edi
   push dword[ebp+8]
   push tf
   call _printf
   add  esp,12
   mov  byte[edi+esi],bl
aut:
   pop  edi
   pop  esi
   pop  ebx
   pop  ebp
   ret
```

9.2.3. PROGRAMM VERNAM.ASM

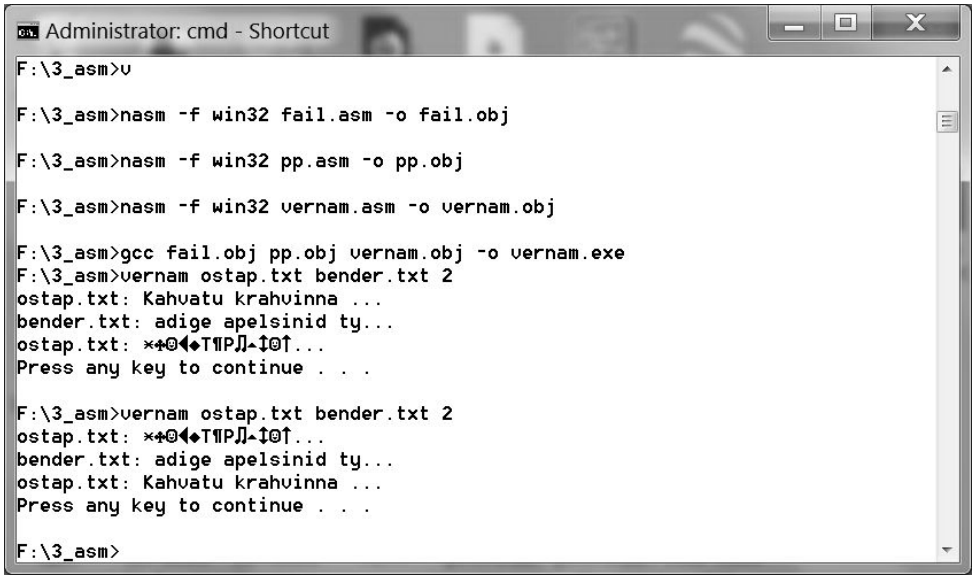
```
;vernam.asm :: Vernami shiffer. 25.04.19. Mina Ise
global _main
extern _fail
extern _pp
extern _printf
extern _fopen
extern _fclose
extern _fwrite
extern _atoi
extern _system
section .data
   viga db ‘v6ti on liiga lyhike’,10,0
   mood db ‘wb’,0
   kr   db ‘vale k2surida’,10,0
   paus db ‘pause’,10,0
section .bss
```

```
   struc F
     .nimi resd 1
     .n    resd 1  ;faili pikkus baitides kettal
     .buf  resd 1  ;faili aadress m2lus
     .mf   resd 1
   endstruc
section .text
_main:
   push ebp
   mov  ebp,esp
   push ebx
   push esi
   push edi
   mov  eax,dword[ebp+8] ;argc
   cmp  eax,3
   jge  oki
   push kr
   call _printf
   add  esp,4
   jmp  aut
oki:
   mov  ebx,dword[ebp+12]  ;*argv
;loen dokumendi
   push dword[ebx+4] ;argv[1]
   call _fail
   add  esp,4
   test eax,eax
   jz   aut
   mov  esi,eax  ;dokumendi parm
   push 19
   push dword[esi+F.n]
   push dword[esi+F.buf]
   push dword[esi+F.nimi]
   call _pp
   add  esp,16
;loen v6tme
   push dword[ebx+8] ;argv[2]
```

```
   call _fail
   add  esp,4
   test eax,eax
   jz   aut
   mov  edi,eax  ;dokumendi parm
;kas KReal on antud v6tme nihe?
   mov  eax,dword[ebp+8]  ;argc
   cmp  eax,4
   jne  kontroll
   push dword[ebx+12] ;nihe
   call _atoi
   add  esp,4
   add  dword[edi+F.buf],eax  ;buf=buf+nihe
   sub  dword[edi+F.n],eax ;n=n-nihe
;kas v6ti on sama pikk v6i pikem kui dokument?
kontroll:
   mov  eax,dword[esi+F.n] ;dok. pikkus
   cmp  dword[edi+F.n],eax
   jnl  kodeeri
   push viga
   call _printf
   add  esp,4
   jmp  aut
kodeeri:
   push 19
   push dword[edi+F.n]
   push dword[edi+F.buf]
   push dword[edi+F.nimi]
   call _pp
   add  esp,16
   push esi
   push edi
   mov  ecx,dword[esi+F.n] ;tsykliloendaja
   mov  esi,dword[esi+F.buf]
   mov  edi,dword[edi+F.buf]
   xor  eax,eax
   mov  edx,0  ;tsykli-indeks i
```

```
ring:
   mov  al,byte[esi+edx]
   xor  al,byte[edi+edx]
   mov  byte[esi+edx],al
   add  edx,1 ;i++
   loop ring
   pop  edi
   pop  esi
   push 19
   push dword[esi+F.n]
   push dword[esi+F.buf]
   push dword[esi+F.nimi]
   call _pp
   add  esp,16
   push mood
   push dword[ebx+4] ;argv[1]
   call _fopen
   add  esp,8
   mov  dword[esi+F.mf],eax
;fwrite(buf,n,1,mf)
   push eax ;*mf
   push 1
   push dword[esi+F.n]
   push dword[esi+F.buf]
   call _fwrite
   add  esp,16
   push dword[esi+F.mf] ;dok. kinni
   call _fclose
   add  esp,4
   push dword[edi+F.mf] ;v6ti kinni
   call _fclose
   add  esp,4
   push paus
   call _system
   add  esp,4
aut:
   pop  edi
```

```
        pop  esi
        pop  ebx
        pop  ebp
        ret
```



Joonis 9.2.3.a. Vernami šiffer: NASM põhi- ja alamprogramm.

9.3. *ASCII*

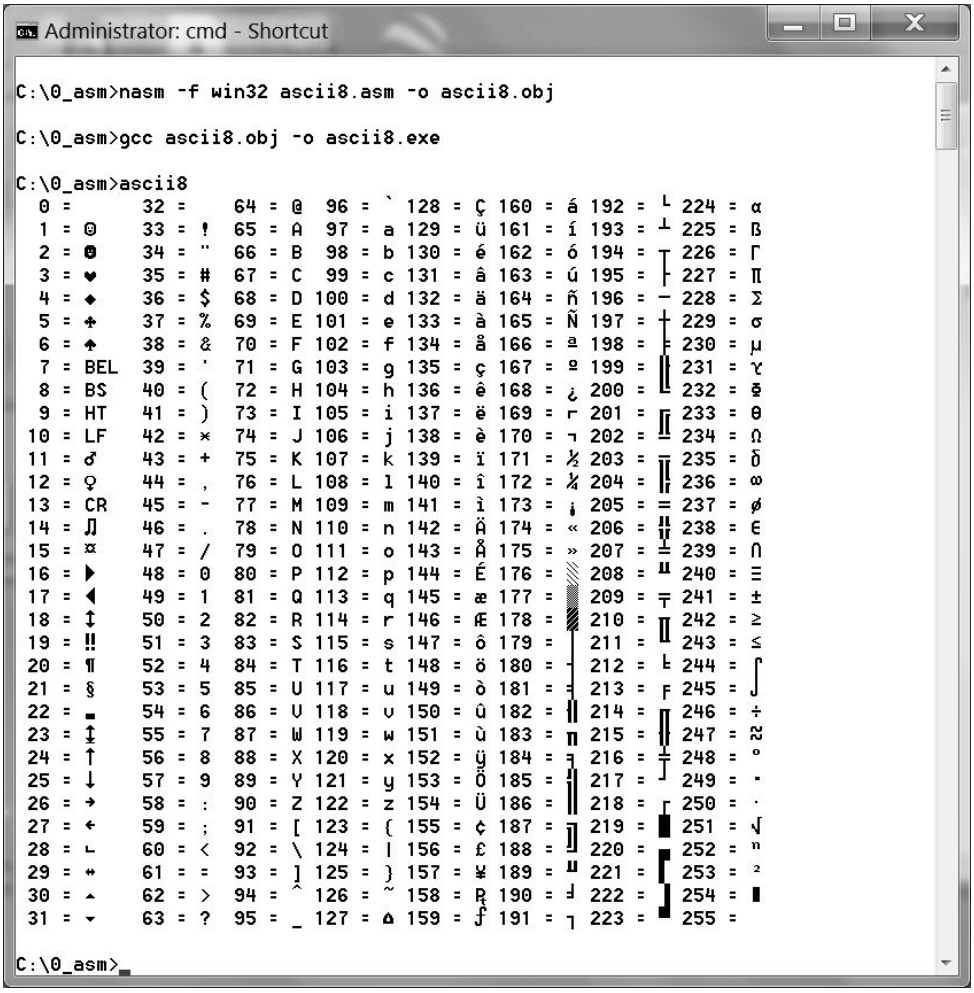
9.3.1. *ASCII*-TABEL

Selles alampeatükis jätkame ühemõõtmelise massiivi (vektori) käsitlemist, ent alustame pisut lihtsamast – programmist, mis kuvab 8-veerulise *ASCII*-koodi[1] tabeli. Iseenesest on tegu triviaalse programmiga; üheveerulise tabeli kuvab C-operaator

```
for(i=0;i<256;i++)printf(„%3d %c\n“,i,i);
```

aga mõnevõrra keerulisemaks teeb selle asja soov kuvada 32 rea ja 8 veeruga ühele ekraanile mahtuv „esteetiline“ tabel. Tabeli väljanägemist rikuksid koodid 7...10 ja 13 ning asja huvides tuleb sümbolite asemel trükkida nende toime, näit. 10 (reavahetus) asemel „*LF*“ või 13 (kursor rea algusse) asemel „*CR*“. Et näha, mida oli vaja programmeerida, esitame esimesena lahenduspildi ja seejärel *asm*-programmi.

1 *American Standard Code for Information Interchane* [CD, lk. 21].



Joonis 9.3.a. *ASCII*-tabel

Tabeli teeb ja trükib järgmine programm:

```
;ascii8.asm :: ASCII-tabeli trykk. 12.05.19. Mina Ise
global _main
extern _printf

section .data
   pf db '%3d = %c   %3d = %c %3d = %c %3d = %c %3d = %c
%3d = %c %3d = %c %3d = %c',10,0
  pfs db '%3d = %s %3d = %c %3d = %c %3d = %c %3d = %c
%3d = %c %3d = %c %3d = %c',10,0
   a7 db 'BEL',0
   a8 db 'BS ',0
```

```
    a9 db ‘HT ‘,0
   a10 db ‘LF ‘,0
   a13 db ‘CR ‘,0
   eri dd a7,a8,a9,a10  ;NB!

section .text
_main:
   push ebp  ;caller’s frame base
   mov  ebp,esp ;callee’s frame base
   push esi

   mov  esi,0  ;i=0
   mov  ecx,32
ring:
   push ecx ;printf-i eest peitu

   mov  eax,esi
   add  eax,224  ;8. veerg
   push eax
   push eax

   mov  eax,esi
   add  eax,192  ;7. veerg
   push eax
   push eax

   mov  eax,esi
   add  eax,160   ;6. veerg
   push eax
   push eax

   mov  eax,esi
   add  eax,128   ;5. veerg
   push eax
   push eax

   mov  eax,esi
```

```
   add  eax,96    ;4. veerg
   push eax
   push eax

   mov  eax,esi
   add  eax,64    ;3. veerg
   push eax
   push eax

   mov  eax,esi
   add  eax,32    ;2. veerg
   push eax
   push eax
;-------------------------------------------
   cmp  esi,13    ;1. veerg
   jne  muud
   push a13
   jmp  pes
muud:
   cmp  esi,7
   jl   norm
   cmp  esi,10
   jg   norm
   mov  eax,esi
   sub  eax,7
   shl  eax,2
   push dword[eri+eax]
pes:
   push esi
   push pfs
   jmp  tryki
norm:
   push esi
   push esi
   push pf
tryki:
   call _printf
```

```
        add  esp,68
        pop  ecx
        inc  esi
        loop ring

        pop  esi
        pop  ebp  ;restore caller's frame base
        ret       ;pop eip
```

Programmis tehtavast võimaldab loodetavasti aru saada trükiformaat („i“ pannakse magasini „paremalt vasakule“ 8-verulise sammuga), aga omaette kommentaari väärib rida

```
        eri dd a7,a8,a9,a10  ;NB!
```

Näeme, et neljabaidiste konstantide asemel on *dd* argumentideks etiketid – lootes, et translaator kirjutab nende asemele stringide aadressid – ning see toimib.

9.3.2. SÜMBOLITE SAGEDUSTABEL

ASCII-koodide suhtarv (esinemissageduste osatähtsus) failis võib pakkuda lihtsalt huvi – et kui palju on nulle ikkagi *.exe*-failis – aga on valdkondi, kus see on oluline. Näiteks maailmasõdadeaegne agentuurluure šifrogrammide dešifreerimise esimene etapp: mis keelt on kasutatud? Vastuluuretel olid kasutada erinevate keelte tähtede esinemissageduste tabelid ja piisavalt paljude kinnipüütud sõnumite lahtimuukimiseks oli sellest suur abi[1]. Mõned näited [letter] kasutamissageduse langevas suunas:

- Inglise keeles: e t a o i n s r h l d c u m f p g w y b v k x j q z
- Saksa keeles: e n i s r a t d h u l c g m o b w f k z v ü p ä ß j ö y q x
- Prantsuse keeles: e s a i t n r u l o d c m p é v q f b g h j à x è y ê z ç ô ù â û î œ w k ï ë ü æ ñ
- Vene keeles: о е а и н т с в л р к д м п у ё я г б з ч й х ж ш ю ц щ е ф (ъ ы ь)
- Soome keeles: e n a t r s i l d o k g m v f aa u p h ä c b ö j y x z w (q)

1 Sel juhul, kui oli kasutatud koodiraamatu-süsteemi (see oli valdav): Venelased kasutasid „käsitsi-Vernami“ süsteemi, seal keelestatistika ei aidanud. (vt. näit. [Isotamm C, lk. 103 jj.]).

Eesti keelt selle allika valikus pole, ja ka meie programm seda lünka ei täida, moodustame täieliku 256-baidiste koodide sagedustabeli (huvi korral on sealt lihtne välja sõeluda tähtede A/a...Ü/ü (või Z /z) sagedused).

Esmalt esitame koodide sageduste leidmise programmi C-s:

```
//freq.c ::  faili symbolite sagedustabel. 12.09.18.
A.I.
#include<stdio.h>
#include<stdlib.h>
int main(int argc,char **argv){
      int i,n1;
      char *dokument;
      FILE *mfd=NULL;
      int ST[256];
    if(argc!=2){
      printf("parameetrite arv ei klapi\n");
      return 1;
      }
//avame dokumendifaili
      mfd=fopen(argv[1],"rb");
      if(mfd==NULL){
            printf("ei saa faili %s lahti\n",argv[1]);
            return 1;
          }
      fseek(mfd,0,SEEK_END);
      n1=ftell(mfd);
      fseek(mfd,0,SEEK_SET);
      dokument=malloc(n1);
      fread(dokument,n1,1,mfd);
//teeme dokumendi symbolite sagedustabeli
    for(i=0;i<256;i++)ST[i]=0;
    for(i=0;i<n1;i++)ST[dokument[i]]++;
    for(i=0;i<256;i++){
       if(ST[i])printf("%3d %c %d\n",i,i,ST[i]);
       }
//fail kinni
      fclose(mfd);
```

```
}
```

Ja sama tööd tegev NASM-programm:

```
;freq.asm ::  faili koodide sagedused. 28.05.19. A.I.
global _main
extern _fail
extern _printf

section .data
   viga db ‘pole faili’,10,0
   pf   db ‘%c %d %d:%d’,10,0
   ty db ‘%d %s’,10,0
   sada dd 100
section .bss
struc F
   .nimi resd 1
   .n resd 1
   .buf resd 1
   .mf resd 1
endstruc
    stabel resb 256
section .text
_main:
   push ebp
   mov  ebp,esp
   push ebx
   push esi
   push edi
;k2surea kontroll : >freq <fail>
   mov  eax,dword[ebp+8] ;argc
   cmp  eax,2
   jnl  oki
   push viga  ; ‘pole faili’
   call _printf
   add  esp,4
   jmp  aut
oki:
```

```
;nullin sagedustabeli
   cld   ; dest-flag: vasakult paremale
   mov  ecx,256
   mov  eax,0
   mov  edi,stabel
   rep stosb

;-------------------------------------
   mov  ebx,dword[ebp+12]  ;**argv
   push dword[ebx+4]  ;argv[1]
   call _fail
   add  esp,4
   cmp  eax,0
   je   aut
   mov  ebx,eax  ;parameetrite kirje
;kontrolltrykk:
   push dword[ebx+F.buf]
   push dword[ebx+F.n]
   push ty
   call _printf
   add  esp,12
;-----------------------------------
;sagedusvektori t2itmine
   mov  ecx,dword[ebx+F.n]
   mov  edi,dword[ebx+F.buf]
   mov  edx,stabel
   mov  esi,0
   xor  eax,eax
ring:
   mov  al,byte[edi+esi]
   add  byte[edx+eax],1
   inc  esi
   loop ring
;esinevate symbolite sageduste trykk
   mov  ecx,256
   mov  esi,0
   mov  edi,stabel
```

```
ring2:
   xor  eax,eax
   mov  al,byte[edi+esi]
   cmp  eax,0
   je   next
   push ecx  ;peitu
;osatähtsus: (sagedus * 100)/n :: quot:rem
   mov  edx,0
   mov  ecx,100
   imul ecx
   cdq
   mov  ecx,dword[ebx+F.n]
   idiv ecx
   push edx  ;jääk
   push eax  ;täisosa
   xor  eax,eax ;sagedus
   mov  al,byte[edi+esi]
   push eax  ;ASCII kood
   push esi
   push pf
   call _printf
   add  esp,20
   pop  ecx  ;tsykliloendaja taastamine
next:
   inc  esi
   loop ring2
aut:
   pop  edi
   pop  esi
   pop  ebx
   pop  ebp
   ret
```

```
Administrator: cmd - Shortcut

C:\freq>freq krahv.txt
30 Kahvatu krahvinna lidus tiiki.
  3 10:0
. 1 3:10
K 1 3:10
a 4 13:10
d 1 3:10
h 2 6:20
i 5 16:20
k 2 6:20
l 1 3:10
n 2 6:20
r 1 3:10
s 1 3:10
t 2 6:20
u 2 6:20
v 2 6:20

C:\freq>
```

Joonis 9.3.2.a. Sümbolite sagedustabel

9.4. Vektoridirektiivid

*NASM*i – ja võimalik, et ka teiste x86 jaoks tehtud assemblerite direktiivide hulgas – on mitmeid, mida võiksime nimetada „poolmakrodeks“ – need on direktiivid, mis võtavad kokku mitu tavadirektiivi ning võimaldavad ökonoomsemalt programmeerida. Seejuures on nad „läbipaistvalt“ programmeeritavad käskhaaval. Laename siin Paul Carteri [Carter] teksti – oma raamatus annab ta selleks lahkelt loa.

Kõigi vektoridirektiivide kontrollitavaks toimimiseks tuleb alati „heisata“ *flags*-registris suunalipp: *cld* (*clear destination flag*): vektori elemendi indeks *i=0, i++* või *std* (*set direction flag*): *i=n-1, i--*. Seega: kas vektorit töödeldakse vasakult paremale või vastupdi. Ja tähelepanu! Sellel lipul pole vaikimisiolekut – grupitööde eel tuleb ta alati ise paika panna.

Tehetes osaleate vetorite aadressid tuleb eenevalt laadida registritesse *esi* (*source*, lähtekoht) ja *edi* (*destination*, sihtkoht) ning vektori(te ühine) pikkus registrisse *ecx*. Viimast kasutab tsüklikäsk *loop*. Indekseid ei kasutata, tsükli iga sammu lõpus nihutatakse vektorite aadresse olenevalt andmete tüübist kas

ühe, kahe või nelja baidi[1] võrra (olenevalt suunalipust kas vasakule või paremale). Töömäluna kasutavad nad *eax*-registrit (või osa sellest, *ax* või *al*).

Näidetes kasutame Carteri keskkonda:

```
Segment .data
   array1 dd 1,2,3,4,5
   tekst1 db 'abcd',0
segment .bss
   array2 resd 5
   tekst2 resb 4
   vektor resd 100

section .text
   ...
;kopeerimine
      Cld
      mov  esi,array1
      mov  edi,array2
      mov  ecx,5
   lp:
      lodsd
      stosd
      loop lp
;Kopeerimise teine variant
      cld
      mov  esi,tekst1
      mov  edi,tekst2
      mov  ecx,4
      rep movsb[2]
;"memset": vektori täitmine etteantud sümboliga
      cld
      mov  edi,vektor
      xor  eax,eax  ;täiteväärtus
      mov  ecx,100
      rep stosd
```

1 Nende grupikäskude sufiks on alati s (single, üksik) ja b (byte, bait), w (word, 2 baiti) või d (double word, 4 baiti).

2 käsu prefiks rep toimib nagu tsüklikäsk, loendaja on registris ecx.

```
;elemendi otsimine vektorist
      mov  edi,tekst1
      mov  al,'c'
      mov  ecx,4
otsi:
      scasb
      je   leidsin
      jmp  aut  ;pole otsitavat
leidsin:
         sub  edi,1  ;viit on järgmisel sümbolil
         ...
;vektorite võrdlemine
         cld
         mov  esi,tekst1
         mov  edi,tekst2
         mov  ecx,4
         repe[1] cmpsb
         je   vordsed
;kood: stringid erinevad
vordsed:
;tekst1=tekst2
```

Paul Carter [Carter, lk. 111jj.] esitab assemblerkoodi mitme C-keele stringifunktsiooni jaoks ülaltoodud „poolmakrode“ abil: *strcpy, strchr, strlen* ja *memcpy*.

Siinkirjutaja soovitus: kuni assembleris programmeerimine pole nende ridade lugeja jaoks jõudnud veel rutiiniks muutuda, tuleks „poolmakrode“ – nii selle alapeatüki omade, lisaks *enter*i ja *leave*'i – kasutamist vältida, et säiliks täielik ülevaade ja arusaamine oma programmist.

1 repe, repz: korratakse kuni nulli-lipp ZF pole heisatud -- aga mitte rohkem, kui ecx lubab.

10 FIBONACCI: JADA JA ROOMA NUMBRID

10.1. FIBONACCI

Fibonacci (1170–1250)

„Euroopa keskaja kestuseks loetakse tavaliselt aastaid 476 (Lääne-Rooma riigi langus) kuni 1500 (renessansi, humanismi ja tsentraalvõimuga riigikorra laiema leviku algus). Keskaja lõpuks peetakse aastat 1492, mil Kolumbus jõudis Ameerikasse.“ Ning kõrgkeskaeg kestis 11. sajandist 13. sajandi lõpuni [keskaeg]. Selle kõrgkeskaja üks mõjukaimaid teadlasi oli mees, keda me tunneme lihtsalt nimega Fibonacci ning kõik reaalainete tudengid teavad temanimelist arvude jada (mida ta lihtsalt tutvustas, talle näitasid seda araablased, kelleni see oli jõudnud Indiast, kus toda jada teati juba paarsada aastat enne Kristust) järgi. Ent sootuks olulisem Fibonacci teene oli india-araabia positsioonilise arvusüsteemi toomine Euroopasse.

Leonardo Bonaccio oli Guglielmo Bonaccio – mõjuka Pisa ärimehe (kes esindas tänapäeva Alžeerias oma linnriigi huvisd) poeg, kelle isa varakult endaga kaasa võttis ja kes hakkas elavalt suhtlema araabia õpetlastega.

Kaasajal oli Fibonacci tuntud mitme nimega, lisaks meieni kandunule ka kui Leonardo da Pisa või Leonardo Pisano või Leonardo Bigallo (reisimees), aga eeskätt ikkagi kui oma tuntud isa mõnesugust tähelepanu pälvinud poeg – *filius Bonacci* – lühemalt, Fibonacci [Fibonacci].

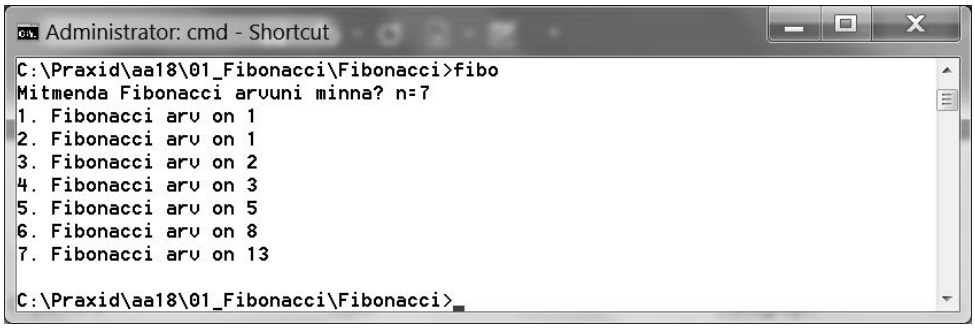
10.2. FIBONACCI JADA

Legendi järgi näitavad selle jada liikmed, kui palju järglasi annab ideaalis üks küülikupaar põlvkondade kaupa.
Algseisus on neid kaks, ükshaaval:
1 1
Nad hakkavad teineteisele meeldima, neist saab paar, neid on juba kaks:
1 1 2 (1+1)
Siis saavad nad poja, neid saab kolm:
1 1 2 3 (2 vana +1 poeg)
Edasi saab neid viis (3 olemasolevat +2 juurde)
Ja nii edasi.
Selle jada programmeerimiseks on mugav alustada nullist (ehkki küülikutenäite puhul tuleb pisut fantaseerida – mida see 0 ikkagi tähendada võiks):
0 1 1 2 3 5 jne.

Mingitpidi on Fibonacci jada arvutamine töö vektoriga: meil on reserveeritud mälu näit. 40 esimese Fibonacci arvu salvestamiseks ja selle jada iga uue liikme salvestamiseks võime suurendada indeksit või nihutada vektoriviita.
Allpool esitame paar programmivarianti. Lihtvariant C-keeles:

```
//Fibo.c :: trykib n esimest Fibonacci arvu. 13.01.17.
A.I.
#include<stdio.h>

int main(){
    int i,n;
    int prev=0;
    int curr=1;
    int next;
    char arv[10];
    printf("Mitmenda Fibonacci arvuni minna? n=");
    gets(arv);
    n=atoi(arv);
```

```
        printf("%d. Fibonacci arv on %d\n",1,1);
        for(i=1;i<n;i++){
           next=prev+curr;
           prev=curr;
           curr=next;
           printf("%d. Fibonacci arv on %d\n",i+1,next);
           }
    }
```



Joonis 10.2.a. C-programmi jooksutamine.

Portaal *Code Project* publitseeris kaks minimalistlikku assembler-programmi [Zuoliu Ding], meie kirjutasime neile pisut koodi ümber, et saada terviklikud alamprogrammid, tinglike nimedega *india.asm* ja *hiina.asm* ning C-keelse põhiprogrammi *fibonacci.c*. Toome allpool nende programmide tekstid ära.

```
;india.asm :: trykib n-nda Fibonacci arvu. 13.01.17
;p6hiprogramm on Fibonacci.c, seal void india(int n);
global _india
extern _printf
;-------------------------------------
section .data
      form db '%d. Fibonacci arv on %d',10,0
;--------------------------------
section .text
_india:
      push ebp
      mov ebp,esp
      push ebx
      mov ecx,dword[ebp+8]
```

```
      test ecx,ecx
      jnz fibo
      xor eax,eax
      jmp jokk
;-----------------------------------
;Fibonacci                (https://www.codeproject.com/
Articles/1116188/
;Basic-Practices-in-Assembly-Language-Programming)
fibo:
      xor eax,eax
      mov ebx,1
L:
      xchg eax,ebx  ;vahetab registrite sisud eax ↔ ebx
      add eax,ebx
      loop L
jokk:
      push eax   ;resultaat
      push dword[ebp+8]   ;n
      push form ;'%d. Fibonacci arv on %d',10,0
      call _printf
      add esp,12
      pop ebx
      pop ebp
      ret
;-------------------------------------
;hiina.asm :: trykib n-nda Fibonacci arvu. 13.01.17
;p6hiprogramm on Fibonacci.c, seal void hiina(int n);
global _hiina
extern _printf
section .data
      form db '%d. Fibonacci arv on %d',10,0
section .text
_hiina:
      push ebp
      mov ebp,esp

      push ebx
```

```
      mov ecx,dword[ebp+8]  ;n
      test ecx,ecx
      jnz fibo
      xor eax,eax
      jmp jokk
;Fibonacci                (https://www.codeproject.com/
Articles/1116188/
;Basic-Practices-in-Assembly-Language-Programming)
fibo:
        xor   eax,eax
        mov   ebx,1
L1:
        xadd eax,ebx   ; first exchange and then add
        loop   L1
jokk:
      push eax   ;resultaat
      push dword[ebp+8]   ;n
      push form ;'%d. Fibonacci arv on %d',10,0
      call _printf
      add esp,12
      pop ebx
      pop ebp
      ret
;-------------------------------------------
//Fibonacci.c  ::  trykib  n  esimest  Fibonacci  arvu.
13.01.17
#include<stdio.h>
#include<stdlib.h>

void india(int n);
void hiina(int n);
```

```
Administrator: cmd - Shortcut

D:\public_html18\aa17\01_Fibonacci>fibonacci
Mitmenda Fibonacci arvuni minna? n=10
india:
0. Fibonacci arv on 0
1. Fibonacci arv on 1
2. Fibonacci arv on 1
3. Fibonacci arv on 2
4. Fibonacci arv on 3
5. Fibonacci arv on 5
6. Fibonacci arv on 8
7. Fibonacci arv on 13
8. Fibonacci arv on 21
9. Fibonacci arv on 34
10. Fibonacci arv on 55

hiina:
0. Fibonacci arv on 0
1. Fibonacci arv on 1
2. Fibonacci arv on 1
3. Fibonacci arv on 2
4. Fibonacci arv on 3
5. Fibonacci arv on 5
6. Fibonacci arv on 8
7. Fibonacci arv on 13
8. Fibonacci arv on 21
9. Fibonacci arv on 34
10. Fibonacci arv on 55

D:\public_html18\aa17\01_Fibonacci>
```

Joonis 10.2.b. Assemblerprogrammide test.

```
int main( ){
    int i,n;
    char arv[10];
    printf("Mitmenda Fibonacci arvuni minna? n=");
    gets(arv);
    n=atoi(arv);
    printf("india:\n");
    for(i=0;i<n+1;i++) india(i);
    printf("\nhiina:\n");
    for(i=0;i<n+1;i++) hiina(i);
}
```

10.3. ARAABIA → ROOMA

Siin kasutame etteantud arvu ühest arvusüsteemist teise teisendamises **kahemõõtmelist massiivi** ja üksiti vaatame, kuidas programmeerida lõpliku olekute hulgaga automaati.

	×1	×10	×100	×1000
1	I	X	C	M
2	II	XX	CC	MM
3	III	XXX	CCC	MMM
4	IV	XL	CD	MMMM
5	V	L	D	MMMMM
6	VI	LX	DC	MMMMMM
7	VII	LXX	DCC	MMMMMMM
8	VIII	LXXX	DCCC	MMMMMMMM
9	IX	XC	CM	MMMMMMMMM

Joonis 10.3.a. Teisendustabel positsioonilistest kümnendarvudest rooma süsteemi.

Tabelit on lihtne kasutada. Näiteks, teisendame arvu 1432.

Tuhandeliste (veerg 4) positsioonis on 1 (rida 1): kirjutame M

Sajaliste (veerg 3) positsioonis on 4 (rida 4): lisame CD

Kümneliste (veerg 2) positsioonis on 3 (rida 3): lisame XXX

Üheliste (veerg 1) positsioonis on 2 (rida 2): lisame II

Tulemus on MCDXXXII. Allpool esitame esmalt C-programmi, mis teisendab kuni neljakohalisi arve rooma kujule teisendustabeli abil – see on „sisse programmeeritud“ kahemõõtmelise massiivina, mille elemendid on stringid. Indekseerimise hõlbustamiseks on selles tabelis ka 0-rida ja 0-veerg.

```
//torome.c teisendab kuni 4-kohase kümnendarvu “rooma
kujule”. 8. //okt. 2012
#include <stdio.h>
#include <string.h>
#include <ctype.h>
```

```
char a[5];  /* araabia */
int n,m,olek,i;
char d;
//0-rida ja 0-veerg on “tabelis” loomuliku indekseeri-
mise jaoks
char *roma[5][10]={
    {},
        {“”,”I”,”II”,”III”,”IV”,”V”,”VI”,”VII”,”VIII”,”
IX”},
   {“”,”X”,”XX”,”XXX”,”XL”,”L”,”LX”,”LXX”,”LXXX”,”XC”},
   {“”,”C”,”CC”,”CCC”,”CD”,”D”,”DC”,”DCC”,”DCCC”,”CM”},
         {“”,”M”,”MM”,”MMM”,”MMMM”,”MMMMM”,”MMMMMM”,”M-
MMMMMM”,”MMMMMMMM”,”MMMMMMMMM”}
};
int main(){
      printf(“seansi l6petab Ctrl+c\n”);
R:    printf(“\n kuni neljakohaline kymnendarv: “);
      scanf(“%4s”,a);
      fflush(stdin);
      n=strlen(a);
      printf(“ = “);
      i=0;
      for(olek=n; olek>0; olek--){
        d=a[i];
            if(d==’0’) goto next;
          if(isdigit(d)==0) goto viga;
          m=d-’0’;
          printf(“%s”,roma[olek][m]);
            next: i++;
            }
      goto R;
      viga: printf(“\n%c pole number\n”,d);
      goto R;
}
```

Sama algoritm *NASM*is realiseerituna annab mõnevõrra pikema programmi. Tähelepanu võiks pöörata kahemõõtmelise massiivi sisseprogrammeeri-

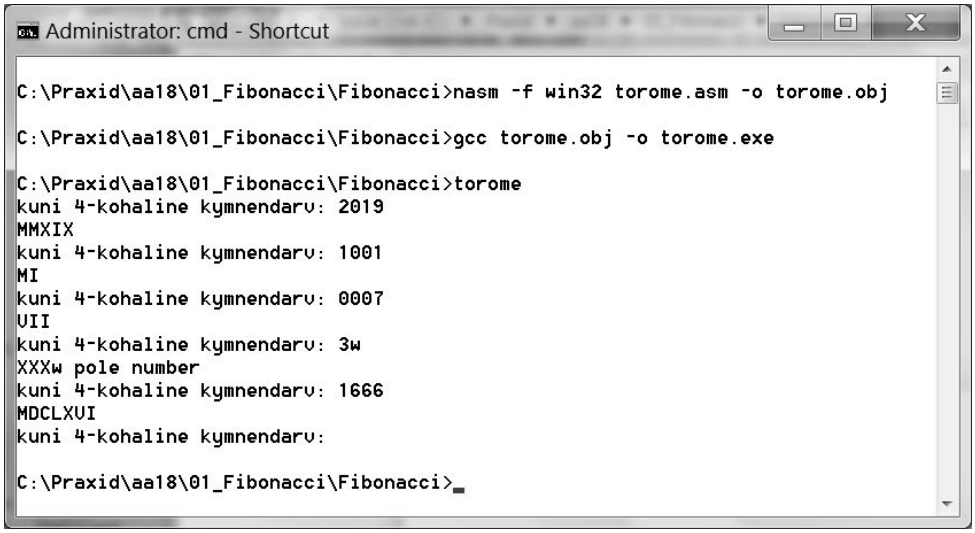
misele, sj. eriti sellele, et *.data*-sektsioonis on legaalne kirjutada *dd* argumendiks etikett, mille väärtuseks omistab translaator vastava objekti aadressi.

```
;torome.asm :: kuni 4-kohaline arv => rooma kujule.
23.05.19. A.I.
global _main
extern _printf
extern _gets
extern _isdigit
extern _strlen
section .data
   anna db ‘kuni 4-kohaline kymnendarv: ‘,0
   rome db ‘%s’,0
   reva db 10,0
   pole db ‘%c pole number’,10,0
   null db ‘’,0

   i    db ‘I’,0
   ii   db ‘II’,0
   iii  db ‘III’,0
   iv   db ‘IV’,0
   v    db ‘V’,0
   vi   db ‘VI’,0
   vii  db ‘VII’,0
   viii db ‘VIII’,0
   ix   db ‘IX’,0

   x    db ‘X’,0
   xx   db ‘XX’,0
   xxx  db ‘XXX’,0
   xl   db ‘XL’,0
   l    db ‘L’,0
   lx   db ‘LX’,0
   lxx  db ‘LXX’,0
   lxxx db ‘LXXX’,0
   xc   db ‘XC’,0

   c    db ‘C’,0
```

```
   cc   db 'CC',0
   ccc  db 'CCC',0
   cd   db 'CD',0
   d    db 'D',0
   dc   db 'DC',0
   dcc  db 'DCC',0
   dccc db 'DCCC',0
   cm   db 'CM',0

   m db 'M',0
   m2 db 'MM',0
   m3 db 'MMM',0
   m4 db 'MMMM',0
   m5 db 'MMMMM',0
   m6 db 'MMMMMM',0
   m7 db 'MMMMMMM',0
   m8 db 'MMMMMMMM',0
   m9 db 'MMMMMMMMM',0

   yhed dd null,i,ii,iii,iv,v,vi,vii,viii,ix
   kymned dd null,x,xx,xxx,xl,l,lx,lxx,lxxx,xc
   sajad dd null,c,cc,ccc,cd,d,dc,dcc,dccc,cm
   tuhanded dd null,m,m2,m3,m4,m5,m6,m7,m8,m9
   rooma dd null,yhed,kymned,sajad,tuhanded

section .bss
   arv resb 10

section .text
_main:
   push ebp
   mov  ebp,esp
   push ebx
   push esi
   push edi
ring:
   push anna
```

```
   call _printf
   add  esp,4
   push arv
   call _gets
   add  esp,4
   push arv
   call _strlen
   add  esp,4
   cmp  eax,0
   je   aut
   cmp  eax,4
   jng  neli
   mov  eax,4  ;olek=tuhanded
neli:
   mov  esi,eax  ;reaindeks
   mov  ebx,arv
   mov  edi,0    ;arv[0]
teen:
   xor  eax,eax
   mov  al,byte[ebx+edi]
   push eax
   call _isdigit
   add  esp,4
   cmp  eax,0
   je   polenr
   xor  eax,eax
   mov  al,byte[ebx+edi]
   sub  al,'0'  ;number=>int
   cmp  al,0
   je   nextolek
   mov  ecx,dword[rooma+esi*4]
   push dword[ecx+eax*4]
   push rome
   call _printf
   add  esp,8
nextolek:
   sub  esi,1
```

```
        cmp  esi,0
        je   tehtud
        add  edi,1
        jmp teen
    tehtud:
        push reva
        call _printf
        add  esp,4
        jmp  ring
    polenr:
        mov  eax,0
        mov  al,byte[ebx+edi]
        push eax
        push pole
        call _printf
        add  esp,8
        jmp  ring
    aut:
        pop  edi
        pop  esi
        pop  ebx
        pop  ebp
        ret
```



Joonis 10.3.b. Arvude teisendamine: assembler.

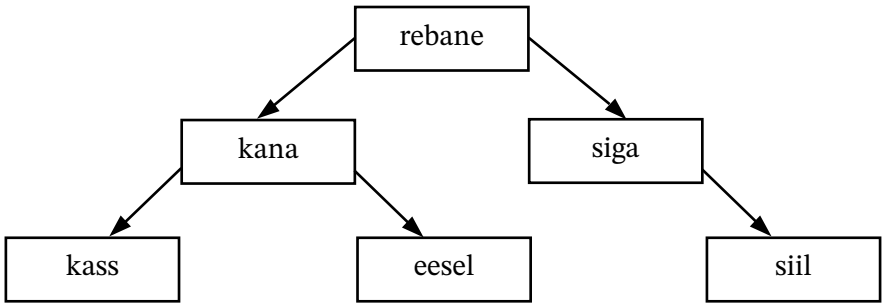
11 OTSIMIS- JA JÄRJESTAMISKAHENDPUU

11.1. KAHENDPUU

Selles peatükis tuleks lugejal tähelepanu pöörata viidastruktuuridele ja rekursiivsetele algoritmidele. Näiteprogramm ehitab lihtsa kahendpuu, igas tipus on kolm välja: tekstiline võti ja kaks viita alampuudele. Puufaili nimi loetakse käsurealt; kui seda faili pole, siis dialoogi käigus ta ehitatakse, kui on, siis loetakse mällu ja modifitseeritakse. Seansi lõpus teisendatakse mälus olev puu tasakaalustatud (*AVL*-) puuks ning kirjutatakse kettale.

Tuletame meelde mõningaid kahendpuuga seonduvaid seiku:

- Puu juur on tipp, millel pole ülemustippu ja millest alates on puu kõik ülejäänud tipud kättesaadavad.



Joonis 11.1.a. Otsimis- ja järjestamiskahendpuu.

- Otsimis- ja järjestamiskahendpuu suvalise alampuu juure märgendi väärtus on suurem ta vasaku alampuu juure märgendist ja väiksem kui parema alampuu juure märgendi väärtus. Märgendite (võtmete) väärtused peava olema unikaalsed (so, üksteisest erinevad)[1].

1 Otsimis- ja järjestamiskahendpuu on üks võimalikest abstraktse andmestruktuuri tabel esitusviisidest.

D. Knuthi [Knuth III, lk. 519-521] andmetel kuulub optimaalsete otsimiskahendpuude idee toonasele IBMi insenerile Hans Peter Luhnile; tema oli ka mees, kes mõtles välja *välisaheldusega paisksalvestuse* meetodi (1953). Knuth nendib, et tõenäoliselt oli see ka esimene kord, kui kasutati *ahelaid*.

Hans Peter Luhn (1896, Saksamaa– 1964, USA)

Kahendpuu läbimise viisid on seotud binaarse tehte (näiteks '+') erinevate esitusviisidega:

- *Prefiks*-kuju: +ab ning sellega seondatav kahendpuu läbimise moodus preorder (eesjärje-kord) rekursiivse valemiga *juur* → *vasak* → *parem* (või *juur* → *parem* → *vasak*). Olekus *juur* „tehakse tööd": kirjutatakse tipp kettale või loetakse sealt või trükitakse tipu-info. Kirjutame joonisel 11.a kujutatud puu tipumärgendid eesjärjekorras välja:
  rebane kana eesel kass siga siil
- Infiks-kuju: a+b ning sellega seondatav kahendpuu läbimise moodus inorder (keskjärjekord) rekursiivse valemiga vasak→ juur → parem (või parem → juur → vasak). Kirjutame joonisel 11.a kujutatud puu tipumärgendid keskjärjekorra esimest varianti kasutades välja:
  eesel kana kass rebane siga siil või kasutades eeskirja *parem* → *juur* → *vasak*: **siil siga rebane kass kana eesel**
  Keskjärjekorra peamine rakendus ongi tippude võtmeväärtuste kasvavas või kahanevas järjekorras järjendite genereerimine.
- *Postfiks*-kuju: ab+ ning sellega seondatav kahendpuu läbimise moodus *postorder* (lõppjärjekord) rekursiivse valemiga vasak→ parem → juur (või parem → vasak → juur). Kirjutame joonisel 11.a kujutatud puu tipumärgendid lõppjärjekorras välja: **eesel kass kana siil siga rebane**

Otsimis- ja järjestamiskahendpuu puhul pole lõppjärjekorral praktilist

Tabel on kirjete hulk, kirje koosneb loogilisel tasemel võtmest ja sellega seotud infost. Võtmed peavad olema unikaalsete väärtustega; juurdepääs kirjetele (otsimine) toimub võtmete abil. Õpikutes on tavaline, et „info" jäetakse ära – nagu meiegi teeme.

väärtust, ent avaldiste kahendpuude korral on see variant asendamatu. Selliste puude märgendid on mitterippuvates tippudes operatsioonid (tehtemärgid või funktsioonide nimed) ning rippuvates tippudes – operandid. Avaldise puu lõppjärjekorras läbimise käigus saab välja kirjutada märgendite jada, mida nimetatakse avaldise *inverteeritud Poola kujuks* (vt. näit [Isotamm PKd], lk.226).

11.2. OTSIMIS- JA JÄRJESTAMIS-KAHENDPUU ASSEMBLERPROGRAMM

Iseenesest on „puuprogrammi“ algoritm lihtne, ent vältimatud detailid teevad selle isegi C-keeles, rääkimata assemblerist, keerulisemalt programmeeritavaks. Vaatame, mida see programm peab tegema.

- Käsurealt käivitatakse programm kui `>puu <puufail>`. Esimene töö on tuvastada, kas etteantud nimega puu on kettal või ei ole. Kui pole, siis tuleb alustada dialoogi (otsi – lisa – kustuta), kui aga on, siis tuleb puu kettalt mällu lugeda, seisust ülevaate saamiseks loetu mingil moel trükkida ja minna üle dialoogile.
- Kui kettal on etteantud nimega puu, siis on ta meie programmi poolt eelnevalt kettale kirjutatud – vajame puu kirjutamise moodulit, ja seansi lõpus kirjutame (ehitatud või (võimalik, et modifitseeritud)) puu kettale. Puu lugemiseks on vaja kirjutajaga „sümmeetrilist“ moodulit.
- Dialoogi käigus sisestame võtme ja teeme uue, puuga sidumata tipu (seal on võti ja kaks tühiviita). Kui uue võtmega tippu puus polnud, siis see tipp lisatakse, kui aga oli, siis on kaks võimalust: kui tippu ei taheta kustutada, siis kuvatakse tema info ning vabastatakse „uue tipu“ mälu. Kui aga tahetakse tippu kustutada, siis fikseeritakse leitud tipu alampuude viidad, kustutatakse viit kustutatavale tipule, vabastatakse tema ja “uue tipu“ mälu, ning lisatakse kustutatud tipu alampuud puusse. See seletab, miks „ronijale“ ei anta ette otsimisvõtit, vaid antakse viit tipule: kustutamise puhul on need viidad alampuudele, mis puusse lisatakse (ja „ronija“ töötab rekursiivselt).
- Seansi lõpus antakse kasutajale võimalus kirjutada kettale

tasakaalustatud (*AVL*)-puu; selleks tuvastatakse puu tippude arv, tehakse tipuviitade (võtmeväärtuste järgi kasvavas järjekorras) vektor ning modifitseeritakse seni aktuaalne olnud puu.

11.2.1. PUU PÕHIPROGRAMM

```
;puu.asm :: ots-jrs-kahendpuu. >puu nimi 30.04.19. Mina
Ise
global main
extern _printf
extern _fopen
extern _fclose
extern _wp
extern _rp
extern _pp
extern _vjp
extern pjv
extern uus
extern gets
extern _strlen
extern roni
extern kriips
extern _reva
extern _makeavl
;--------------------------------------------------
section .data
   viga db ‘faili nimi?’,10,0
   mood db ‘rb’,0
   w_mood db ‘wb’,0
   anna db ‘key=’,0
;-----------------------------------------
section .bss
   struc top
     .key  resb 32
     .v    resd 1
     .p    resd 1
   endstruc
```

```
   mf resd 1 ;FILE *mf
   juur resd 1
   new  resd 1
   key  resb 32
;-----------------------------------------------
section .text
_main:
   push ebp
   mov  ebp,esp
   push ebx
   push esi
   push edi

   mov  eax,dword[ebp+8] ;argc
   cmp  eax,2 ;käsurea kontroll
   jnl  ava  ;puufaili avama

   push viga  ; db ‘faili nimi?’,10,0
   call _printf
   add  esp,4
   jmp  aut

ava:
;mf=fopen(nimi,mood)
   mov  ebx,dword[ebp+12] ;*argv
   push mood
   push dword[ebx+4] ;argv[1] : puu nimi
   call _fopen
   add  esp,8
   test eax,eax  ;cmp eax,0
   jz   ring  ;puu pole kettal : uus
   mov  dword[mf],eax  ;loen puu kettalt
;struct top *rp(FILE *mf)
   push eax
   call _rp
   add  esp,4
```

```
   cmp  eax,0
   je   aut  ;arusaamatu viga
   mov  dword[juur],eax

   call _kriips
   push dword[juur]
   call _pp  ;puu trükk eesjärjekorras
   add  esp,4
   call _kriips

   push dword[juur]
   call _vjp  ;võtmete trükk väärtuste kasvavas jrk-s
   add  esp,4
   call _reva

   push dword[juur]
   call _pjv  ;võtmete trükk väärtuste kahanevas jrk-s
   add  esp,4
   call _reva
;---- DIALOOG --------------
ring:
   push anna  ; db ‘key=’,0
   call _printf
   add  esp,4
   push key
   call _gets
   add  esp,4
   push key
   call _strlen
   add  esp,4
   cmp  eax,0
   je   ots  ;võtme asemel ’Enter’: dialoogi lõpp

   push key
   call _uus  ;teen uue ’vaba’ tipu (vt. kustutamine)
   add  esp,4
   mov  dword[new],eax
```

```
   mov  ecx,dword[juur]
   cmp  ecx,0
   jne  ronima  ;puu on (juba) olemas
   mov  dword[juur],eax
   jmp  ring
;puu läbimine v->j->p
ronima:
   push dword[new]
   push dword[juur]
   call _roni
   add  esp,8
   mov   dword[juur],eax ;kui vana juur kustutati
   jmp  ring
;----Dialoogi lõpp ------
ots:
   call _kriips
   push dword[juur]
   call _pp  ;puu trükk mälust, eesjrk-s
   add  esp,4

   call _kriips

  push dword[juur]
   call _vjp
   add  esp,4
   call _reva
;puu => AVL
   push dword[juur]
   call _makeavl
   mov  [juur],eax
   add  esp,4
   call _kriips
   push dword[juur]
   call _pp  ;AVL-puu trükk eesjrk-s
   add  esp,4
   call _kriips
;puu-faili sulgemine ja uue avamine kirjutamiseks
```

```
   push dword[ebx+4]
   call _fclose
   add  esp,4
;mf=fopen(nimi,mood)
   push w_mood
   push dword[ebx+4] ;argv[1]  ;puu nimi
   call _fopen
   add  esp,8
   mov  dword[mf],eax
;void wp(struct top *juur,FILE *mf)
   push eax
   push dword[juur]
   call _wp
   add  esp,8
aut:
   pop  edi
   pop  esi
   pop  ebx
   pop  ebp
   ret
```

11.2.2. ERALDI TRANSLEERITAVAD MOODULID

Need moodulid on koondatud eraldi faili *jupid.asm* ning neist transleeritakse objektfail käsuga

```
>nasm -f win32 jupid.asm -o jupid.obj
```

Allpool toome selle faili teksti. Algoritmid esitame mõnel puhul C-keelsete avakommentaariga – lootes, et nii on lihtsam assemblertekstist aru saada.

```
;jupid.asm :: puu.asm'i moodulid. 30.04.19. Mina Ise
global _wp
global _rp
global _pp
global _vjp
global _pjv
global _uus
global _roni
global _kriips
```

```
global _reva
global _makeavl

extern _fwrite
extern _malloc
extern _fread
extern _printf
extern _strcpy
extern _strcmp
extern _free
extern _gets
extern _strlen
;----------------------------------------
section .data
   sees db ‘sees: %s’,10,0
   tf db ‘key=%s v=%s p=%s’,10,0
   anna db ‘key=’,0
  kf db ‘%s ‘,0
   rv db 10,0
   della db ‘kustutan? j/e:     ‘,0
   jutt db ‘-----------------------------------’,0
   jf   db ‘%s’,10,0
   p_tipa db ‘tippude arv=%d’,10,0
   kt db ‘%s ‘,0
   reva db 10,0
;--------------------------------------------
section .bss
   struc top
     .key  resb 32
     .v    resd 1
     .p    resd 1
   endstruc
   n resd 1
   prev resd 1
   jee resb 6
   tipa resd 1  ;topude arv
   tvektor resd 1  ;tipuviitade vektor kasvavas jrk-s
```

```
   J resd 1  ;0..(tipa-1)*4
   low  resd 1 ;binary-sort-index
   high resd 1 ;binary-sort-index
;---------------------------------------------
section .text
_kriips:  ;vormistus: kriips teemade vahele
   push ebp
   mov  ebp,esp
   push jutt
   push jf
   call _printf
   add  esp,8
   pop  ebp
   ret
;----------------------------------------
_reva:  ;vormistus: reavahetus
   push ebp
   mov  ebp,esp
   push rv
   call _printf
   add  esp,4
   pop  ebp
   ret
;--------------------------------------
;void wp(struct top *t,FILE *mf)  j->v->p : puu kettale
_wp:
   push ebp
   mov  ebp,esp
   push ebx
   mov  ebx,dword[ebp+8]  ;*t
   mov  dword[n],top_size
;fwrite(t,sizeof(struct top),1,mf)
   push dword[ebp+12] ;*mf
   push 1
   push dword[n]
   push ebx
   call _fwrite
```

```
   add  esp,16
   mov  eax,dword[ebx+top.v]
   cmp  eax,0
   je   wkp
   push dword[ebp+12]  ;*mf
   push eax
   call _wp
   add  esp,8
wkp:
   mov  eax,dword[ebx+top.p]
   cmp  eax,0
   je   waut
   push dword[ebp+12]  ;*mf
   push eax
   call _wp
   add  esp,8
waut:
   pop  ebx
   pop  ebp
   ret
;---------------------------------------------

;struct top *rp(FILE *mf) : j->v->p : loe kettalt puu
_rp:
   push ebp
   mov  ebp,esp
   push ebx
   mov  dword[n],top_size  ;sizeof(struct top)
   push dword[n]
   call _malloc
   add  esp,4
   mov  ebx,eax ;struct top *t
;fread(t,sizeof(struct top),1,mf)
   push dword[ebp+8] ;*mf
   push 1
   push dword[n]
   push ebx
```

```
   call _fread
   add  esp,16
   push ebx
   push sees
   call _printf
   add  esp,8
   mov  eax,dword[ebx+top.v]
   cmp  eax,0
   je   kp
   push dword[ebp+8]  ;*mf
   call _rp
   add  esp,4
   mov  dword[ebx+top.v],eax
kp:
   mov  eax,dword[ebx+top.p]
   cmp  eax,0
   je   aut
   push dword[ebp+8]  ;*mf
   call _rp
   add  esp,4
   mov  dword[ebx+top.p],eax
aut:
   push dword[ebx+top.p]
   push dword[ebx+top.v]
   push ebx
   push tf
   call _printf
   add  esp,16
   mov  eax,ebx
   pop  ebx
   pop  ebp
   ret
;-----------------------------------------------
;void pp(struct top *t) : puu trükk : j->v->p
_pp:
   push ebp
   mov  ebp,esp
```

```
   push ebx
;juur          --          printf("key=%s          v=%s
p=%s\n",t->key,t->v->key,t->p->key)
   mov ebx,dword[ebp+8] ;struct top *t
   push dword[ebx+top.p]
   push dword[ebx+top.v]
   push ebx
   push tf
   call _printf
   add  esp,16
;vasak -- kui on vasak alampuu, roni sinna
   mov  eax,dword[ebx+top.v]
   cmp  eax,0
   je   kpp
   push eax  ;*t
   call _pp
   add  esp,4
;parem -- kui on parem alampuu, roni sinna
kpp:
   mov  eax,dword[ebx+top.p]
   cmp  eax,0
   je   paut
   push eax  ;*t
   call _pp
   add  esp,4
paut:
   pop  ebx
   pop  ebp
   ret
;-----------------------------------------------
;void vjp(struct top *t) : võtmete trükk : v->j->p
_vjp:
   push ebp
   mov  ebp,esp
   push ebx
   mov ebx,dword[ebp+8] ;struct top *t
;vasak
```

```
   mov  eax,dword[ebx+top.v]
   cmp  eax,0
   je   jvjp
   push eax  ;*t
   call _vjp
   add  esp,4
;juur
jvjp:
   push ebx
   push kf
   call _printf
   add  esp,8
;parem
   mov  eax,dword[ebx+top.p]
   cmp  eax,0
   je   autvjp
   push eax  ;*t
   call _vjp
   add  esp,4
autvjp:
   pop ebx
   pop ebp
   ret
;-------------------------------------------------

;void pjv(struct top *t) : võtmete trükk : p->j->v
_pjv:
   push ebp
   mov  ebp,esp
   push ebx

   mov ebx,dword[ebp+8] ;struct top *t
;parem
   mov  eax,dword[ebx+top.p]
   cmp  eax,0
   je   jpjv
   push eax  ;*t
```

```
   call _pjv
   add  esp,4
;juur
jpjv:
   push ebx
   push kf
   call _printf
   add  esp,8
;vasak
   mov  eax,dword[ebx+top.v]
   cmp  eax,0
   je   autpjv
   push eax  ;*t
   call _pjv
   add  esp,4
autpjv:
   pop ebx
   pop ebp
   ret
;-------------------------------------------------
;struct top *uus(char *key) : tee uus vaba tipp
_uus:
   push ebp
   mov  ebp,esp
   push ebx

   mov  dword[n],top_size  ;sizeof(struct top)
   push dword[n]
   call _malloc
   add  esp,4
   mov  ebx,eax ;struct top *t
;strcpy(top->key,key)
   push dword[ebp+8]
   push ebx
   call _strcpy
   add  esp,8
   mov  eax,0
```

```
   mov  dword[ebx+top.v],eax
   mov  dword[ebx+top.p],eax
   mov  eax,ebx
   pop  ebx
   pop  ebp
   ret
;-----------------------------------------
;struct top *roni(struct top *juur,struct top *new)
;juure tagastamine: juhuks, kui vana juur kustutati
_roni:
   push ebp
   mov  ebp,esp
   push ebx
   push esi
   push edi
   mov  ebx,dword[ebp+8]  ;juur
    mov  edi,ebx   ;tava-väljundväärtus (muu, kui juur
kustutati)
   mov  eax,0
   mov  dword[prev],eax  ;kustutamise jaoks: 0, kui juur
;ronimine mööda puud: v->j->p
check:
   push ebx ;tee alates juurest, trüki jaoks
   push kf
   call _printf
   add  esp,8
   push ebx  ;top.key (ebx+top.key, aga top.key=0)
   push dword[ebp+12]  ;key
   call _strcmp ;võti ⇔ tipuvõti
   add  esp,8
   cmp  eax,0
   jl   vasak
   jg   parem
leitud:
   call _reva
   push dword[ebx+top.p]
   push dword[ebx+top.v]
```

```
   push ebx
   push tf
   call _printf  ;leitud tipu täisinfo
   add  esp,16
   push della  ; db ‘kustutan? j/e:
   call _printf
   add  esp,4
   push jee
   call _gets
   call _strlen
   add  esp,4
   cmp  eax,0  ;tühi ’Enter’: ei kustuta
   je   vaba
   mov  al,byte[jee]
   cmp  al,’j’
   jne  vaba  ;ei kustuta
;tipu eemaldamine puust
   mov  eax,dword[prev]  ;ülemustipp
   cmp  eax,0
   jne  polejuur
;juure kustutamine
   mov  eax,dword[ebx+top.v]
   cmp  eax,0
   je   pal
   mov  edi,eax  ;uus juur
   mov  eax,dword[ebx+top.p]
   cmp  eax,0
   je   vaba
;kustutatud tipu alampuud -> puu
   push eax
   push edi
   call _roni
   add  esp,8
   jmp  vaba
pal:
   mov  eax,dword[ebx+top.p]
   cmp  eax,0
```

```
   je   vaba
   mov  edi,eax
vaba:  ;annan 'vaba tipu' mälu tagasi
   push ebx
   call _free
   add  esp,4
   jmp  raut
;----------------------------------------
polejuur:
   mov  esi,dword[prev] ;kustutatava tipu ülemus
   cmp  ebx,dword[esi+top.v]
   jne  pall
   mov  eax,0
   mov  dword[esi+top.v],0
   jmp  reku
pall:
   mov  eax,0
   mov  dword[esi+top.p],eax
reku:
   mov  eax,dword[ebx+top.v]
   cmp  eax,0
   je   pa
   push eax
   push edi
   call _roni ;kustutatud tipu vasak alampuu => puu
   add  esp,8
pa:
   mov  eax,dword[ebx+top.p]
   cmp  eax,0
   je   vaba
   push eax
   push edi
   call _roni ;kustutatud tipu parem alampuu => puu
   add  esp,8
   jmp  vaba
;--- tavaronimine ----------------------
vasak:
```

```
   mov  eax,dword[ebx+top.v]
   cmp  eax,0
   jne  valla
   mov  eax,dword[ebp+12]
   mov  dword[ebx+top.v],eax  ;uus tipp puusse

   push dword[ebp+12]
   push kf
   call _printf
   add  esp,8
    push rv
   call _printf
   add  esp,4
   jmp  raut
valla:
   mov  dword[prev],ebx
   mov  ebx,eax
   jmp  check
;--- tavaronimine -----
parem:
   mov  eax,dword[ebx+top.p]
   cmp  eax,0
   jne  palla
   mov  eax,dword[ebp+12]
   mov  dword[ebx+top.p],eax  ;uus tipp puusse
   push dword[ebp+12]
   push kf
   call _printf
   add  esp,8
   push rv
   call _printf
   add  esp,4
   jmp  raut
palla:
   mov  dword[prev],ebx
   mov  ebx,eax
   jmp  check
```

```
raut:
   mov  eax,edi  ;eax=juur
   pop  edi
   pop  esi
   pop  ebx
   pop ebp
   ret
;=====================================================
=====
;Otsimis- ja järjestamiskahendpuu tasakaalustamine (=>
AVL)
;struct top *makeavl(struct top *juur)
_makeavl:
   push ebp
   mov  ebp,esp
   mov  dword[tipa],0
   push dword[ebp+8]
   call tippude_arv  ;järjestatud vektori pikkus
   add  esp,4
   push dword[tipa]
   push p_tipa  ; db ‘tippude arv=%d’,10,0
   call _printf
   add  esp,8
   mov  eax,dword[tipa]
   shl  eax,2  ;tipa*4
   push eax
   call _malloc
   add  esp,4
   mov  dword[tvektor],eax  ;tipuviitade vektor
   mov  dword[J],0 ;vektori kirjutamis-indeks
   push dword[ebp+8]
   call maketv  ;inorder v->j->p: tipuvektori täitja
   add  esp,4
   mov  eax,dword[tipa]
   sub  eax,1
   push eax
   push 0
```

```
   call v2avl ;'vector to AVL'
   add  esp,8
   pop  ebp
   ret
;-----------------------------------------------------

;void tippude_arv(struct top *t) : eelnevalt tipa=0.
tippude_arv:
   push ebp
   mov  ebp,esp
   push esi
   mov  esi,dword[ebp+8]
   cmp  dword[esi+top.v],0
   je   manu
   push dword[esi+top.v]
   call tippude_arv
   add  esp,4
manu:
   mov  eax,dword[tipa]
   inc  eax
   mov  dword[tipa],eax
   cmp  dword[esi+top.p],0
   je   aidaa
   push dword[esi+top.p]
   call tippude_arv
   add  esp,4
aidaa:
   pop  esi
   pop  ebp
   ret
;-----------------------------------------------

;void maketv(struct top *t) : v->j->p tipuviidad ->
'tvektor'
;kirjutamisindeks on J: 0..tipa-1. Kohalik moodul.
maketv:
   push ebp
```

```
   mov  ebp,esp
   push esi
   mov  esi,dword[ebp+8]
   cmp  dword[esi+top.v],0
   je   lisa
   push dword[esi+top.v]
   call maketv
   add  esp,4

lisa:
   mov  eax,dword[tvektor]
   mov  ecx,dword[J]
   mov[eax+ecx],esi
   add  ecx,4
   mov  dword[J],ecx

   cmp  dword[esi+top.p],0
   je   yles
   push dword[esi+top.p]
   call maketv
   add  esp,4
yles:
   pop  esi
   pop  ebp
   ret
;-----------------------------------------------

;struct top *vector2AVL(int low,int high)[1]{
;    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);
;       }
```

1 Idee on pärit kahendotsimise algoritmist ([K&R], lk. 58).

```
;    return(t);
;}

v2avl:
   push ebp
   mov  ebp,esp
   push esi
   push edi
      push edx

   mov  edi,0 ;t=NULL
   mov  esi,dword[tvektor]
   mov  edx,dword[ebp+8]  ;low
   mov  ecx,dword[ebp+12]  ;high
   cmp  edx,ecx
   jg   annaviit
   add  edx,ecx
   shr  edx,1  ;mid
   mov  eax,edx
   shl  eax,2
   mov  edi,dword[esi+eax] ;tv[mid]
   mov  eax,edx
   sub  eax,1
   push eax ;high
   mov  eax,dword[ebp+8] ;low
   push eax
   call v2avl
   add  esp,8
   mov  dword[edi+top.v],eax
   mov  eax,dword[ebp+12] ;high
   push eax
   mov  eax,edx
   add  eax,1
   push eax
   call v2avl
   add  esp,8
   mov  dword[edi+top.p],eax
```

```
annaviit:
   mov  eax,edi
   pop  edx
   pop  edi
   pop  esi
   pop  ebp
   ret
;-----------------------------------------------------
-----
```

11.2.3. TESTID

Programmi töö näitlikustamiseks jooksutasime teda mõned korrad ning tegime ekraanipilte, mis on allpool ära toodud ning mõnevõrra ka kommenteeritud. Esimese testina ehitasime joonisel 11.1.a esitatud puu. Ekraanitõmmise (joon. 11.2.3.a) alguses saame näha pakkfaili *p.bat* teksti ja toimimist ja seejärel näeme, et dialoogi lõppedes välja trükitud puu on oodatud kujuga. Seanss päädib puu tasakaalustamisega (milleks pole tegelikult vajadust, puus on ainult üks ühe alluvaga tipp – võtmega *siga*) ning näeme, et puu on mälus ümber mängitud, uus juurtipp on märgendiga *kass* ning puus on endiselt ainult üks ühe alluvaga tipp (*eesel*). Asi on selles, et puus on 6 tippu ning võtmete järgi järjestatud tipuviitade vektori pikkus on seega ka 6 (vektori indeksid on vahemikus 0...5) ning *AVL*-puu[1] uueks juureks saab tipp vektori-indeksiga 2: 5/2=2, vasakult paremale 0) eesel, 1) kana, 2) kass [3) rebane, 4) siga, 5) siil]. Juhime lugeja tähelepanu sellele, et puus „ronimise“ käigus trükitakse ekraanile alati tee puu juuresr „jooksva“ tipuni.

1 Siin on kohane märkida, et *AVL*-puu on tasakaalustatud puud tähistav termin. 1962. a. publitseerisid N. Liidu matemaatikud Georgi Adelson-Velski ja Jevgeni Landis uurimuse, kus defineeriti tasakaalustatud puu omadused ja anti algoritm iga tipu lisamise või kustutamise puhuseks viivitamatuks puu tasakaalustamiseks. Meie programm nii keerulist tööd ei tee. (vt. [AVL]).

```
Administrator: cmd - Shortcut

C:\4_asm>nasm -f win32 jupid.asm -o jupid.obj

C:\4_asm>nasm -f win32 puu.asm -o puu.obj

C:\4_asm>gcc jupid.obj puu.obj -o puu.exe

C:\4_asm>puu fauna
key=rebane
key=siga
rebane siga
key=kana
rebane kana
key=siil
rebane siga siil
key=eesel
rebane kana eesel
key=kass
rebane kana kass
key=
------------------------------------
key=rebane v=kana p=siga
key=kana v=eesel p=kass
key=eesel v=(null) p=(null)
key=kass v=(null) p=(null)
key=siga v=(null) p=siil
key=siil v=(null) p=(null)
------------------------------------
eesel kana kass rebane siga siil
tippude arv=6
------------------------------------
key=kass v=eesel p=siga
key=eesel v=(null) p=kana
key=kana v=(null) p=(null)
key=siga v=rebane p=siil
key=rebane v=(null) p=(null)
key=siil v=(null) p=(null)
------------------------------------

C:\4_asm>_
```

Joonis 11.2.3.a. *Puu.exe* tegemine ja faili *fauna* ehitamine.

Järgmise näitena ehitame täielikult „paremkallakuga“ puu: sisestame võtmed nende väärtuste kasvavas järjekorras ning puu asemel ehitame lihtahela, millest otsimise ajaline keerukus on $O(n^2)$ *AVL*-puu $O(n \times log_2 n)$ asemel. Selle puu tasakaalustamine peaks näitama algoritmi tõhusust.

```
Administrator: cmd - Shortcut

C:\4_asm>puu fauna1
key=ahv
key=alligaator
ahv alligaator
key=boa
ahv alligaator boa
key=eesel
ahv alligaator boa eesel
key=gorilla
ahv alligaator boa eesel gorilla
key=hobune
ahv alligaator boa eesel gorilla hobune
key=
------------------------------------
key=ahv v=(null) p=alligaator
key=alligaator v=(null) p=boa
key=boa v=(null) p=eesel
key=eesel v=(null) p=gorilla
key=gorilla v=(null) p=hobune
key=hobune v=(null) p=(null)
------------------------------------
ahv alligaator boa eesel gorilla hobune
tippude arv=6
------------------------------------
key=boa v=ahv p=gorilla
key=ahv v=(null) p=alligaator
key=alligaator v=(null) p=(null)
key=gorilla v=eesel p=hobune
key=eesel v=(null) p=(null)
key=hobune v=(null) p=(null)
------------------------------------

C:\4_asm>
```

Joonis 11.2.3.b. Puu tasakaalustamine.

Järgmine test on mõeldud näitlikustama rekursiivse programmi tööd: kettalt loetakse puu valemiga *j->v->p* ning allaliikudes trükitakse kõigi loetud tippude märgendid – ekraanitõmmisel on nende marker *sees*. Mooduli lõpus kuvatakse tagastatava tipu info – selle tipu aadressiga kirjutatakse üle eelmise seisu kettalt loetud viit. Ja lõpuks tagastatakse viit esimesena loetud tipule – puu juurele.

```
Administrator: cmd - Shortcut

C:\4_asm>puu fauna1
sees: boa
sees: ahv
sees: alligaator
key=alligaator v=(null) p=(null)
key=ahv v=(null) p=alligaator
sees: gorilla
sees: eesel
key=eesel v=(null) p=(null)
sees: hobune
key=hobune v=(null) p=(null)
key=gorilla v=eesel p=hobune
key=boa v=ahv p=gorilla
------------------------------------
key=boa v=ahv p=gorilla
key=ahv v=(null) p=alligaator
key=alligaator v=(null) p=(null)
key=gorilla v=eesel p=hobune
key=eesel v=(null) p=(null)
key=hobune v=(null) p=(null)
------------------------------------
ahv alligaator boa eesel gorilla hobune
hobune gorilla eesel boa alligaator ahv
```

Joonis 11.2.3.c. Rekursiivne puu tipphaaval kettalt lugemine.

Viimased testid näitavad tipu kustutamise toimimist. Esimeses näites kustutame puu juure.

```
Administrator: cmd - Shortcut

C:\4_asm>puu fauna1
sees: alligaator
sees: ahv
key=ahv v=(null) p=(null)
sees: eesel
sees: hobune
key=hobune v=(null) p=(null)
key=eesel v=(null) p=hobune
key=alligaator v=ahv p=eesel
------------------------------------
key=alligaator v=ahv p=eesel
key=ahv v=(null) p=(null)
key=eesel v=(null) p=hobune
key=hobune v=(null) p=(null)
------------------------------------
ahv alligaator eesel hobune
hobune eesel alligaator ahv
key=alligaator
alligaator
key=alligaator v=ahv p=eesel
kustutan? j/e:     j
ahv eesel
key=
------------------------------------
key=ahv v=(null) p=eesel
key=eesel v=(null) p=hobune
key=hobune v=(null) p=(null)
------------------------------------
ahv eesel hobune
tippude arv=3
------------------------------------
key=eesel v=ahv p=hobune
key=ahv v=(null) p=(null)
key=hobune v=(null) p=(null)
------------------------------------

C:\4_asm>_
```

Joonis 11.2.3.d. Puu juure kustutamine.

Viimases näites kasutame taas oma suuremat puud *fauna* ja kustutame sealt ühe vahetaseme-tipu. Juhime lugeja tähelepanu tõigale, et kustutatud tipu alampuud (kui nad on), antakse „ronijale“ ette kui uued tipud – see garanteerib puu omaduste säilimise. Ekraanitõmmisel kuvatakse teed juurest uu(t)e alampuu(de) juur(t)eni.

```
Administrator: cmd - Shortcut
------------------------------------
eesel kana kass rebane siga siil
siil siga rebane kass kana eesel
key=siga
kass siga
key=siga v=rebane p=siil
kustutan? j/e:     j
kass rebane
kass rebane siil
key=
------------------------------------
key=kass v=eesel p=rebane
key=eesel v=(null) p=kana
key=kana v=(null) p=(null)
key=rebane v=(null) p=siil
key=siil v=(null) p=(null)
------------------------------------
eesel kana kass rebane siil
tippude arv=5
------------------------------------
key=kass v=eesel p=rebane
key=eesel v=(null) p=kana
key=kana v=(null) p=(null)
key=rebane v=(null) p=siil
key=siil v=(null) p=(null)
------------------------------------

C:\4_asm>
```

Joonis 11.2.3.e. Vahetaseme tipu kustutamine

12 GRAAF

12.1. PROGRAMMEERIJA VAADE

Microsoft arvutileksikoni ([CD], lk. 162) järgi on graaf programmeerija jaoks andmesstruktuur, mis koosneb nullist või enamast omavahel seotud tipust (*node*[1]). Iga tipupaar võib (aga ei pruugi) olla omavahel seotud serva (*edge*)[2] abil. Sidusas graafis leidub tee liikumiseks suvalisest tipust suvalisse teise tippu. Seejuures praktilist huvi pakkuvates ülesannetes tegeletakse servade asemel kaartega – kaar on „graafi orienteeritud serv (*a,b*), mis *väljub* tipust *a* ja *suubub* tippu *b*.“ ([Kaasik], lk. 69). Kaarele saab omistada pikkuse. Joonistel kujutatakse tippe tavaliselt ringide, servi joonte ning kaari noolte abil. Pikkus kirjutatakse kaare kõrvale.

Kui graafis on n tippu, siis saab teda kujutada $n \times n$-maatriksina G. Tipud on nummerdatud (1..n) ja kui tipust x suubub kaar tippu y pikkusega z, siis maatriksis $G(x,y)=z$.

Algoritmide ajalise keerukuse mõttes kuuluvad graafiülesanded raskemate hulka. Suhteliselt lihtne ja kiire on lühimate teede leidmine graafis (*Dijkstra* algoritmi ajaline keerukus on sõltuvalt konkreetsest graafist kuni $O(n^2)$) – mis on veel suhteliselt hea. Ent tee-otsingu ülesanne on ka „rändkaupmehe probleem“: alustada mingist tipust, läbida kõik ülejäänud tipud, käies igas neist ainult üks kord, ja naasta lähtetippu. Lühim selline tee on proovireisijaülesande lahend. Keerukuselt kuulub see ülesanne NP-täielike hulka [LPP][3] – mis tähendab, et

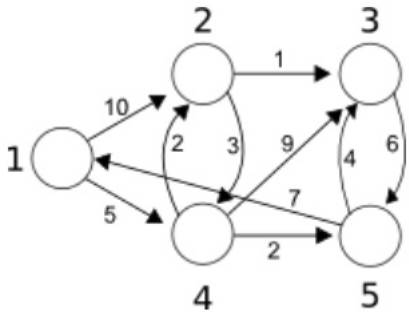
1 Paralleelnimetus on *vertex*.

2 Serv seob tippe; kui tipud A ja B on nii seotud, siis saab liikuda nii A → B kui ka vastupidi. Sel juhul öeldakse, et graaf pole orienteeritud.

3 Intuitiivselt, suhteliselt tihedas $n \times n$-maatriksi puhul tuleb teha äksteise sees n tsüklit, igaüks indeksitega 1..n. Mullimeetodil vektori järjestamisel tehakse üks tsükkel tsüklis ja ajaline keerukus on $O(n^2)$ ning siit üldistades saame variandi „n tsüklit tsüklis“ ajaliseks keerukuseks $O(n^n)$.

praktilise kasutamise jaoks „jõumeetod“ (mida allpool demonstreerime) ei ole vastuvõetav, ning programmeerida tuleb ta heuristikat kasutades[1].

Järgmises alapeatükis esitame *NASM*-programmi *graaf.sm*, mis leiab kõik teed „tipust *a* tippu *b*“, sh. on lubatud, et *a*=*b*. See tähendab, et leitud teede hulgas on nii *Dijkstra tee* kui ka proovireisija oma. Programm kasutab „toorest jõudu“, seega on ta praktilise kasutamise võimalused üsna piiratud. Andmetena kasutame ühe *Dijkstra* algoritmi tutvustava (sh. koos C-programmiga) allika [scvalex[2]] oma – kasvõi sellepärast, et testida oma programmi leitud lühimaid teid.



Joonis 12.1.a. Etteantud graaf [scvalex].

Järgmisel joonisel kujutame selle graafi maatriksina, seejuures lisame 0-rea ja -veeru – hõlbustamaks indekseerimist teid leidvas programmi *teed.asm*.

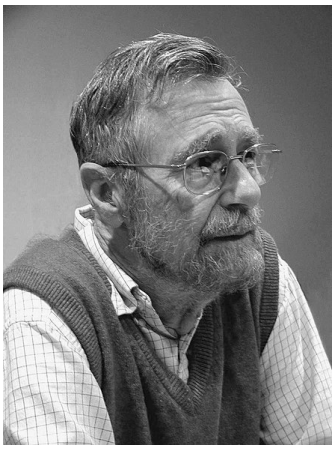
	0	1	2	3	4	**5**
0	0	0	0	0	0	0
1	0	0	10	0	5	0
2	0	0	0	1	3	0
3	0	0	0	0	0	6
4	0	0	2	9	0	2
5	0	7	0	4	0	0

Joonis 12.1.b. Etteantud graafi maatriksesitus.

1 Heuristikast vt. näit. [Isotamm C], lk. 272 jj.

2 Seda varjunime kasutab Alexandru Scvortov.

12.2. DIJKSTRA ALGORITM: SCVALEXI REALISATSIOON



Edsger Wybe Dijkstra, Holland, 1930–2002

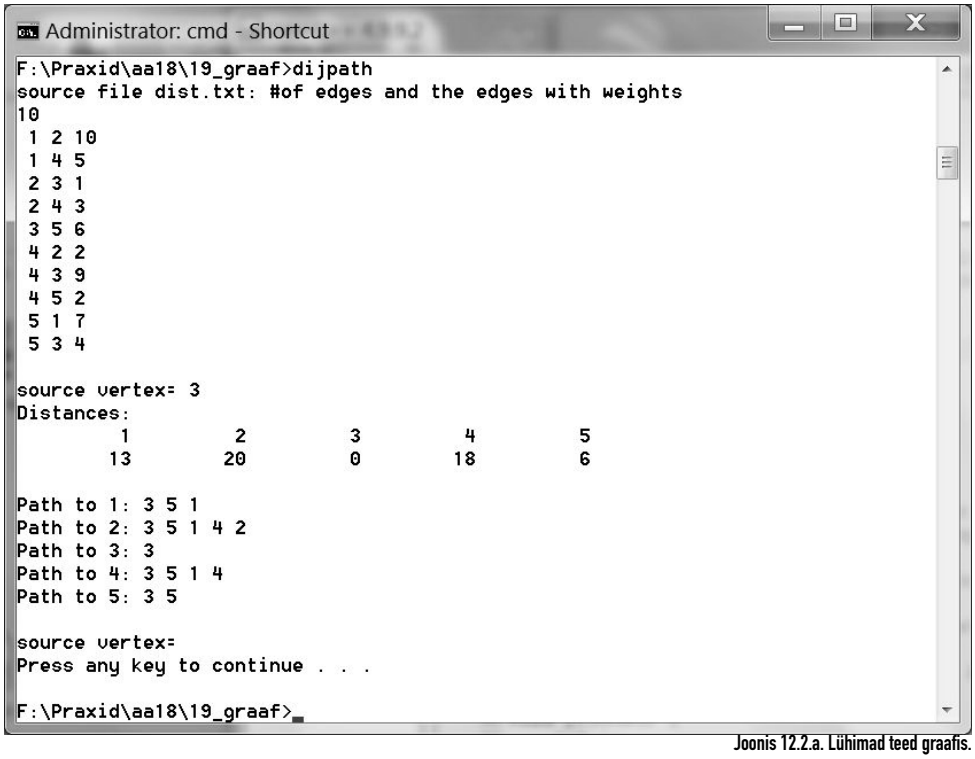
Toome selles alapeatükis *Scvalex*i C-programmi lühimate teede leidmiseks graafis. Nende ridade autor modifitseeris pisut täitmisaegset keskkonda (tekstis on need lisandused kommentaaridena ära märgitud).

```
//dijpath.c :: Dijkstra.c koos teedega.
//Here is the updated source: dijkstraWithPath.c. 30.03.12
//http://compprog.files.wordpress.com/2008/01/dijkstra.c

#include <stdio.h>
#include <stdlib.h> //A.I.
#include<string.h>  //A.I.

#define GRAPHSIZE 2048
#define INFINITY GRAPHSIZE*GRAPHSIZE
#define MAX(a, b) ((a > b) ? (a) : (b))

int e; /* The number of nonzero edges in the graph */
int n; /* The number of nodes in the graph */
long dist[GRAPHSIZE][GRAPHSIZE];
 /* dist[i][j] is the distance between node i and j; or 0 if there is no direct connection */
long d[GRAPHSIZE];
/* d[i] is the length of the shortest path between the source (s) and node i */
int prev[GRAPHSIZE];
/* prev[i] is the node that comes right before i in the shortest path from the source to i*/
```

```
void printD() {
      int i;

      printf("Distances:\n");
      for (i = 1; i <= n; ++i)
            printf("%10d", i);
      printf("\n");
      for (i = 1; i <= n; ++i) {
            printf("%10ld", d[i]);
      }
      printf("\n");
}

/*
 * Prints the shortest path from the source to dest.
 *
 * dijkstra(int) MUST be run at least once BEFORE
 * this is called
 */
void printPath(int dest) {
      if (prev[dest] != -1)
            printPath(prev[dest]);
      printf("%d ", dest);
}

void dijkstra(int s) {
      int i, k, mini;
      int visited[GRAPHSIZE];

      for (i = 1; i <= n; ++i) {
            d[i] = INFINITY;
            prev[i] = -1; /* no path has yet been found
to i */
            visited[i] = 0; /* the i-th element has not
yet been visited */
      }
```

```
	d[s] = 0;

	for (k = 1; k <= n; ++k) {
		mini = -1;
		for (i = 1; i <= n; ++i)
			if (!visited[i] && ((mini == -1) ||
(d[i] < d[mini])))
				mini = i;

		visited[mini] = 1;

		for (i = 1; i <= n; ++i)
			if (dist[mini][i])
				if (d[mini] + dist[mini][i] <
d[i]) {
					d[i] = d[mini] + dist[mi-
ni][i];
					prev[i] = mini;
				}
	}
}

int main(int argc, char *argv[]) {
	int i, j;
	int u, v, w;
	char arv[6];  //A.I.

	FILE *fn = fopen("dist.txt", "r");
	fscanf(fn, "%d", &e);
	for (i = 0; i < e; ++i)
		for (j = 0; j < e; ++j)
			dist[i][j] = 0;
	n = -1;
	for (i = 0; i < e; ++i) {
		fscanf(fn, "%d%d%d", &u, &v, &w);
		dist[u][v] = w;
		n = MAX(u, MAX(v, n));
```

```
            }
            fclose(fn);

//minu osa (A. I.)
    printf("source file dist.txt: #of edges and the edges
with weights\n");
    system("type dist.txt");
kysi:
    printf("\nsource vertex= ");
    gets(arv);
    if(strlen(arv)==0) goto aut;
    j=arv[0]-'0';
    if((j==0)||(j>n)) goto aut;
    dijkstra(j);
    printD( );
    printf("\n");
```



Joonis 12.2.a. Lühimad teed graafis.

```
for (i = 1; i <= n; ++i) {
                printf("Path to %d: ", i);
                printPath(i);
                printf("\n");
            }
   goto kysi;
 aut:
   system("pause");
   return 0;
}
```

12.3. KÕIK TEED GRAAFIS

12.3.1. MAATRIKSI EHITAMINE JA TEEDE LEIDMINE

Scvalex – Alexandru Scvortov – kasutab maatriksi ehitamiseks segastrateegiat: maksimaalne maatriks on staatiliselt reserveeritud[1] ja nullidega täidetud ning aktuaalne maatriks sisestatakse sinna failist *dist.txt*. Selle esimesel real on kaarte arv *k* ja järgmisel *k* real kaared kujul *x y z*, kus *x* on lähte- ja *y* suubumistipp ning *z* pikkus.

Meie järgisime graafi sisestamisel põhilises *Scvalex*i algoritmi. Peamine erinevus on, et tippude arv n on faili *kaared.txt* esimesel real enne kaarte arvu:

```
5 10
```

ja maatriksile võtame mälu dünaamiliselt ning kasutame kahemõõtmelise massiivi vektoresitust – nagu talitavad ka arvutist sõltumatute keelte kompilaatorid; rea- ja veeruindeksite *i* ja *j* järgi arvutatakse elemendi suhtaadress vektoris valemiga ($i \times$ *veergude_arv*)+*j* (*i* ja *j* algavad 0-st)[2].

Strateegia on lihtne: $n \times n$-maatriksi *G* jaoks tehakse vektor tee[n+1] (lisaväli on selle juhu jaoks, kui tuleb naasta algpunkti). Alguseks fikseeritakse lähtetipp *s* (start) ning lõpp-tipp *f* (finiš), nullitakse *tee*, *tee*[0]=*s* ja tee-elementide arv *w*=1, *i*=*s*. Seejärel järgitakse rekursiivselt algoritmi:

(1): Maatriksi *i*-ndal real leitakse *i*-tipu naabrid: kõik veerud *j* (*j*=0...*n* ja *G*[*i*][*j*]

1 *#define GRAPHSIZE 2048* ja *long dist[GRAPHSIZE][GRAPHSIZE]*;

2 Indekseerimise hõlbustamiseks lisasime maatriksisse 0-rea ja 0-veeru, seega on meil (n+1) × (n+1)-maatriks.

> 0), ja kui tippu *j* pole tees, siis tee[++*w*]=j ja kontrollitakse, kas *j*=*f*. Kui jah, siis väljastatakse tee, *w* — ja naastakse eelmisele reale järgmise naabertipuga proovima. Kui tee pole leitud, siis *i*=*j* ja toimub rekursioon (1). Kui aga aktiivses reas on kõik naabrid läbi proovitud, siis minnakse rekursioonis tase kõrgemale tagasi ning kui ka *s*-rida on lõpuni läbi käidud, siis ongi leitud kõik teed tipust *s* tippu *f*. Kusjuures, külastatuse-kontroll arvestab erijuhuga, kus *s*=*f*.

Kõikide teede leidmise assemblerprogramm sai järgmine:

```
;graaf.asm :: K6ik teed graafis. 24.06.19. A.I.
global _main
extern _fopen
extern _fclose
extern _printf
extern _fscanf
extern _malloc
extern _memset
extern _gets
extern _strlen
extern _isdigit
extern _system
;----------------------------------------
section .data
   dist db ‘kaared.txt’,0
   sc_nk db ‘%d%d’,0  ;fscanf(mf,”%d%d”,n,karv)
   sc_kaar db ‘%d%d%d’,0 ;fscanf(mf,”%d%d%d”,x,y,z)
   mood db ‘rb’,0
   pole db ‘pole faili [kaared.txt]’,10,0
   viga db ‘%d on liiga suur’,10,0
   st db ‘start=’,0
   fi db ‘finish=’,0
   wf db ‘%d ‘,0  ;vektori trykk
   wfm db ‘%2d ‘,0  ;maatriksi trykk:  rajastamine
   rv db 10,0  ;reavahetus “\n”
   rada db ‘way= ‘,0
   fdist db ‘ :: %d km’,10,0
;vahelpealkirjad
   ptxt db ‘fail kaared.txt:’,10,0
```

```
   pve db ‘graaf vektorina:’,10,0
   pma db ‘graaf maatriksina:’,10,0
   tyk db ‘type kaared.txt’,0
;-----------------------------------------
section .bss
   mf resd 1 ;FILE *mf
   n resd 1 ;tippude arv
   G resd 1 ;vektori aadress
   karv resd 1 ;kaarte arv
   x resd 1  ;start-tipp
   y resd 1  ;finish-tipp s=>f
   z resd 1
   n1 resd 1  ;n+1
   N resd 1  ;4*(n+1)
   way resd 1 ;vektor
   arv resb 6  ;gets-puhver
   s resd 1  ;start-tipu nr
   f resd 1  ;finish-tipu nr
   k resd 1  ;tee (way) jooksev pikkus
;=================================================
=====
section .text
_main:
   push ebp
   mov  ebp,esp
   call avang
   cmp  eax,0
   je   out
;graafi sisestamine ja ehitamine
   push ptxt  ; db ‘fail kaared.txt:’,10,0
   call _printf
   add  esp,4
   push tyk  ; db ‘type kaared.txt’,0
   call _system
   add  esp,4
   push pve  ; db ‘graaf vektorina:’,10,0
```

```
        call _printf
        add  esp,4
        call p_vektor
        push pma  ; db ‘graaf maatriksina:’,10,0

        call _printf
        add  esp,4
        call p_maatriks
    Ring:
        push st  ;st db ‘start=’,0
        call _printf
        add  esp,4
        call sf
        mov  dword[s],eax
        cmp  eax,0
        je   out  ;dialoogi lõpp
        push fi  ;fi db ‘finish=’,0
        call _printf
        add  esp,4
        call sf
        mov  dword[f],eax
        cmp  eax,0
        je   out  ;dialoogi lõpp
        push way
        call waynull ;tee nullimine
        add  esp,4
        mov  dword[k],0
        mov  edx,dword[way]
        mov  eax,dword[s]
        mov  dword[edx],eax ;way[0]=start
        push 1 ;veerg 1
        push dword[s] ;rida ‘s’
        call pioneer  ;otsib kõik teed ’s’=>’f’
        add  esp,8
        cmp  eax,1
        jne  Ring
        call printway
```

```
   jmp  Ring
out:
   pop  ebp
   ret
;-----------------------------------------------------
---------
;int Gij(int n+1,int i,int j) : anna lahtri suhtaadress
vektoris
Gij:
   push ebp
   mov  ebp,esp
   xor  edx,edx ;korrutamine
   mov  eax,dword[ebp+8]  ;n: veergude arv
   imul eax,dword[ebp+12] ;reaindex i
   add  eax,dword[ebp+16] ;i*n+j
   shl  eax,2  ;sa*=4
   pop  ebp
   ret

;---int *avang(void)------------------------
avang:
   push ebp
   mov  ebp,esp
   push ebx
   push esi
   push edi
   ;avan kaared.txt: graaf
   push mood ;'rb'
   push dist ;kaared.txt
   call _fopen
   add  esp,8
   cmp  eax,0
   jne  on
   push pole  ; db 'pole faili [kaared.txt]',10,0
   call _printf
   add  esp,4
   mov  eax,0  ;faili pole -- lõpetan
```

```
   jmp  aut
on:
   mov  dword[mf],eax
;alustan skaneerimist

   push karv  ;kaarte arv
   push n     ;tippude arv
   push sc_nk
   push dword[mf]
   call _fscanf  ;fscanf(mf,"%d%d",n,karv)
   add  esp,16
   mov  eax,dword[n]
   inc  eax  ;n+1
   mov  dword[n1],eax
   mov  edx,0
   imul eax,eax ;(n+1)x(n+1)
   shl  eax,2   ;x4
   mov  dword[N],eax
   push eax  ;4*(n+1)
   call _malloc  ;ruum graafi-vektorile G
   add  esp,4
   mov  dword[G],eax
   mov  esi,eax  ;esi=G
   push dword[N]
   push 0
   push esi
   call _memset  ;vektori G nullimine
   add  esp,12
   mov  ecx,dword[karv]
;tsükkel: skaneeri x y z (tipp A tipp B kaugus)
ring:
   push ecx  ;peitu
   push z
   push y
   push x
   push sc_kaar
   push dword[mf]
```

```
   call _fscanf  ;fscanf(mf,"%d%d%d",x,y,z)
   add  esp,20
   push dword[y]
   push dword[x]
   mov  eax,dword[n]
   inc  eax
   push eax
   call Gij
   add  esp,12
   mov  edi,eax

   mov  eax,dword[z]  ;G[i][j]=z
   mov  dword[esi+edi],eax
   pop  ecx  ;peidust
   loop ring
;teen vektori 'way'tooriku: tee X => Y
   call vecs
   mov  dword[way],eax
   mov  eax,1  ;signaal: OK
aut:
   pop  edi
   pop  esi
   pop  ebx
   pop  ebp
   ret
;----------------------------------
vecs:
   push ebp
   mov  ebp,esp
   mov  ecx,dword[n1]
   shl  ecx,2  ;n1x4
   push ecx  ;4x(n+1)
   call _malloc
   add  esp,4
   pop  ebp
   ret
;---------------------------
```

```
vecnull:
   push ebp
   mov  ebp,esp
   mov  ecx,dword[n1]
   shl  ecx,2
   push ecx  ;4x(n+1)
   push 0
   push dword[ebp+8]  ;int vektor[n1]
   call _memset
   add  esp,12
   pop  ebp
   ret
;-----------------------------------------
waynull:
   push ebp
   mov  ebp,esp
   push dword[way]
   call vecnull
   add  esp,4
   pop  ebp
   ret
;-----------------------------------
;int sf(void): anna tipu number
sf:
   push ebp
   mov  ebp,esp
   push arv  ;char arv[6]
   call _gets  ;gets(arv)
   add  esp,4
   push arv
   call _strlen  ;strlen(arv)
   add  esp,4
   cmp  eax,0
   je   ots  ;strlen=0 : tühi 'Enter'
   xor eax,eax
   mov  al,byte[arv]
   push eax
```

```
   call _isdigit  ;arv[0] on nuber?
   add  esp,4
   cmp  eax,0
   je   ots  ;pole number
   xor eax,eax
   mov  al,byte[arv]
   sub  al,'0'  ;al=atoi(arv)
   cmp  eax,dword[n]
   jng  ots
   push eax  ;liiga suur tipu-number
   push viga
   call _printf
   add  esp,8
   xor  eax,eax
ots:
   pop  ebp
   ret
;---------------------------------------------
;int kaugus(int veergude_arv,int rida,int veerg): G[i]
[j]
kaugus:
   push ebp
   mov  ebp,esp
   push dword[ebp+16] ;j
   push dword[ebp+12] ;i
   push dword[ebp+8]  ;n
   call Gij
   add  esp,12
   mov  edx,dword[G]
   add  edx,eax ;lahter
   mov  eax,dword[edx]
   pop  ebp
   ret
;---------------------------------------------
;int pioneer(int rida,int veerg): teed X=>Y
pioneer:
   push ebp
```

```
   mov  ebp,esp
   push esi
   push edi
   push ebx
;k on tee (way) jooksev pikkus
   mov  edi,dword[ebp+12]  ;veerg j
   mov  esi,dword[ebp+8] ;rida i

;if((rida==f)&&(k>0))return 1
   cmp  esi,dword[f]
   jne  veerud
   mov  edx,dword[k]
   cmp  edx,0
   je   veerud
   mov  eax,1  ;pioneer leidis uue tee
   jmp  retu
;for(;veerg<n1;veerg++)
veerud:
   cmp  edi,dword[n1]
   jl   edasi
   mov  edx,dword[k]
   sub  edx,1
   mov  dword[k],edx ;k--
   mov  eax,0  ;pioneer ei leidnud teed
   jmp  retu  ;tee viimane tipp maha: tupik
;if(G[rida][veerg]&&(visited(veerg)==0)
edasi:
   push edi
   push esi
   push dword[n1]
   call kaugus
   add  esp,12
   cmp  eax,0  ;G[i][j]=0? ei: alla
   je   paremale  ;pole teed G[i]=>G[j]
   push edi
   call visited  ;kas j-tipus on oldud?
   add  esp,4
```

```
   cmp  eax,1
   je   paremale  ;tipus G[j] on juba oldud
;way[++k]=veerg ++j
   mov  ebx,dword[way]
   mov  edx,dword[k]
   inc  edx
   mov  dword[k],edx
   shl  edx,2  ;kx4: indekseerimiseks
   mov  dword[ebx+edx],edi
   push 1  ;alla: alustan veerust 1
   push edi ;alla: rida j
   call pioneer ;pioneer(j,1)
   add  esp,8
   cmp  eax,0  ;0: tupik
   je   paremale ;proovi järgmist veergu
   call printway ;leidsin tee, trükin välja
   mov  edx,dword[k]
   sub  edx,1
   mov  dword[k],edx  ;tee viimane tipp maha, jätkan
paremale:
   inc  edi  ;j++
   jmp  veerud  ;for(;veerg<n1;veerg++)
retu:
   pop  ebx
   pop  edi
   pop  esi
   pop  ebp
   ret
;-----------------------------------------
;//t: kandidaat-tipp, k: way pikkus
;int visited(int t){
;    int i;
;    for(i=1;i<=k;i++){
;      if((way[i]==t)||(way[i]==s))return(1);
;      }
;    return(0);
;}
```

```
visited:
   push ebp
   mov  ebp,esp
   push esi
   push edi

   mov  esi,dword[way]
   mov  edi,1  ;i=1
   mov  ecx,dword[k]
   mov  edx,dword[ebp+8]  ;t: kontrollitav veerg ‘j’
vaata:
   cmp  edi,ecx
   jg   v6ib
   mov  eax,dword[esi+edi*4]  ;way[i]
   cmp  eax,dword[ebp+8]  ;way[i]=t?
   je   olemas
   cmp  eax,dword[s]
   jne  veel
olemas:
   mov  eax,1  ;return 1
   jmp  v_aut
veel:
   inc  edi  ;i++
   jmp vaata
v6ib:
   mov  eax,0  ;return 0
v_aut:
   pop  edi
   pop  esi
   pop  ebp
   ret
;----------------------------------------
printway:
   push ebp
   mov  ebp,esp
   push ebx
   push esi
```

```
   push rada
   call _printf
   add  esp,4
   mov  ebx,dword[way]
   mov  esi,0
   mov  ecx,dword[k]
   cmp  ecx,0
   je   reva
   inc  ecx
pw:
   push ecx
   push dword[ebx+esi*4]
   push wf
   call _printf
   add  esp,8
   inc  esi
   pop  ecx
   loop pw
reva:
   call distants

   pop  esi
   pop  ebx
   pop  ebp
   ret
;---------------------------------------------
;teepikkuse arvutamine „paremalt vasakule“
distants:
   push ebp
   mov  ebp,esp
   push ebx
   push esi
   push edi

   mov  edi,dword[way]
   mov  esi,dword[k]  ;i=k
```

```
   shl  esi,2  ;i*4
   mov  ebx,0  ;s=0

jooks:
   cmp  esi,0
   je   tehtud
   push dword[edi+esi]
   sub  esi,4
   push dword[edi+esi]
   push dword[n1]
   call kaugus
   add  esp,12
   add  ebx,eax
   jmp  jooks
tehtud:
   push ebx
   push fdist
   call _printf
   add  esp,8
   pop  edi
   pop  esi
   pop  ebx
   pop  ebp
   ret
;--------------------------------
p_vektor:  ;graafi trükk vektorina
   push ebp
   mov  ebp,esp
   push esi
   push edi
   mov  edx,0 ;korrutamise jaoks
   mov  ecx,dword[n1]  ;n+1
   mov  eax,ecx
   imul eax,ecx
   mov  ecx,eax  ;n1 x n1: tsykliloendaja
   mov  edi,dword[G]
   mov  esi,0
```

```
liige:
   push  ecx  ;peitu
   push dword[edi+esi*4]
   push wf
   call _printf
   add  esp,8
   inc  esi
   pop  ecx
   loop liige
   push rv  ;reavahetus
   call _printf
   add  esp,4
   pop  edi
   pop  esi
   pop  ebp
   ret
;-------------------------------------
p_maatriks:  ;graafi trükk maatriksina
   push ebp
   mov  ebp,esp
   push esi
   push edi
   mov  edx,0 ;korrutamise jaoks
   mov  ecx,dword[n1]  ;n+1
   mov  eax,ecx
   imul eax,ecx
   mov  ecx,eax  ;n1xn1: tsykliloendaja
   mov  edi,dword[G]  ;'graafivektor'
   mov  esi,0 ;i=0
element:
   push ecx  ;peitu
   mov  eax,esi
   cdq
   mov  ecx,dword[n1]
   idiv ecx
   cmp  edx,0 ;edx: jääk i/n1
   jne  yle
```

```
        push rv  ;reavahetus: '\n'
        call _printf
        add  esp,4
    yle:
        push dword[edi+esi*4]
        push wfm  ;db '%2d ',0
        call _printf
        add  esp,8
        inc  esi
        pop  ecx  ;tsükliloendaja taastamine
        loop element
        push rv  ;reavahetus
        call _printf
        add  esp,4
        pop  edi
        pop  esi
        pop  ebp
        ret
```

12.3.2. SKANEERIMISEST

Ülalpool nägime, kui lihtsalt (ja elegantselt) A. Scvartov sai graafi tekstifailist teha *int*-maatriksi *crt*-funktsiooni *fscanf* abil.

Siinkohal näib olevat sobiv koht rääkida funktsiooni *scanf* kasutamisest. Tundub, et ta on kirjutatud skaneerimiseks tekstifailist (versioonis *fscanf*): tekstifail on programmeerija kontrolli all ja kui seal on vigu, siis on need hõlpsasti leitavad ning parandatavad. Käsurealt muutujate skaneerimist (versioon *scanf*) tuleks vältida. Miks: loomuldasa on see funktsioon mõeldud tsükliliseks tööks. Käsureale kirjutatu loetakse vaikimisi faili *stdin* ning teisendamata tekst jääb sinna ootama järgmist pöördumist *scanf*i abil. Kindlasti jääb sinna *Enter*i kood. Järgmisel pöördumisel lisatakse uus tekst *stdin*-puhvri lõppu ning teisendatakse miski puhvri järjest alustades. Ebameeldivaid ootamatusi saab vältida, kui enne skaneerimist puhastada sisendpuhver:

```
fflush(stdin);
```

Ent käsurealt skaneerimine pole kunagi programmeerija kontrolli all. Näiteks, tahame sisestada muutuja *x* väärtust. Programmis on:

```
int x;
...
scanf(„%d“,&x);
```

ning kui arvu asemel anda klaviatuurilt (küsimise „anna x“ peale) midagi muud – näiteks sümbol „x“ – annab *scanf* vea (tagastab signaali *EOF*, mida peab alati kontrollima), ent milles viga seisneb, jääb täpsustamata.

Siin on veel üks nüanss. Kui sisestamised käsurealt toimuvad dialoogi-tsüklis, siis on mugav signaliseerida dialoogi lõpust „tühja *Enter*iga“ – mis skaneerimisvariandis ei tööta. Soovitame seda nii programmeerida:

```
char arv[9];
int x;
...
ring:
printf („anna x: “);
gets(arv);
if(strlen(arv)==0)goto dialoogi_l6pp;
if(!isdigit(arv[0]){
            printf(„%c pole number\n“,arv[0]);
            goto ring;
            }
x=atoi(arv);
```

12.3.3. TESTID

Allpool esitame programmi *graaf* kaks testimispilti. Mainima peab, et meie ülalesitatud programm ei too välja ei lühimat teed graafis ega anna ka rändkaupehe-ülesande (start=finiš) lahendit – need tuleb vaatajal ise leida (programselt oleks see tehniliselt lihtne esitada, aga meie raamatu jaoks ei annaks see eriti midagi juurde): leidke ise lühim distants X=>Y või – kui X=Y – lühim tee, mille pikkus on n+1 (meie näites, 6-liikmeline tee).

Ja et programm väljastab kohe iga järjekordse tee, näitab kontrolltrükk algoritmi tööd: vaadakem nende teede sammhaaval muutuvaid prefikseid ja lisanduvaid sufikseid (teede jätke).

```
Administrator: cmd - Shortcut

C:\19_graaf>nasm -f win32 graaf.asm -o graaf.obj

C:\19_graaf>nasm -f win32 graaf.asm -o graaf.obj

C:\19_graaf>gcc graaf.obj -o graaf.exe

C:\19_graaf>graaf
fail kaared.txt:
5 10
 1 2 10
 1 4 5
 2 3 1
 2 4 3
 3 5 6
 4 2 2
 4 3 9
 4 5 2
 5 1 7
 5 3 4
 graaf vektorina:
0 0 0 0 0 0 0 0 10 0 5 0 0 0 0 1 3 0 0 0 0 0 0 6 0 0 2 9 0 2 0 7 0 4 0 0
graaf maatriksina:

 0  0  0  0  0  0
 0  0 10  0  5  0
 0  0  0  1  3  0
 0  0  0  0  0  6
 0  0  2  9  0  2
 0  7  0  4  0  0
start=

C:\19_graaf>
```

Joonis 12.3.3.a. Graafi sisestamine ja mälus ehitamine.

```
Administrator: cmd - Shortcut
start=1
finish=1
way= 1 2 3 5 1  :: 24 km
way= 1 2 4 3 5 1  :: 35 km
way= 1 2 4 5 1  :: 22 km
way= 1 4 2 3 5 1  :: 21 km
way= 1 4 3 5 1  :: 27 km
way= 1 4 5 1  :: 14 km
start=1
finish=2
way= 1 2  :: 10 km
way= 1 4 2  :: 7 km
start=1
finish=3
way= 1 2 3  :: 11 km
way= 1 2 4 3  :: 22 km
way= 1 2 4 5 3  :: 19 km
way= 1 4 2 3  :: 8 km
way= 1 4 3  :: 14 km
way= 1 4 5 3  :: 11 km
start=1
finish=4
way= 1 2 4  :: 13 km
way= 1 4  :: 5 km
start=1
finish=5
way= 1 2 3 5  :: 17 km
way= 1 2 4 3 5  :: 28 km
way= 1 2 4 5  :: 15 km
way= 1 4 2 3 5  :: 14 km
way= 1 4 3 5  :: 20 km
way= 1 4 5  :: 7 km
start=

C:\19_graaf>
```

Joonis 12.3.3.b. Tipust 1 algavad teed graafis.

Viimasest pildist – näiteks – selgub, et lühim tee 1=>2 on 7 km (1 4 2) või rändkaupmehe-ülesande lahend on pikkusega 21 km (1 4 2 3 5 1).

Ent, veelkord: meie teedeleidmise algoritm (moodul pioneer) on lühike ja lihtne, ent kaugel optimaalsusest. Kusjuures rändkaupmehe ülesande jaoks pole teada vastuvõetava kiirusega töötavat *algoritmi* ning graafis sellist teed otsivad reaalsed programmid kasutavad kõik heuristikat – mis annab tavaliselt konkreetse ülesande jaoks vastuvõetava kiirusega töötava programmi, ent ei anna algoritmi – mis peab olema universaalne.

13 MATEMAATILISE KAASPROTSESSORI (X87) KASUTAMINE

13.1. UJUPUNKTARVUD

Matemaatiline kaasprotsessor opereerib ujupunktarvudega[1]. Ülo Kaasiku definitsioon: „Liikuva koma arv, ujukomaarv – kujul m•p esitatud arv, kus m ∈ (-1, 1) on arvu *mantiss*, *k* vastava positsioonilise arvusüsteemi alus ja täisarv *p* arvu *järk*“ [Kaasik, lk. 99]. Arvutis on *k* mõistagi 2. Arvutileksikon [CD] ütleb, et sel moel saab arvutis kujutada nii väga suuri kui ka väga väikseid arve. Mantiss esitab arvu numbreid ning järk (eksponent) „komakohta“ ja toob näiteks kaks kümnendarvu: 314600000 ja 0.0000451, mis ujupunktesituses on vastavalt 3146E5 ja 451E-7 (arvusüsteemi alus on 10).

Paul A. Carter [Carter, lk. 119] kirjutab, et Intel kasutab kaht *IEEE*[2] poolt välja töötatud ujupunktkahendarvude formaati (C *float* (32 bitti) ja *double* (64 bitti))[3]. Arvu märgi määrab mõlemal juhul vasakpoolseima biti (vastavalt kohal 31 või 63) väärtus (0 – positiivne, 1 – negatiivne). Esimese formaadi puhul on järgu jaoks bitid 30 ... 23 ja teise puhul 62 ... 52. Kõik ülejäänud bitid on mantissi jaoks.

Lõpuks, x87 kasutab ise **alati 80-bitilist registri-formaati** (järgu jaoks 15 bitti). Sellisesse registrisse saab mälust laadida nii ühe- kui ka kahekordse täpsusega (*float* ja *double*) ujupunktarve ning neljabaidiseid *int*-arvusid (need teisendatakse laadimise käigus 80-bitisteks ujupunktarvudeks). Mällu kirjuta-

1 i..k *floating point notation*, ka exponential notation, eesti k. ujukomaarv, ujupunktarv või liikuva koma arv.

2 *Institute of Electrical and Electronic Engineers.*

3 Vastavalt ühekordse ja topelttäpsusega (ik. *single ja double precision*). Katsetused näitavad, et meie *NASM* toetab ainult topelttäpsust (vt. joonis 13.3.1.a).

misel teisendatakse arv „kolmandast formaadist“ programmis deklareeritud 32-bitiseks *int*-arvuks või salvestatakse ta 64-bitise *double*-arvuna.

Kaasprotsessoril x87 pole „oma mälu“ – ta kasutab x86 oma – ning arvutuste operandide jaoks on 8 registrit (á 80 bitti); x86 ei saa kasutada kaasprotsessori x87 registreid ja x87 ei saa kasutada x86 omi. Koostööks tuleb „vahepuhvrina“ kasutada op-mälu.

Kaasprotsessoril on 8 10-baidilist üldregistrit nimedega *st0, st1,.. st7*[1]. Loogilisel tasemel on nad realiseeritud kui *LIFO*-tüüpi magasin. *Push*-operatsiooni analoog (*fld* või *fild*) „lükkab“ selle magasini viida 80 bitti „allapoole“, tippu jääb alati viimati-lisatu (metakeeles *st0*), ning ülejäänute „aadressid“ nihkuvad: *st(i)=st(i+1)*. Magasin saab täis, kui „*i*“ > 7[2]. Tipmise registri roll meenutab kaugeid „pesamasina“-aegu, kus enamik tehteid käis läbi *summaatori*. Register *st0* on enamiku tehete vaikimisi-operand. Registriindeksite nihkumine tundub esmapilgul arusaamatu ja eksitavana, aga noist enamikku läheb reaalselt vaja keerukate arvutusvalemite programmeerimisel muutujate salvestamiseks ning nende nimed ja rollid pannakse paika juba arvutuskäigu planeerimisel ja arvutamise käigus on nad kõik kontrolli all.

Põhiprotsessor x86 käivitab kaasprotsessori x87 programmis ujupunktkäsu leidmisel ning edasi töötavad nad paralleelselt. See tekitab sünkroniseerimisprobleemi siis, kui ujupunktarvutuste programmilõigu tulemusi tahab põhiprogramm kasutada: resultaati tuleb töö jätkamiseks oodata. Varasemates assemblerites tuli programmeerijal kasutada direktiivi *fwait* („oota“), nüüdsed assemblertranslaatorid lisavad selle direktiivi ise.

13.2. KÄSUSTIK

Ujupunktprotsessori mnemokoodid algavad kõik „*f*“-tähega. Käsu operandideks on kas üks või kaks ujupunktregistrit, või x86 mäluväli (täis- või ujupunktarv)[3].

1 st = *stack*.

2 Vt. näidet osas 13.3.2.

3 x87 opereerib lisaks 10-baidiste pakitud kümnendarvudega, ent need ei mahu meie raamatu raamidesse.

Vahetut operandi kasutada ei saa. Otstarbe järgi jagunevad nad järgmistesse gruppidesse:

- Andmeedastus (*data transfer*);
- Aritmeetika;
- Võrdlemine;
- Varia.

Selle peatüki koostamisel on kasutatud peamiselt Paul A. Carteri raamatut [Carter, lk. 125 – 141], ning veebimaterjale [Amrozek], [Motorola], [Smith] ja [Ray].

13.2.1. ANDMEEDASTUS

- *fld* – laadi ujupunktarv mälust registrisse *st0*;
- *fild* – teisenda mälust *int*-arv ja laadi registrisse *st0*;
- *fst* – teisenda ja kanna registrist *st0* 8-baidine ujupunktarv mällu;
- *fstp* – nagu *fst*, lisaks „pop“ *st0*;
- *fist* – teisenda *st0* → *int* ja kirjuta mällu;
- *fistp* – *fist* + „pop“;
- *fld1* – 1.0 → *st0*;
- *fldz* – 0.0 → *st0*;
- *fldpi* – π → *st0* (eks ole, kolm viimast korvavad pisut vahetute operandide puudumist);
- *fxch sti* – vahetus: *st0* ↔ *sti*;
- *ffree sti* – *sti* märgitakse kui „kasutamata“

Nelja viimase direktiivi testisime; et registrite vahetust vaadata, kirjutasime alati *st0*-st mällu käsuga *fst* – nii jäid kõik registritessekanded alles.

```
Administrator: cmd - Shortcut

C:\freq>nasm -f win32 pilt.asm -o pilt.obj

C:\freq>gcc pilt.obj -o pilt.exe

C:\freq>pilt
fldpi:
3.1416
fldz:
0.0000
fld1:
1.0000
fxch st2:
3.1416

C:\freq>_
```

Joonis 13.2.1.a. Konstantide kandmine magasini ja *fxch*-käsk.

13.2.2. ARITMEETIKA

Nelja põhitehte[1] jaoks on võimalustelt sarnased direktiivide komplektid, sj lahutamise ja jagamise jaoks on mittekommutatiivsusest tingitud lisavariandid. Allpool tähistab *src* (*source*) registrisse mälust laaditava operandi aadressi või mõne teise 80-bitise registri magasinielementi ning *dest* (*destination*) — kaasprotsessori registrit.

- *fadd|fsub|fmul|fdiv src : st0 ⊕ = src ;*
- *fadd|fsub|fmul|fdiv dest, st0 : dest ⊕ = st0 ;*
- *faddp|fsubp|fmulp|fdivp src : st0 ⊕ = src koos operatsiooniga „pop“;*
- *faddp|fsubp|fmulp|fdivp dest, st0 : dest ⊕ = st0 ja „pop“;*
- *fsubr|fdivr src : st0 = src ⊕ st0 ;*
- *fsubr|fdivr dest,st0 : dest = st0 ⊕ dest ;*
- *fsubrp|fdivrp dest,st0 : dest = st0 ⊕ dest ja „pop“;*
- *fiadd|fisub|fimul|fidiv src : st0 ⊕ = (float) src ;*
- *fiaddp|fisubp|fimulp|fidivp src : st0 ⊕ = (float) src ja „pop“;*
- *fisubr|fidivr src : st0 = (float) src ⊕ st0 ;*

Aritmeetikadirektiivide test on allpool, jaotises 13.3.1.

1 Liitmine (*fadd*), lahutamine (*fsub*), korrutamine (*fmul*) ja jagamine (*fdiv*). Tähistame kommentaarides neid üldistatult sümboli ⊕ abil, see on vastavalt kas +, -, × või /.

13.2.3. VÕRDLEMINE

- *fcom src* – võrreldakse *st0* ja src, viimane on ikka kas x87 register või ujupunktarv op-mälu aadressil;
- *fcomp src* – nagu eelmine, lisaks „pop“;
- *fcompp* – võrreldakse *st0* ja *st1*, mõlemad võrreldavad „popitatakse“ magasinist välja;
- *ficom src* – võrreldakse *st0* ja *src*, viimane on mälus olev *int*-arv, mis tehte sooritamise ajaks teisendatakse ujupunktarvuks;
- *ficomp src* – nagu eelmine, lisaks „pop“;
- *fist – st0 = 0?*
- *fcomi sti;*
- *fcomip sti* – kaasneb „pop“.

Paul Carteri raamatu [Carter] kirjutamise ajal olid probleemid x87 võrdlemisdirektiivide tule-muste kasutamisega x86 suunamisdirektiividega (x87 signaale ei kantud x86 *FLAGS*-registrisse) – ja nagu testimine näitas, pole *NASM*i 32-bitine variant seda siiani teinud; signaalide ülekandmiseks on vahendid olemas, ja kui kellelgi meie raamatu lugejaist on ülaltoodud direktiividest mõni tingimata vajalik, siis soovitame tal *Google*'i (või [Carter, lk. 129 jj.]) abil tutvuda x87 olekuregistri ja signaalide *FLAGS*-registrisse ülekandmise tehnikaga.

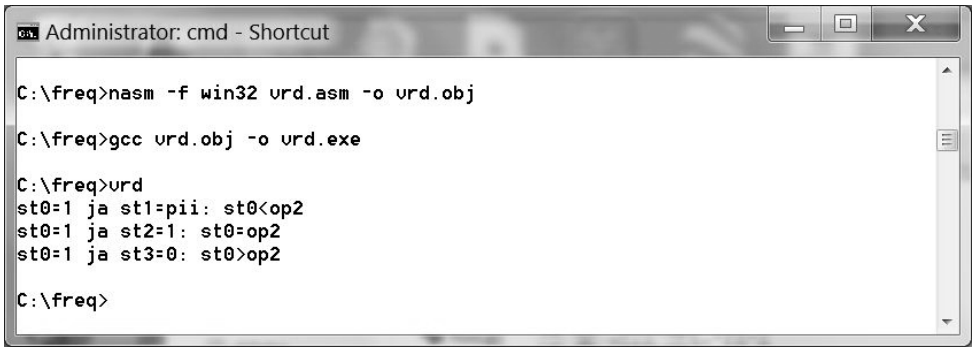
Ent kaks viimast (*fcomi* ja *fcomip*) – mis on kasutusel alates Intel Pentium II-st – saavad hakkama.

Allpool toome toimiva võrdlusdirektiivi testimise programmi ja selle lahendamise ekraanipildi.

```
;vrd.asm :: ujupunktarvude v6rdlemine: fcomi. 21.07.19.
A.I.
global _main
extern _printf

section .data
   n1 db ‘st0=1 ja st1=pii: ‘,0
   n2 db ‘st0=1 ja st2=1: ‘,0
   n3 db ‘st0=1 ja st3=0: ‘,0
```

```
   vk db ‘st0<op2’,10,0
   vr db ‘st0=op2’,10,0
   sr db ‘st0>op2’,10,0

section .text
_main:
   push ebp
   mov  ebp,esp
;-----------------------------------
   fldz
   fld1
   fldpi
   fld1
;st0=1, st1=pi, st2=1,st3=0
   push n1 ; db ‘st0=1 ja st1=pii: ‘,0

   call _printf
   add  esp,4
   fcomi st1 ;1 ja 1
   call switch
   push n2 ; db ‘st0=1 ja st2=1: ‘,0
   call _printf
   add  esp,4
   fcomi st2 ;1 ja pi
   call switch
   push n3 ; db ‘st0=1 ja st3=0: ‘,0
   call _printf
   add  esp,4
   fcomi st3 ;1 ja 0
   call switch
   pop  ebp
   ret
;-----------------------------------
switch:
   jb   veike
   je   vrd
   ja   suur
```

```
veike:
   push vk
   call _printf
   add  esp,4
   jmp  kokku

vrd:
   push vr
   call _printf
   add  esp,4
   jmp  kokku
suur:
   push sr
   call _printf
   add  esp,4
kokku:
   ret
;-----------------------------------
```



Joonis 13.2.3.a. Direktiivi *fcomi* test.

13.2.4. MUUD

- *fchs : st0 = -st0* – märgi muutmine;
- *fabs : st0 = abs(st0);*
- *fsqrt : st0 = ;*
- *fscale : st0 = st0×2st*[1].

Siin pole esitatud kõiki ujupunktoperandidega opereerivaid direktiive ning kui programmi koostamisel tundub, et midagi on vajaka, tasub alati

„guugeldada“. Nii puuduvad siinkohal näiteks trigonomeetriafunktsioonid, logaritmarvutuste abivahendid jpm.

13.3. KATSENÄITED

13.3.1. TESTID

Kirjutasime ujupunkt-kaasprotsessori käima saamiseks palju testprogramme, ja tulemused olid esialgu arusaamatud – katsetuste käigus selgus, et meie *NASM* ei toeta teisendust 80-bitisest ujupunktarvust 32-bitiseks – tugi on 8-baidisele formaadile – ja teine murekoht oli, kuidas *printf*-le edastada 8-baidist väärtust *res*[1].

Katsetasime lihtsa omistamistehtega $z=x+y$. Liidetavad on alul ujupunktarvud, ja siis *int*-arvud ning liitmiseks kasutasime erinevaid variante:

- x ja y ning z on 32-bitised ujupunktarvud; x pannakse x87 magasini ning talle liidetakse y, summa salvestatakse z-i teisendusega 80-bit => 32-bit. Teisendus ei toimi, väljatrükis $z=0$.
- z on 64-bitine (*qword*) ning väljatrükiks pannakse ta x86 magasini kahes osas. Töötab.
- Järgmistes testides on z jätkuvalt 64-bitine. Nii x kui ka y pannakse x87 magasini ning liitmiskäsk on ilmutatud kujul: *fadd st0,st1*.
- Sama, mis eelmine, aga liitmiskäsk on ilmutamata kujul (vaikimisi-operandid on magasini kaks tipmist elementi *st0* ja *st1*). Lihtsalt *fadd*.
- Sama, mis eelmine, aga liidetavad on *int*-arvud, need teisendatakse x87 magasini panekul ujupunktarvudeks.

Katseprogrammi(de) viimane versioon sai nime *pilt.asm* (mõeldes lahendusaegsele ekraanitõmmisele). Testprogramm sai järgmine:

```
 ;pilt.asm :: ujupunkti näited. 16.07.19. A.I.
global _main
extern _printf

section .data
   kaks dd 2.0
```

1 Selgus, et lihtne: *push res+4, push res.*

```
   kolm dd 3.0
   a    dd 1
   b    dd 2
   form db ‘%2.1f’,10,0
   n1 db ‘res0 resd 1..fld dword[kaks],fadd dword[kolm],
fstp res0 push dword[res0]:’,10,0
   n2 db ‘res1 resq 1..fld dword[kaks], fadd dword[kolm],
fstp res1 push dword[res1+4] push dword[res1]:’,10,0
   n3 db ‘fld dword[kaks], fld dword[kolm],fadd st0,st1,
fstp res1:’,10,0
    n4 db ‘fld dword[kaks], fld dword[kolm],fadd, fstp
res1:’,10,0
   n5 db ‘res1 resq 1..fild dword[a], fiadd dword[b], fstp
res1:’,10,0


section .bss
   res0 resd 1
   res1 resq 1
section .text
_main:
   push ebp
   mov  ebp,esp
;-------------------------------------
   push n1
   call _printf
   add  esp,4
   fld  dword[kaks]
   fadd dword[kolm]
   fstp dword[res0] ;80bit-float => 32-bit float
   push dword[res0] ;ei tööta
   push form
   call _printf
   add  esp,8
;-----------------------------------
   push n2
   call _printf
```

```
   add  esp,4
   fld  dword[kaks]
   fadd dword[kolm]
   fstp qword[res1] ;80-bit float => 64-bit float
   push dword[res1+4] ;töötab. 8 baiti 'push'
   push dword[res1]   ;kahes osas
   push form
   call _printf
   add  esp,12
;-------------------------------------
   push n3
   call _printf
   add  esp,4
   fld  dword[kaks] ;=> st0
   fld  dword[kolm] ;=> st0, 'kaks' => st1
   fadd st0,st1     ;st0 += st1
   fstp qword[res1]
   push dword[res1+4]
   push dword[res1]
   push form
   call _printf
   add  esp,12
;-----------------------------------------------
   push n4
   call _printf
   add  esp,4
   fld  dword[kaks]
   fld  dword[kolm]
   fadd              ;st0 += st1
   fstp qword[res1]
   push dword[res1+4]
   push dword[res1]
   push form
   call _printf
   add  esp,12
;-----------------------------------------------
   push n5
```

```
        call _printf
        add  esp,4
        fild  dword[a]  ;int a => (float) st0
        fiadd dword[b]  ;st0 += (float)int b
        fstp qword[res1]
        push dword[res1+4]
        push dword[res1]
        push form
        call _printf
        add  esp,12
    ;-----------------------------------
        pop  ebp
        ret
```

Testimistulemused on järgmisel joonisel.

```
Administrator: cmd - Shortcut

C:\freq>nasm -f win32 pilt.asm -o pilt.obj

C:\freq>gcc pilt.obj -o pilt.exe

C:\freq>pilt
res0 resd 1..fld dword[kaks],fadd dword[kolm], fstp res0 push dword[res0]:
0.0
res1 resq 1..fld dword[kaks], fadd dword[kolm], fstp res1 push dword[res1+4] pus
h dword[res1]:
5.0
fld dword[kaks], fld dword[kolm],fadd st0,st1, fstp res1:
5.0
fld dword[kaks], fld dword[kolm],fadd, fstp res1:
5.0
res1 resq 1..fild dword[a], fiadd dword[b], fstp res1:
3.0

C:\freq>
```

Joonis 13.3.1.a. Ujupunktarvutuste testid.

13.3.2. SÜMBOLITE SAGEDUSED

Ülesande püstitasime juba ülalpool (vt. 9.3.2) – seal näitasime sümbolite esinemissagedusi kujul osatähtsuse *täisosa::jääk.* – protsentide asemel. Allpool – kasutades ujupunktprotsessorit – esitame oma programmi uue versiooni. Aga – alustame hoiatavast kogemusest. Esmaversioonis me lisasime x87 magasini üle 8 elemendi, magasin sai täis, ja tulemus oli järgmine:

```
Administrator: cmd - Shortcut

C:\freq>nasm -f win32 f87.asm -o f87.obj

C:\freq>gcc fail.obj f87.obj -o f87.exe

C:\freq>f87 bender.txt
49 Laadige apelsinid tynnidesse. Vennad Karamazovid.
    4 8.16%
  . 2 4.08%
  K 1 2.04%
  L 1 2.04%
  V 1 2.04%
  a 7 14.29%
  d 5 10.20%
  e 5 10.20%
  g 1 -1.#J%
  i 5 -1.#J%
  l 1 -1.#J%
  m 1 -1.#J%
  n 5 -1.#J%
  o 1 -1.#J%
  p 1 -1.#J%
  r 1 -1.#J%
  s 3 -1.#J%
  t 1 -1.#J%
  v 1 -1.#J%
  y 1 -1.#J%
  z 1 -1.#J%
```

Joonis 13.3.2.a. „Stack overflow": x87 magasin sai täis.

Lahendus on magasini tipu välja kirjutamises koos kustutamisega, *fstp vs. fst.*

Programm:

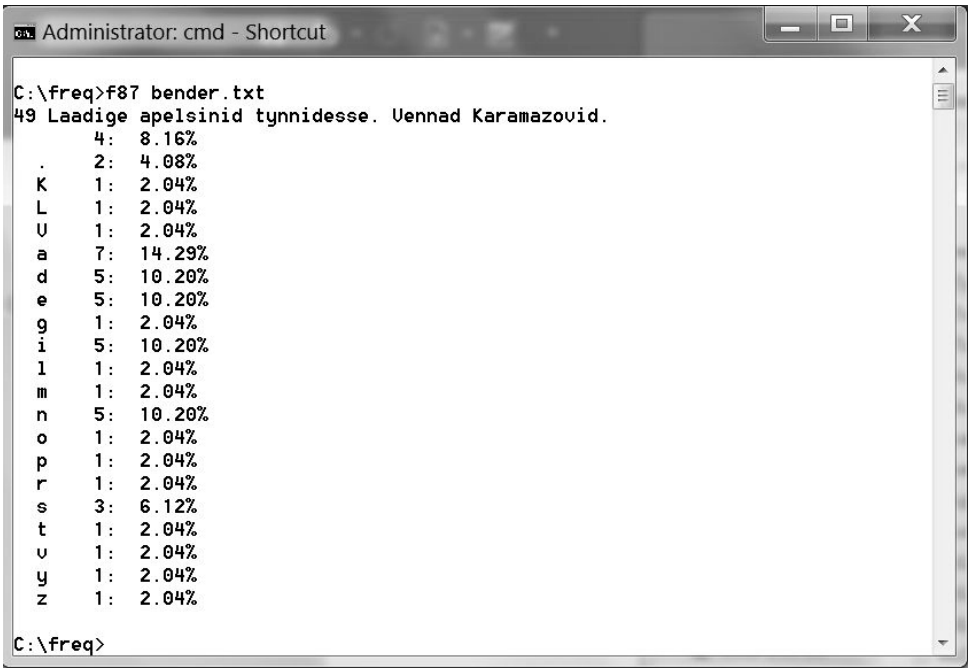
```
;f87.asm  ::  etteantud  faili  symbolite  sagedused.
15.07.19. A.I.
global _main
extern _fail   ;meie funktsioon
extern _printf

section .data
   viga db 'pole faili',10,0
   pf   db '  %c %4d:  %4.2f%%',10,0
   ty db '%d %s',10,0
   sada dd 100.0
section .bss
struc F
   .nimi resd 1
   .n resd 1
   .buf resd 1
   .mf resd 1
```

```
endstruc
    stabel resb 256
    prots resq 1
      tp resd 1
section .text
_main:
   push ebp
   mov  ebp,esp
   push ebx
   push esi
   push edi
;käsurea kontroll : >f87 <fail>
   mov  eax,dword[ebp+8] ;argc
   cmp  eax,2
   jnl  oki
   push viga  ; ‘pole faili’
   call _printf
   add  esp,4
   jmp  aut

oki:
;nullin sagedustabeli
   cld ; dest-flag: vasakult paremale
   mov  ecx,256
   mov  eax,0
   mov  edi,stabel
   rep stosb

;-------------------------------------
   mov  ebx,dword[ebp+12]  ;**argv
   push dword[ebx+4]  ;argv[1]
   call _fail
   add  esp,4
   cmp  eax,0
   je   aut
   mov  ebx,eax  ;parameetrite kirje
;kontrolltrykk:
```

```
   push dword[ebx+F.buf]
   push dword[ebx+F.n]
   push ty
   call _printf
   add  esp,12
;-----------------------------------
;sagedusvektori t2itmine
   mov  ecx,dword[ebx+F.n]
   mov  edi,dword[ebx+F.buf]
   mov  edx,stabel
   mov  esi,0
   xor  eax,eax
ring:
   mov  al,byte[edi+esi]
   add  byte[edx+eax],1
   inc  esi
   loop ring
;-----------------------------------
;esinevate symbolite sageduste trykk
   mov  ecx,256
   mov  esi,0
   mov  edi,stabel
ring2:
   xor  eax,eax
   mov  al,byte[edi+esi]
   cmp  eax,0
   je   next ;trykin ainult tekstis olevaid symboleid
   push ecx  ;peitu
;osatähtsus: (sagedus * 100)/n
   mov  dword[tp],eax
   fild dword[tp]
   fmul dword[sada]
   fidiv dword[ebx+F.n]
   fstp qword[prots] ;x87 magasin tyhjaks.
   push dword[prots+4] ;qword magasini kahes osas
   push dword[prots]
   xor  eax,eax ;sagedus
```

```
        mov  al,byte[edi+esi]
        push eax  ;ASCII kood
        push esi
        push pf
        call _printf
        add  esp,20
        pop  ecx  ;tsykliloendaja taastamine
    next:
        inc  esi
        loop ring2
    ;=====================================
    aut:
        pop  edi
        pop  esi
        pop  ebx
        pop  ebp
        ret
```

Korrektne sagedustabel – x87 magasini ei täideta üle – on järgmisel ekraanitõmmisel:



Joonis 13.3.2.b. Korrektne sagedustabel.

13.4. DIJKSTRA SORTEERIMISJAAM

Kolmandaks ujupunktarvutuste näiteks valisime aritmeetilise konstant-avaldise[1] (kus tehete täitmise järjekorra muutmiseks kasutatakse ümarsulge) väärtuse arvutamise. Konsoolilt sisestatud tekstikujul-avaldis (näit. -3*(5-2)) viiakse Dijkstra sorteerimisjaama-algoritmiga[2] aritmeetiline avaldis inverteeritud Poola kujule ning arvutatakse *LIFO*-tehnikat kasutades avaldise väärtus (vt. näit. [Isotamm, C, lk. 153 jj.]). Et meie raamatu kontekstis on oluline avaldise väärtuse arvutamine magasini abil assembleris, siis esitame allpool avaldise Poola kujule viimise ning keskkonna loomise programmi C-keeles. Assembler opereerib avaldise Poola kujuga.

13.4.1. PÕHIPROGRAMM

Põhiprogramm on C-keelne: võimaldab sisestada aritmeetilisi konstant-avaldisi ning lõpetab töö, kui järjekordse avaldise asemel anti '*Enter*'. Programm kasutab kolme magasini – kõigi ühine element on lüli-tüüpi – konsoolilt saadud tekst skaneeritakse *FIFO*[3]-tehnikaga lekseemide (operand või tehtemärk või sulg) ahelasse *AP* ja sellest tehakse lekseemide Poola kuju ahel – taas *FIFO*-tehnikaga, nimega *Pol*. Kolmas magasin (T) on *LIFO*[4]-tüüpi, see on avaldise Poola kujule viimise käigus säilitatavate tehtemärkide jaoks.

```
//ShY.c ::Dijkstra sorteerimisjaam. 31.07.19. Mina Ise
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>
struct lyli{
   char oper[32]; //operandi tekst
   double a;  //operand masinkujul
   struct lyli *next; //viit järgmisele või 0
   char tehe;  //'tehte' märk: (,),+,-,*,/
```

1 Operandid on arvkonstandid.

2 *Shunting yard.*

3 *First-In-First-Out.*

4 *Last-In-First-Out.*

```
};

void polint(struct lyli *pol); //assembler
struct lyli *AP=NULL; //(sulg)avaldise ahela pea
struct lyli *AS=NULL; //(sulg)avaldise ahela saba
struct lyli *Pol=NULL; //Poola kuju ahela pea
struct lyli *PolS=NULL; //Poola kuju ahela saba
struct lyli *T=NULL; //Dijkstra LIFO-tupik
struct lyli *cur,*tc,*tx,*ty;
char av[257]; //(sulg)avaldis klaviatuurilt
char *P; //sisendteksti järg
int i,j,k,n;
int rim; //0:veel pole arv, 1:kerin numbreid
//----------------------------------------
struct lyli *uuslyli( ){
   struct lyli *L;
   L=malloc(sizeof(struct lyli));
   memset(L,'\0',sizeof(struct lyli));
   return L;
}

void Apush(struct lyli *lyl){
     struct lyli *L;
     L=AS;
     if(L==NULL)AP=AS=lyl;
     else{
       L->next=lyl;
       AS=lyl;
       }
}

struct lyli *Apop(void){
     struct lyli *L;
     L=AP;
     if(L)AP=L->next;
     return(L);
}
```

```
void Ppush(struct lyli *lyl){
     struct lyli *L;
     L=PolS;
     if(L==NULL)Pol=PolS=lyl;
     else{
       L->next=lyl;
       PolS=lyl;
       }
}

struct lyli *Ppop(void){
     struct lyli *L;
     L=Pol;
     if(L)Pol=L->next;
     return(L);
}

void Tpush(struct lyli *lyl){
     lyl->next=T;
     T=lyl;
}

struct lyli *Tpop(void){
     struct lyli *L;
     L=T;
     if(L)T=L->next;
     return(L);
}

int isdd(char c){
    if(isdigit(c))return 1;
    if(c=='.')return 1;
    return 0;
}

int istehe(char c){
```

```
    switch(c){
      case '(': return 1;
      case ')': return 1;
      case '+': return 1;
      case '-': return 1;
      case '*': return 1;
      case '/': return 1;
      }
    return 0;
}

//ahela trykk
int pr_ahel(struct lyli *lyl){
    struct lyli *L;
    L=lyl;
next:
    if(L==NULL)return 0;
    if(L->tehe)printf("%c ",L->tehe);
    else printf("%s ",L->oper);
    L=L->next;
    goto next;
}

//formaalne kontroll:
int fork(void){
    int i;
    for(i=0;i<n;i++){
       if(istehe(av[i]))goto oki;
       if(isdd(av[i]))goto oki;
       if(av[i]==',')){
          av[i]='.';
          goto oki;
          }
       printf("ootamatu symbol %c\n",av[i]);
       return 0;
    oki:
       continue;
```

```
        }
    return 1;
}

  //sulgude paarsuse kontroll
  int lisp(void){
    int i,k=0;
    for(i=0;i<n;i++){
       if(av[i]=='(')k++;
       else if(av[i]==')')k--;
       }
    if(k==0)return 1;
    if(k>0)printf("%d (-sulg(usid) yle\n",k);
    else printf("%d )-sulg(usid) yle\n",-1*k);
    return 0;
}

//skanner: avaldise lekseemid => avaldise ahel A
void scan( ){
     int lipp; //0: ootan arvu 1: ootan (,),+,-,*,/
     P=&av[0];
     lipp=0;
next:
     if(*P=='\0')goto trykk;
     cur=uuslyli( );
     if(lipp==0)goto arv;
tehe:
     if(istehe(*P)){
        cur->tehe=*P;
        Apush(cur);
        *P++;
        lipp=0;
        goto next;
        }
arv:
     if(isdd(*P))goto j0; //operand algab numbriga
     if((*P=='+')||(*P=='-')){
```

```
            *P++;
            if(isdd(*P)){   //kas on +arv v6i -arv?
               *P--;
               cur->oper[0]=*P;  //+ v6i - => arv
               *P++;
               j=1;
               goto kanna;   //numbri(te) ylekanne
               }
            else{
               *P--;
               goto tehe;  //+ v6i - on tehe
               }
            }
         else goto tehe;  //muu märk kui + v6i -
j0:
         j=0;
kanna:
         while(isdd(*P)){
            cur->oper[j]=*P;
            *P++;
            j++;
            }
         Apush(cur);
         lipp=1;
         goto next;
trykk:
        printf("skaneeritud avaldis:\n");
        pr_ahel(AP);
        printf("\n");
}

//Dijkstra sorteerimisjaam: A => Pol. 'Shunting Yard'
int ShY( ){
next:
           cur=Apop( );
           if(cur==NULL)goto otsas;
           cur->next=NULL;
```

```
 //operand -->  Pol
       if(cur->tehe==0){
           cur->a=atof(cur->oper); //tekst=>ujupunkt
           Ppush(cur);
           goto next;
           }
//tehe --> LIFO-tupik
       switch(cur->tehe){
              case '(': Tpush(cur); break;
              case ')': check: tc=Tpop( );
                        if(tc->tehe!='('){
                             tc->next=NULL;
                             Ppush(tc);
                             goto check;
                             }
                        break;
                   case '*':
                   case '/': Tpush(cur); break;
                   case '-':
                   case '+': if(T==NULL)goto yle;
                       if(T->tehe=='+'||T->tehe=='-'){
                                 tc=Tpop( );
                                 tc->next=NULL;
                                 Ppush(tc);
                                 goto yle;
                                 }
                             uuri:
                                if(T->tehe=='*'||T->te-
he=='/'){
                                      tc=Tpop( );
                                      tc->next=NULL;
                                      Ppush(tc);
                                      if(T)goto uuri;
                                      else goto yle;
                                    }
                             yle:
                             Tpush(cur);
```

```
                                    break;
                        }
        goto next;

//avaldise ahel on otsas, tyhjendan tupiku (kui vaja)
otsas:
//stack => Poola
   while(T){
      tc=Tpop( );
      tc->next=NULL;
      Ppush(tc);
      }
    printf("avaldis inverteeritud Poola kujul:\n");
    pr_ahel(Pol);
    printf("\n");
}
int main( ){
ring:
    AP=AS=NULL; //avaldise FIFO-stacki ahela pea ja saba
     Pol=PolS=NULL; //inv. Poola kuju FIFO-ahela pea ja
saba
    T=NULL; //'Shunting Yardi' tupik: LIFO-stacki ahela
pea
    printf("\navaldis: ");
    gets(av);
    n=strlen(av);
    if(n==0)return 0; //tühi 'Enter'
    if(fork( )==0) goto ring; //näpuviga
    if(lisp( )==0) goto ring; //sulgude paarsuse viga
    scan( ); //lekseemid => avaldise ahel
    ShY( ); //avaldis=>Poola kuju
    polint(Pol); //Poola kuju interpretaator (NASM)
    goto ring;
}
```

13.4.2. X87 MAGASINI KASUTAMINE

Konstantavaldise – mis antakse ette *Pol*-ahelana (tüüp on *FIFO*) – väärtus arvutatakse x87 vahendite abil. Arvutusprogrammi tekst on järgmine:

```
;polint.asm :: inventeeritud Poola kuju ahela interpre-
taator. ;polint(struct lyli *Pol). 28.07.19. A.I.
global _polint
extern _printf
;-----------------------------------------
section .data
   frm db ‘res=%4.2f’,10,0
   pole db ‘pole operandi’,10,0
   eimahu db ‘fp-magasin ajab yle’,10,0
   jagaja db ‘jagaja = 0’,10,0
;---------------------------------------
section .bss
   struc lyli
      .oper resb 32
      .a    resq 1
      .next resd 1
      .tehe resb 1
   endstruc
   Poola resd 1
   res resq 1
;-------------------------------------------------
section .text
;void polint(struct lyli *Pol)
_polint:
   push ebp
   mov  ebp,esp
   push ebx
   call Fritz  ;8 ujupunktregistri vabastamine
   mov  ebx,dword[ebp+8]  ;Poola kuju ahela pea
   mov  dword[Poola],ebx ;muutuja: ahela 1. lyli
   cmp  ebx,0
   je   viga
   xor  ecx,ecx  ;stacki elementide arv
```

```
   jmp  algus
interpreet:
   mov  ebx,dword[Poola] ;jooksev 1. lyli
   cmp  ebx,0
   je   tryki  ;Pol-ahela l6pp
   mov  eax,dword[ebx+lyli.next]
   mov  dword[Poola],eax
   cmp  eax,0
   je   tryki  ;Pol-ahel on ammendatud
   mov  ebx,eax
algus:
   xor  eax,eax
   mov  al,byte[ebx+lyli.tehe]
   cmp  al,0
   jne  aritm
   cmp  ecx,8  ;kas x87 ujupunkt-stack on täis?
   jl   mahub
   push eimahu
   call _printf
   add  esp,4
   jmp  aut
mahub:
   fld  qword[ebx+lyli.a]
   inc  ecx  ;fp-stack'i kontrolli jaoks
   jmp  interpreet
aritm:
   cmp  ecx,1
   jne  binaar  ;2 operandi
   jl   viga  ;operande pole
   cmp  al,'+'  ;unaarne '+'?
   jne  um
   jmp  tryki
um:   ;unaarne '-'?
   cmp  al,'-'
   jne  viga
   fchs  ;muuda resultaadi märki
   jmp  tryki
```

```
binaar:   ;2 operandi
   cmp  al,'+'
   jne  lahuta
   faddp st1
   dec  ecx  ;st1-võrra vähem
   jmp  interpreet
lahuta:
   cmp  al,'-'
   jne  korruta
   fsubp st1
   dec  ecx
   jmp  interpreet
korruta:
   cmp  al,'*'
   jne  jaga
   fmulp st1
   dec  ecx
   jmp  interpreet
jaga:
   fldz  ;kas nulliga jagamine? 0=>St0
   fcomip st1  ;see on jagaja
   je jnull    ;ongi 0
   fdivp st1
   dec  ecx
   jmp  interpreet
tryki:
   cmp  ecx,0
   je   viga  ;st0=tühi
   fstp qword[res]  ;res0 => res
   push dword[res+4]  ;qword => x86-stack 2s 4-b osas
   push dword[res]
   push frm
   call _printf
   add  esp,12
   jmp  aut
jnull:
   push jagaja
```

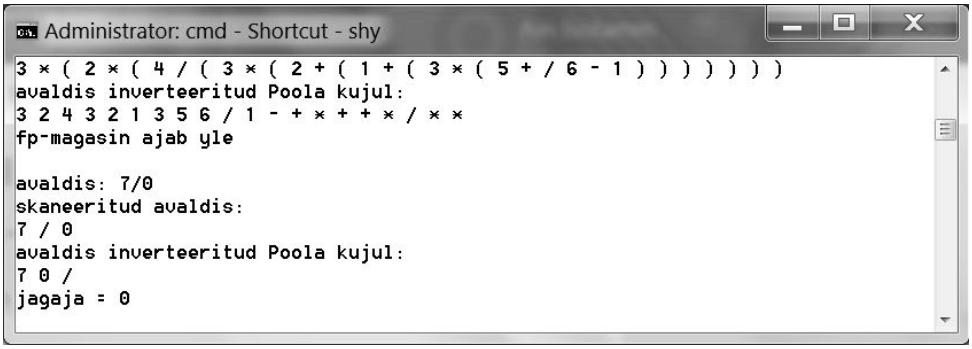
```
   call _printf
   add  esp,4
   jmp  aut
viga:
   push pole
   call _printf
   add  esp,4
aut:
   pop  ebx
   pop  ebp
   ret
;-----------------------------------------
Fritz:  ;ujupunkt-magasini tühjendamine
   ffree st0
   ffree st1
   ffree st2
   ffree st3
   ffree st4
   ffree st5
   ffree st6
   ffree st7
   ret
;-------------------------------------------
```

```
Administrator: cmd - Shortcut

C:\shy>shy

avaldis: 1+2+3+4+5
skaneeritud avaldis:
1 + 2 + 3 + 4 + 5
avaldis inverteeritud Poola kujul:
1 2 + 3 + 4 + 5 +
res=15.00

avaldis: 9-1+2-1+3
skaneeritud avaldis:
9 - 1 + 2 - 1 + 3
avaldis inverteeritud Poola kujul:
9 1 - 2 + 1 - 3 +
res=12.00

avaldis: -(5+1)/(8/4)
skaneeritud avaldis:
- ( 5 + 1 ) / ( 8 / 4 )
avaldis inverteeritud Poola kujul:
5 1 + 8 4 / / -
res=-3.00

avaldis:

C:\shy>_
```

Joonis 13.4.2.a. Konstantavaldiste lahedamine.

```
Administrator: cmd - Shortcut
avaldis: -3×(-7/1,19)
skaneeritud avaldis:
-3 × ( -7 / 1.19 )
avaldis inverteeritud Poola kujul:
-3 -7 1.19 / ×
res=17.65

avaldis: 3,14+-8/1.99
skaneeritud avaldis:
3.14 + -8 / 1.99
avaldis inverteeritud Poola kujul:
3.14 -8 1.99 / +
res=-0.88

avaldis: -1×2×-3
skaneeritud avaldis:
-1 × 2 × -3
avaldis inverteeritud Poola kujul:
-1 2 -3 × ×
res=6.00

avaldis:

C:\shy>_
```

Joonis 13.4.2.b. Märgiga operandid.



Joonis 13.4.2.c. Avariisituatsioonid.

13.4.3. LISAKS C- JA ASSEMBLERPROGRAMMIDE RISTKASUTUSEST

Eelmises alapeatükis tutvustatud Poola kuju interpretaatori programmeerimisel tekkis esmapilgul arusaamatu veaolukord: assembler-moodul lõpetas kohe avariiliselt. Vea lokaliseerimiseks kirjutasime lühema testprogrammi *felix.asm* ja kesta *jupid.c*. Viimase tekst:

```
//jupid.c :: felix.asm-i silumisprogramm. 30.07.19
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
struct lyli{
   char tehe;
   char oper[30];
   double a;
   struct lyli *next;
   };

int sz=sizeof(struct lyli);
struct lyli *cur;
struct lyli *felix(double y,double x,struct lyli *cur);

double x=3.0;
double y=7.0;  //avaldise väärtuse arvutamiseks
int main( ){
    cur=malloc(sz);
    memset(cur,'\0',sz);
    cur->tehe='-';
```

```
    felix(y,x,cur);
}
```

Kinnijooksva arvutusprogrammi testprogramm algas nii:

```
;felix.asm :: ShY aritmeetika. 26.07.19. A.I.
global _felix
extern _printf
section .data
   fo db 'res=%4.2f',10
section .bss
   struc lyli
      .tehe resb 1
      .oper resb 30
      .a    resq 1
      .next resd 1
   endstruc
   ree resq 1
```

Programm hakkas tööle, kui meenus mõiste rajastamine[1]: protsessori tasemel adresseerimine toimub kiiremini, kui andmevälja aadress jagub andmevälja pikkusega ilma jäägita, sõna (*word*) on paarisarvulisel aadressil, topeltsõna (*double word*) aadress jagub neljaga, ja et C kompilaator võib seda lihtsat optimeerimisvõtet kasutada (rajastamiskäsud on *int86*-l olemas). Tõepoolest, Kernighani ja Ritchie raamatus [K&R, lk.138] on näide, kus on kirjas, et struktuuri

```
struct {
    char c;
    int i;
};
```

maht pole mitte 5 baiti, vaid 8 – *c* ja *i* vahele jäetakse 3-baidine „auk“. Ja et *NASM*i translaator seda ei tee, on välja *.oper* suhtaadress assembleris 1, *jupid.c* aga edastab mälulõigu, kus tollelt suhtaadressilt algav 4-baidind väli *pole* kaitsepiirkonda kuuluv mäluaadress (kolm esimest baiti on *memset*i poolt „nullitud“) ning selle kasutamine *oper*-aadressi rollis annabki vea. Allpool esitame parandatud ja töötavate programmide tekstid.

```
//jupid.c :: felix.asm-i silumisprogramm. 30.07.19
```

1 i.k. *alignment*.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
struct lylio
   char oper[32]; //suhtaadress on 0
   double a;      //32
   struct lyli *next;  //40
   char tehe;   //44
};
int sz=sizeof(struct lyli);
struct lyli *cur;
struct lyli *felix(double y,double x,struct lyli *cur);

double x=3.0;
double y=7.0;  //avaldise väärtuse arvutamiseks


int main( ){
    printf("C-size=%d\n",sz);
    cur=malloc(sz);
    memset(cur,'\0',sz);
    cur->tehe='-';
    felix(y,x,cur); //cur->a=x-y
}
```

Programmis *felix.c* pöörake tähelepanu, kuidas saadakse magasinist kätte 8-baidised *double*-tüüpi parameetrid ning kuidas edastatakse *printf*-le *double*-arv.

```
;felix.asm :: ShY aritmeetika. 26.07.19. A.I.
global _felix
global _press
extern _printf
section .data
   fo db 'res=%4.2f',10,0
   sz db 'lyli_size=%d',10,0
section .bss
   struc lyli
```

```
      .oper resb 32
      .a    resq 1
      .next resd 1
      .tehe resb 1
   endstruc
ree resq 1

section .text
;struct lyli *felix(double y,double x,struct lyli *cur)
;y ja x on magasinis 8-baidised parameetrid
_felix:
   push ebp
   mov  ebp,esp
   push ebx

   mov  eax,lyli_size
   push eax
   push sz
   call _printf
   add  esp,8

   mov  ebx,dword[ebp+24]  ;*cur
   fld  qword[ebp+16]  ;x
   fld  qword[ebp+8]  ;st0 y, st1=x
   xor  eax,eax
   mov  al,byte[ebx+lyli.tehe]
   cmp  al,'+'
   jne  m
   faddp st1
   jmp  salv
m:
   cmp  al,'-'
   jne  k
   fsubp st1
   jmp  salv
k:
   cmp  al,'*'
```

```
   jne  j
   fmulp st1
   jmp  salv
j:
   fdivp st1
salv:
   fstp qword[ebx+lyli.a] ;cur->a=res
   push ebx
   call _press  ;resultaadi trykk
   add  esp,4
   mov  eax,ebx  ;return(cur)
aut:
   pop  ebx
   pop  ebp
   ret
;--------------------------------
_press:  ;press(cur) : trykib 8-baidise resultaadi
   push ebp
   mov  ebp,esp
   push ebx
   mov  ebx,dword[ebp+8] ;*cur
   push dword[ebx+lyli.a+4] ;double=>stack
   push dword[ebx+lyli.a]
   push fo  ;'res=%4.2f',10,0
   call _printf
   add  esp,12
   pop  ebx
   pop  ebp
   ret
;--------------------------
```

```
Administrator: cmd - Shortcut

C:\shy>nasm -f win32 felix.asm -o felix.obj

C:\shy>gcc felix.obj jupid.c -o jupid.exe

C:\shy>jupid
C-size=48
lyli_size=45
res=-4.00

C:\shy>
```

Joonis 13.4.3. Struktuuri pikkus C- ja *NASM*-programmis.

Nagu ülaloleval pildil näeme, on C lisanud 1-baidisele väljale *tehe* 3 baiti. Selle ülesande puhul pole sel seigal tähtsust (väljade suhtaadressid on samad), küll aga tuleb assembleris defineeritud struktuur panna C omaga täpselt klappima siis, kui struktuursetest kirjetest koosneva faili kirjutab kettale ühes keeles kirjutatud programm ja loeb kettalt mällu teises keeles kirjutatud programm.

LISA 1. ROGER JEGERLEHNERI KOODITABEL[1] [JEGERLEHNER]

Roger Jegerlehner

1 R. Jegerlehner teatab tabeli joonealuse märkusena, et seda tohib vabalt reprodutseerida ja levitada. Meilt kaks märkust: esiteks, Jegerlehneri tabel on tehtud 16-bitise mudeli jaoks, teiseks: assembleri direktiivide nimed ja semantika on sõltumatud protsessori versioonist.

1. TRANSFER		Code	Operation	Flags								
Name	Comment			O	D	I	T	S	Z	A	P	C
MOV	Move (copy)	MOV Dest,Source	Dest:=Source									
XCHG	Exchange	XCHG Op1,Op2	Op1:=Op2 , Op2:=Op1									
STC	Set Carry	STC	CF:=1									1
CLC	Clear Carry	CLC	CF:=0									0
CMC	Complement Carry	CMC	CF:= –, CF									±
STD	Set Direction	STD	DF:=1 (string op's downwards)		1							
CLD	Clear Direction	CLD	DF:=0 (string op's upwards)		0							
STI	Set Interrupt	STI	IF:=1			1						
CLI	Clear Interrupt	CLI	IF:=0			0						
PUSH	Push onto stack	PUSH Source	DEC SP, [SP]:=Source									
PUSHF	Push flags	PUSHF	O, D, I, T, S, Z, A, P, C 286+: also NT, IOPL									
PUSHA	Push all general registers	PUSHA	AX, CX, DX, BX, SP, BP, SI, DI									
POP	Pop from stack	POP Dest	Dest:=[SP], INC SP									
POPF	Pop flags	POPF	O, D, I, T, S, Z, A, P, C 286+: also NT, IOPL	±	±	±	±	±	±	±	±	±
POPA	Pop all general registers	POPA	DI, SI, BP, SP, BX, DX, CX, AX									
CBW	Convert byte to word	CBW	AX:=AL (signed)									
CWD	Convert word to double	CWD	DX:AX:=AX (signed)	±				±	±	±	±	±
CWDE	Conv word extended double	CWDE 386	EAX:=AX (signed)									
IN *i*	Input	IN Dest, Port	AL/AX/EAX := byte/word/double of specified port									
OUT *i*	Output	OUT Port, Source	Byte/word/double of specified port := AL/AX/EAX									

i for more information see instruction specifications

Flags: ±=affected by this instruction ?=undefined after this instruction

ARITHMETIC		Code	Operation	Flags								
Name	Comment			O	D	I	T	S	Z	A	P	C
ADD	Add	ADD Dest,Source	Dest:=Dest+Source	±				±	±	±	±	±
ADC	Add with Carry	ADC Dest,Source	Dest:=Dest+Source+CF	±				±	±	±	±	±
SUB	Subtract	SUB Dest,Source	Dest:=Dest-Source	±				±	±	±	±	±
SBB	Subtract with borrow	SBB Dest,Source	Dest:=Dest-(Source+CF)	±				±	±	±	±	±
DIV	Divide (unsigned)	DIV Op	Op=byte: AL:=AX / Op AH:=Rest	?				?	?	?	?	?
DIV	Divide (unsigned)	DIV Op	Op=word: AX:=DX:AX / Op DX:=Rest	?				?	?	?	?	?
DIV 386	Divide (unsigned)	DIV Op	Op=doublew.: EAX:=EDX:EAX / Op EDX:=Rest	?				?	?	?	?	?
IDIV	Signed Integer Divide	IDIV Op	Op=byte: AL:=AX / Op AH:=Rest	?				?	?	?	?	?
IDIV	Signed Integer Divide	IDIV Op	Op=word: AX:=DX:AX / Op DX:=Rest	?				?	?	?	?	?
IDIV 386	Signed Integer Divide	IDIV Op	Op=doublew.: EAX:=EDX:EAX / Op EDX:=Rest	?				?	?	?	?	?
MUL	Multiply (unsigned)	MUL Op	Op=byte: AX:=AL*Op if AH=0 ♦	±				?	?	?	?	±
MUL	Multiply (unsigned)	MUL Op	Op=word: DX:AX:=AX*Op if DX=0 ♦	±				?	?	?	?	±
MUL 386	Multiply (unsigned)	MUL Op	Op=double: EDX:EAX:=EAX*Op if EDX=0 ♦	±				?	?	?	?	±
IMUL *i*	Signed Integer Multiply	IMUL Op	Op=byte: AX:=AL*Op if AL sufficient ♦	±				?	?	?	?	±
IMUL	Signed Integer Multiply	IMUL Op	Op=word: DX:AX:=AX*Op if AX sufficient ♦	±				?	?	?	?	±
IMUL 386	Signed Integer Multiply	IMUL Op	Op=double: EDX:EAX:=EAX*Op if EAX sufficient ♦	±				?	?	?	?	±
INC	Increment	INC Op	Op:=Op+1 (Carry not affected !)	±				±	±	±	±	
DEC	Decrement	DEC Op	Op:=Op-1 (Carry not affected !)	±				±	±	±	±	
CMP	Compare	CMP Op1,Op2	Op1-Op2	±				±	±	±	±	±
SAL	Shift arithmetic left (= SHL)	SAL Op,Quantity	-	*i*				±	±	?	±	±
SAR	Shift arithmetic right	SAR Op,Quantity		*i*				±	±	?	±	±
RCL	Rotate left through Carry	RCL Op,Quantity		*i*								±
RCR	Rotate right through Carry	RCR Op,Quantity		*i*								±
ROL	Rotate left	ROL Op,Quantity		*i*								±
ROR	Rotate right	ROR Op,Quantity		*i*								±

i for more information see instruction specifications ♦ then CF:=0, OF:=0 else CF:=1, OF:=1

LOGIC		Code	Operation	Flags								
Name	Comment			O	D	I	T	S	Z	A	P	C
NEG	Negate (two-complement)	NEG Op	Op:=0-Op if Op=0 then CF:=0 else CF:=1	±				±	±	±	±	±
NOT	Invert each bit	NOT Op	Op:=–, Op (invert each bit)									
AND	Logical and	AND Dest,Source	Dest:=Dest∧Source	0				±	±	?	±	0
OR	Logical or	OR Dest,Source	Dest:=Dest∨Source	0				±	±	?	±	0
XOR	Logical exclusive or	XOR Dest,Source	Dest:=Dest (exor) Source	0				±	±	?	±	0
SHL	Shift logical left (= SAL)	SHL Op,Quantity	LSB CF ◄ MSB CF ►	*i*				±	±	?	±	±
SHR	Shift logical right	SHR Op,Quantity	CF ► 0 0 ► CF	*i*				±	±	?	±	±

MISC		Code	Operation	Flags								
Name	Comment			O	D	I	T	S	Z	A	P	C
NOP	No operation	NOP	No operation									
LEA	Load effective address	LEA Dest,Source	Dest := address of Source									
INT	Interrupt	INT Nr	interrupts current program, runs spec. int-program			0	0					

JUMPS (flags remain unchanged)		Code	Operation	Name	Comment	Code	Operation
Name	Comment						
CALL	Call subroutine	CALL Proc		RET	Return from subroutine	RET	
JMP	Jump	JMP Dest					
JE	Jump if Equal	JE Dest	(= JZ)	JNE	Jump if not Equal	JNE Dest	(= JNZ)
JZ	Jump if Zero	JZ Dest	(= JE)	JNZ	Jump if not Zero	JNZ Dest	(= JNE)
JCXZ	Jump if CX Zero	JCXZ Dest		JECXZ	Jump if ECX Zero	JECXZ Dest	386
JP	Jump if Parity (Parity Even)	JP Dest	(= JPE)	JNP	Jump if no Parity (Parity Odd)	JNP Dest	(= JPO)
JPE	Jump if Parity Even	JPE Dest	(= JP)	JPO	Jump if Parity Odd	JPO Dest	(= JNP)

JUMPS Unsigned (Cardinal)				JUMPS Signed (Integer)			
JA	Jump if Above	JA Dest	(= JNBE)	JG	Jump if Greater	JG Dest	(= JNLE)
JAE	Jump if Above or Equal	JAE Dest	(= JNB = JNC)	JGE	Jump if Greater or Equal	JGE Dest	(= JNL)
JB	Jump if Below	JB Dest	(= JNAE = JC)	JL	Jump if Less	JL Dest	(= JNGE)
JBE	Jump if Below or Equal	JBE Dest	(= JNA)	JLE	Jump if Less or Equal	JLE Dest	(= JNG)
JNA	Jump if not Above	JNA Dest	(= JBE)	JNG	Jump if not Greater	JNG Dest	(= JLE)
JNAE	Jump if not Above or Equal	JNAE Dest	(= JB = JC)	JNGE	Jump if not Greater or Equal	JNGE Dest	(= JL)
JNB	Jump if not Below	JNB Dest	(= JAE = JNC)	JNL	Jump if not Less	JNL Dest	(= JGE)
JNBE	Jump if not Below or Equal	JNBE Dest	(= JA)	JNLE	Jump if not Less or Equal	JNLE Dest	(= JG)
JC	Jump if Carry	JC Dest		JO	Jump if Overflow	JO Dest	
JNC	Jump if no Carry	JNC Dest		JNO	Jump if no Overflow	JNO Dest	
				JS	Jump if Sign (= negative)	JS Dest	
				JNS	Jump if no Sign (= positive)	JNS Dest	

LISA 2. X64 LÜHIÜLEVAADE

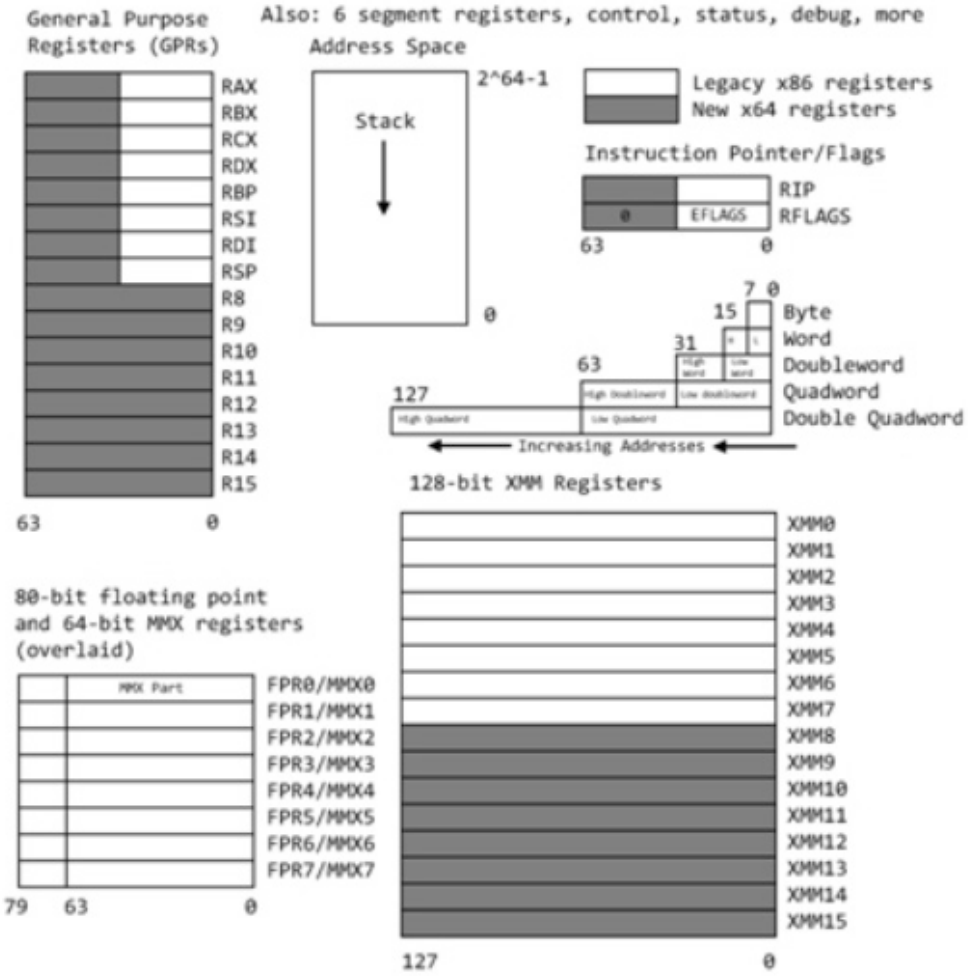
Selles lisas refereerime Chris Lomonti raamatut [Lomont]. x64 on üldmõiste, mida Lomont kasutab Inteli ja AMD 64-bitiste protsessorite arhitektuuri kohta[1]. Meie raamatu jaoks on sellest valdkonnast oluline joonisel L2.a. Enamik asju tõlget ei vaja (registrite nimed), muu teksti võime eesti keelde tõlkida järgmiselt:

- *General Purpose Registers (GPR)* – üldregistrid;
- *Also: 6 segment registers, control, status, debug, more* – lisaks 6 segmentregistrit, juhtregistrid, olekuregistrid, silumisregistrid jm.;
- *Address space* – aadressruum;
- *Instruction Pointer* – käsuviit;
- *Flags* – (signaal)lipud;
- *80-bit Floating point and 64-bit MMX registers (overlaid)* – 80-bitised ujupunkt- ja 64-bitised *MMX*-registrid (ülekattega);
- *Legacy x86 registers* – x86 pärandregistrid[2].

1 Intel kasutab ka nimetust *IA-64*.

2 Tavast ja Hanson [AKS]: *legacy system* – pärandandsüsteem, riistvara, tarkvara, võrk vms, mis jääb käiku pärast uue kasutuselevõttu.

Uued 64-bitised registrid r8...r15 on struktuursed, näit. r8 on 8-baidine, r8d on selle 4 madalamat baiti, r8w – 2 baiti ja r8l – 1 bait (ent r8h-d pole).



Joonis L2.a. x86 ja 64-bitine x64 arhitektuur [Lomont].

MMX-registrid on kasutusel meie x87 kaasprotsessoris, ehkki teiste nimedega (assembler-programm adresseerib neid nimedega *st0...st7*) ja on ujupunktregistrid, *MMX*-tehnoloogia kasutab neid *int*-tüüpi andmete jaoks. Wikipedia andmetel [wMMX] ei tähenda *MMX* midagi muud kui pelgalt nime ja hiljem on seda tõlgendatud näiteks kui *MultiMedia eXtension, Multiple Math eXtension*, või *Matrix Math eXtension*. Nendega opereerimiseks on omaette käsustik (*SIMD*[1] – *Single Instruction Multiple Data*). Arvatavasti me ei eksi,

1 *SIMD architecture* – ühe käsuvoo ja mitme andmevooga arvutiarhitektuur [AKS, lk.191]. Nii

kui nii selle kui ka *SSE* (*Streaming SIMD Extensions*) – tehnoloogia peamine rakendusvaldkond on arvutigraafika, sh. näit. arvutimängud.

Technology	Register size/type	Item type	Items in Parallel
MMX	64 MMX	Integer	8, 4, 2, 1
SSE	64 MMX	Integer	8,4,2,1
SSE	128 XMM	Float	4
SSE2/SSE3/SSSE3…	64 MMX	Integer	2,1
SSE2/SSE3/SSSE3…	128 XMM	Float	2
SSE2/SSE3/SSSE3…	128 XMM	Integer	16,8,4,2,1

Joonis L2.b. Graafikatehnoloogiad [Lomont].

Ujupunkt-kaasprotsessor evib 8 *MMX*-registrit (*MMX* on dešifreeritud kui *"MultiMedia eXtension, Multiple Math eXtension, või Matrix Math eXtension"*), – registrid eeskätt 80-bitiste *int*- väärtuste jaoks: *MMX0 … MMX7*. Need registrid võimaldavad töödelda struktureeritud andmeid *XMM* 128-bitisese formaadi piire erinevate alamformaatidega:

SSE-formaat võimaldab hoida ühes *XMM*-registris nelja 32-bitist ujupunktarvu,

SSE2-formaat aga võib hoida ühes *XMM*-registris kas kaht 64-bitist ujupunktarvu, nelja 32-bitist *int*-arvu, kaheksat 16-bitist või 16 ühebitist *int*-arvu.

XMM-registritega manipuleerimiseks on omaette käsustik ja programmeerimisvõtted; *SSE*-tehnoloogia areneb edasi (täna (2019) on evitatud 256-bitised *YMM*-registrid).

Nood „pikad registrid" on ainult andmete/andmehulkade jaoks ja mitte kunagi mäluaadresside jaoks; lihtsustatult: nende abil saab ühe käsuga teha sama tehte paljude operandidega.

LISA 3. KESKKOND

Meie raamatu *NASM*- ja *C*- näited on läbimängitavad, kui lugejal (kasutajal) on installeeritud vajalik tarkvara. Me anname endale aru, et see valdkond on

SIMD- kui ka erinevad *SSE*- käskude komplektid on x86 lisakomplektid (*extensions*). Nende tuvastamiseks on *CPUID*-käsk.

loomult kiire muutuma – või ka kaduma – aga 2019. aasta suvel on *NASM*-programmeerimiseks vaja paigaldada kaks tarkvara-komplekti: *MinGW-W64* ja *NASM*. *NASM* võimaldab transleerida *.asm*-failist objektfaili, *MinGW* aga sisaldab *GNU* kompilaatorite komplekti *gcc*, mida kasutame *NASM*i objektfaili komplekteerimiseks, saamaks *.exe*-faili.

*NASM*i kõige uuema versiooni saab aadressilt *https://nasm.us*. Valida tuleb vastav *win32* väljalase, näiteks "*nasm-2.15.05-win32.zip*" ning pakkida see lahti sobivasse kausta, näiteks:

```
C:\nasm\nasm.exe
```

Esialgne MinGW projekt on praeguseks hetkeks hüljatud, ning arenduse on üle võtnud haruprojekt *MinGW-W64*. Kõige uuema versiooni saab aadressilt *http://mingw-w64.org/doku.php/download*, kus tuleb valida „*MingGW-W64-builds (Windows)*", mis viitab vastavale „*mingw-w64-install.exe*" installerile.

MinGW-W64 installeri valikutes, valida sobiv *GCC* versioon näiteks 8.1.0. Arhitektuur i686 mis on 32-bit *x86* programmide kompileerimise jaoks. Kusjuures, 64-bit arhitektuuri *x86_64* saab ka lisaks eraldi kausta installeerida, kui on tarvis 64-bit assembleri programme linkida. Valida sobiv paigalduskaust, näiteks "`C:\mingw-w64\`" puhul paigaldatakse kompilaator kausta:

```
C:\mingw-w64\i686-8.1.0-posix-dwarf-rt_v6-rev0\
mingw32\bin\gcc.exe
```

Selleks, et pärast paigaldusi oleksid *nasm* ja *gcc* käsud käsurealt nähtavad, on tarvis avada Windows *PowerShell* (Admin) ja sisestada järgnevad käsud (kaustad tuleb enne muuta vastavalt *NASM* ja *MinGW-W64* asukohtadele):

```
$newPath = "C:\nasm\;C:\mingw-w64\
i686-8.1.0-posix-dwarf-rt_v6-rev0\mingw32\bin\"
$oldPath = [Environment]::GetEnvironmentVari-
able('PATH', 'Machine');
[Environment]::SetEnvironmentVariable('PATH',
"$newPath;$oldPath",'Machine');
```

Mille käivitamisel kantakse jäädavalt kasutaja *PATH* muutujasse markeeritud kaustad. Uute lisatud teede aktiveerimiseks tuleb sulgeda kõik lahtiolevad käsureaaknad. Alternatiivselt saab Windows *Control Panel*'i kaudu ka käsitsi *Environment Variables* muudatusi teha.

Käivitame uuesti käsurealt järgnevad käsud, et paigaldust kontrollida:

```
> where nasm
C:\nasm\nasm.exe
> where gcc
C:\mingw-w64\i686-8.1.0-posix-dwarf-rt_v6-rev0\
mingw32\bin\gcc.exe
```

Kui väljundites pole näha õiget *gcc* kompilaatorit, siis tasub üle vaadata *PATH* muudatused. Arvestada tuleb sellega, et süsteemis võib juba olla mitu erinevat ja mitteühilduvat *gcc* kompilaatorit.

– Jorma Rebane

KASUTATUD MATERJALID

[AKS] Arvi Tavast, Vello Hanson, Arvutikasutaja sõnastik, inglise- eesti, Ilo sõnastik, Tallinn 2003.

[Amrozek] http://home.agh.eTekstis käsitletakse mõistet AVLdu.pl/~amrozek/x87.pdf 19.07.19

[Assemblers] https://en.wikibooks.org/wiki/X86_Assembly/x86_Assemblers 7.04.19

[ateh] http://nasm.ateh10.net/ 28.08.19

[AVL] https://www.coursehero.com/file/p164i55/Adelson-Velski-%C4%B1-and-Landis-1-introduced-in-1962-a-criterion-for-constructing/ 19.06.19
https://en.wikipedia.org/wiki/AVL_tree 19.0d.19

[BIOS] https://www.howtogeek.com/56958/htg-explains-how-uefi-will-replace-the-bios/ 19.04.19

[blogspot] http://arvutikomponendid.blogspot.com/2017/11/arvutite-komponendid-ja-arhitektuur.html 4.08.19

[cache] https://www.quora.com/What-is-the-typical-size-of-cache-memory 3.04.19

[Carter] Paul A. Carter, PC Assembly Language, July 23, 2006. http://pacman128.github.io/static/pcasm-book.pdf 25.05.19

[CD] Computer Dictionary, Microsoft Press, 1991.

[Chemnitz] Encoding x86 Instructions, https://www-user.tu-chemnitz.de/

~heha/viewchm.php/hs/x86.chm/x86.htm 1.01.19

[Chen] Becky Chen, Steps of compiling a C program https://medium.com/@bchen720/steps-of-compiling-a-c-program-7a9a531eb9f8 12.04.19

[CIS-77] http://www.c-jump.com/CIS77/CIS77syllabus.htm 1.01.19

[Chourdakis] Michael Chourdakis, The Real, Protected, Long mode assembly tutorial for PCs, https://www.codeproject.com/Articles/45788/The-Real-Protected-Long-mode-assembly-tutorial-for-PCs 1.01.19

[CP] https://compprog.wordpress.com/2007/12/01/one-source-shortest-path-dijkstras-algorithm/ 21.06.19

[Crt L] https://docs.microsoft.com/en-us/cpp/c-runtime-library/c-run-time-library-reference?view=vs-2019 16.04.19

[cryptowiki] http://cryptowiki.net/index.php?title=Vernam_cipher 25.05.19

[dp] http://compprog.files.wordpress.com/2008/01/dijkstra.c

[Fibonacci] https://www.google.ee/search?q=fibonacci&ie=UTF-8&hl=et 2.06.19
http://www-history.mcs.st-andrews.ac.uk/Biographies/Fibonacci.html 2.06.19

[GAS] https://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax 12.04.19

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

[github] https://github.com/dbohdan/compilers-targeting-c 14.08.19

[idsia] http://people.idsia.ch/~juergen/bauer.html 9.11.17
[int21] http://spike.scu.edu.au/~barry/interrupts.html 20.04.19.

[Isotamm, C] Ain Isotamm, Programmeerimine C-keeles Algoritmide ja andmestruktuuride näidetel, Tartu Ülikool, Matemaatika-informaatikateaduskond, Arvutiteaduse instituut, Tartu 2009.

[Isotamm, PKd] Ain Isotamm, Programmeerimiskeeled, Tartu Ülikool, Matemaatika-informaatikateaduskond, Arvutiteaduse instituut, Tartu 2007.

[Isotamm, TTS] Ain Isotamm, Translaatorite tegemise süsteem, Tartu Ülikool, Matemaatika-informaatikateaduskond, Arvutiteaduse instituut, Tartu 2012.

[Jegerlehner] http://www.jegerlehner.ch/intel/IntelCodeTable.pdf 26.04.19

[Kaasik] Ülo Kaasik, Matemaatikaleksikon, Tallinn, „Valgus“, 1982.

[keskaeg] https://et.wikipedia.org/wiki/Keskaeg 1.06.19

[K&R] Brian W. Kernighan, Dennis M. Ritchie, The C Programming Language, Second Edition (ANSI C), Prentice Hall Software Series. 14.08.19

[Knott] http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibBio.html 23.05.19

[Knuth III] Д. Кнут, Искусстство программирования для ЭВМ, т. 3, Сортировка и поиск, „Мир“, М 1978.

[Lebherz] Eric Lebherz, Computing Language List, http://www.hyperernews.org/HyperNews/get/computig/lang-list.html 07.09.05

[letter] http://letterfrequency.org/letter-frequency-by-language/ 28.05.19

[Lomont] Chris Lomont, Introduction to x64 Assembly, https://software.intel.com/en-us/articles/introduction-to-x64-assembly 23.07.19

[LPP] https://en.wikipedia.org/wiki/Longest_path_problem 23.06.19

[MinGW] http://www.mingw.org/ 28.04.19
http://www.mingw.org/wiki/InstallationHOWTOforMinGW 28.04.19
https://sourceforge.net/projects/mingw-w64/ 28.04.19
https://en.wikipedia.org/wiki/MinGW 28.04.19

[Motorola] http://www.intel-assembler.it/portale/5/motorola-fpu-programming/68881-68882-68040-command-reference.asp 19.07.19

[NASM] https://www.nasm.us/ 14.04.19

[nasm] NASM – The Netwide Assembler, version 2.10.09, © 1996 – 2012 The NASM Development Team.

[PDP] https://history-computer.com/ModernComputer/Electronic/PDP-11.html 8.05.19.

[princetown]http://www.cs.princeton.edu/courses/archive/spr16/cos217/lectures/16_MachineLang.pdf 7.01.19

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

[Ray] https://cs.lmu.edu/~ray/notes/nasmtutorial/ 19.07.19

[Rebane] Jorma Rebane, Makroassembler x86 Nasm, Tartu 2015 (arvutifail)

[scvalex] https://compprog.wordpress.com/2007/12/01/one-source-shortest-path-dijkstras-algorithm/ 23.06.19

[Smith] http://www.science.smith.edu/dftwiki/index.php/CSC231_Floating-Point_Assembly_Examples 19.07.19

[Stallman] https://en.wikipedia.org/wiki/Richard_Stallman 14.08.19

[Streib] James T. Streib, Guide to Assembly Language, A Concise Introduction, Springer. https://books.google.ee/books?id=7arN6Ht59u4C&pg=PA34&lpg=PA34&dq=how+to+use+edx:eax&source=bl&ots=cvaCZMiQuB&sig=ACfU3U0274Rvh1uxVOzxurycs2YQqqSzmw&hl=et&sa=X&ved=2ahUKEwiD9qHcxKXiAhVsposKHa1eC7w4ChDoATABegQICBAB#v=onepage&q=how%20to%20use%20edx%3Aeax&f=false. 19.05.19

[Tatham] https://www.chiark.greenend.org.uk/~sgtatham/ 14.04.19

[TF] Tom Fisher, https://www.lifewire.com/dos-commands-4070427 19.04.19.

[Zuoliu Ding] https://www.codeproject.com/Articles/1116188/Basic-Practices-in-Assembly-Language-Programming 2.06.19

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

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

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

[wcc] https://en.wikipedia.org/wiki/X86_calling_conventions 13.04.19.

[Wirth] Wirth N., Kolmkümmend aastat programmeerimiskeeli ja translaatoreid, tõlkinud Jaan Penjam, A&A, nr.1, 1993, lk. 13– 24.

[wMMX] https://en.wikipedia.org/wiki/MMX_(instruction_set) 1.08.19

[wnasm] https://en.wikipedia.org/wiki/Netwide_Assembler 15.04.19

[wx86] x86 instruction listings, https://en.wikipedia.org/wiki/X86_instruction_listings 8.01.19

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

[x86IS] https://c9x.me/x86/html/file_module_x86_id_5.html 8.01.19

[x87is] http://www.intel-assembler.it/portale/5/The-8087-Instruction-Set/A-one-line-description-of-x87-instructions.asp 8.07.19

[Yale] x86 Assembly Guide, http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html 6.11.17
http://arvutikomponendid.blogspot.com/2017/11/arvutite-komponendid-ja-arhitektuur.html 3.08.17

INDEKS

L

M

N

O

P

R

S

T

U

V

W

X

Y