Prosessoriarkkitehtuurista, konekielestä ja ohjelman suorituksesta

Tämä lehdykkä korvaa kesän 2007 Käyttöjärjestelmät -kurssilla luentomonisteen sivut 6-21. Jos intoa riittää, ei varmasti haittaa lukea sama asia luentomonisteestakin, mutta tämä on nyt pääasiallinen lähestyminen konekieleen ja käyttöjärjestelmän rajapinnan toteutukseen... Tässä on aika paljon enemmän sivuja kuin luentomonisteessa oli. En pidä asiaa pahana; olen materiaaliin ihan tyytyväinen. Mielestäni se kuvailee taustoja ja asioiden liittymäkohtia, joita olisi vähemmillä sanoilla vaikeampi esittää. Jonkin verran on päällekkäisyyttä luentomonisteen myöhempien osioiden kanssa, mutta ei kai liikaa. Ilmoittakaa, jos tässä on jotain virheitä tai epäselvyyksiä... ei niitä itse huomaa...

Tämä 29.6.2007 valmistunut versio on kesäkurssin 2007 mielessä "lopullinen", eli se versio, jonka tulostan ja monistan kurssilaisille. Muuten kuin kesän 2007 mielessä tämä on selvästi kaukana lopullisesta, mutta johonkin on vedettävä raja, kun deadline meni ja kesän tentit lähestyvät...

Sisältö

1 Esitiedot **3**
- 1.1 Tietokoneen rakenne . . . 3
- 1.2 Prosessorin toimenpiteet . . . 3
- 1.3 Väylän rakenne ja toimenpiteet . . . 4
- 1.4 Digitaalisen järjestelmän rajat . . . 5
- 1.5 Erilaisista prosessoreista . . . 5

2 Alustus tähän lehdykkään **7**

3 Prosessorin toiminnasta yleisesti **8**

4 Yleistä assemblereista ja notaatioista **12**

5 Käyttäjän näkemät rekisterit x86-64:ssa **14**

6 Käskykanta x86-64 -arkkitehtuurissa **15**
- 6.1 MOV-käskyt . . . 16
- 6.2 Pinokäskyt . . . 17
- 6.3 Aritmeettisia käskyjä . . . 17
- 6.4 Bittilogiikkaa . . . 18
- 6.5 Muita bittien muokkausoperaatioita . . . 19
- 6.6 Suoritusjärjestyksen ohjaus: mistä on kyse . . . 19
- 6.7 Suoritusjärjestyksen ohjausta: ehdot ja toistot . . . 20
- 6.8 Suoritusjärjestyksen ohjausta: aliohjelmat . . . 21
- 6.9 Suoritusjärjestyksen ohjausta: keskeytyspyyntö . . . 22
- 6.10 Huomio keskeytyskäsittelystä . . . 24

7 Keskeytykset ja lopullisempi visio fetch-execute -syklistä **26**

8 Yksinkertainen esimerkki konekieliohjelmasta **27**

9 Koodi, tieto ja suorituspino; osoittimen käsite **28**

10 Virtuaalimuisti ja osoitteenmuodostus **30**

11 Aliohjelman suoritus (== ohjelman suoritus) **36**
- 11.1 Lyhyesti aliohjelmista ja metodeista . . . 36
- 11.2 Aliohjelma-aktivaatio (eli kutsu) prosessorin toimenpiteenä . . . 36
- 11.3 Moderni laajennos: System V ABIn C-kutsumalli x86-64:lle . . . 39

12 Liite: Entä jos keskeytyksiä ei olisi? **42**

1 Esitiedot

Esitietoina edellytetään ihan vähän tietokonelaitteiston ymmärrystä. Noin 135% tarpeellisesta tietämysmäärästä käsiteltiin kesän alkupään luennoilla. Lähdemateriaalina oli Tietotekniikan Perusteet -moniste, jossa on havainnollisia kuvia. Varsinaisessa kurssitarjonnassa taitaa löytyä myös nimike "digitaalilogiikka", mutta mitään kaikille pakollista laitteistokurssia ei taida tällä hetkellä olla. Jostakin lähteestä sinun tulee olla saavuttanut jonkinlainen ymmärrys seuraavista asioista. Uskokaa huvikseen, että jokaisesta kohdasta olisi olemassa myös sellaista *ihan oikeasti teknistä* tietoa, mutta tässä on vain köykäinen ja "kuvalehtityyppinen", *vähemmän tekninen* lähestyminen...

1.1 Tietokoneen rakenne

Nykyaikainen tapa suunnitella tietokoneen perusrakenne on pysynyt pääpiirteissään samana yli puoli vuosisataa. Rakenteella on nimi: **tietokonearkkitehtuuri**. Joskus sanotaan **Von Neumann -arkkitehtuuri** 1950-luvulla vaikuttaneen henkilön mukaan, vaikka kyseinen John Von Neumann ei keksinyt tietokonetta yksin, vaan perusarkkitehtuuri on monen henkilön pitkän työn tulos. Joitakin huomioita tietokoneesta:

- Digitaalinen laskin (*digital computer*) eli tietokone toimii tietyssä mielessä erittäin yksinkertaisesti. Kaikki niin sanottu "tieto", vielä enemmän "informaatio" tai "informaation käsittely", jota digitaalisella laskimella tehdään, on täysin ihmisen tekemää (ohjelmia ja järjestelmiä luomalla), eikä kone taustalla tarjoa mitään älykkyyttä (vaikka onkin älykkäiden ihmisten luoma sähköinen automaatti).
- Tietokoneen kaikki osat rakennetaan yksinkertaisista elektroniikkakomponenteista koostamalla
- Tietokoneessa on keskusyksikkö, muisti ja I/O-laitteita. Komponentteja yhdistää väylä.

1.2 Prosessorin toimenpiteet

- Tietokone toimii nopean kellopulssin ohjaamana: Aina kun kello "lyö" eli antaa jännitepiikin (esim. noin 3 000 000 000 kertaa sekunnissa), prosessorilla on mahdollisuus aloittaa joku toimenpide. Toimenpide voi kestää useita kellojaksoja; prosessori ei voi kuunnella kelloa uuden toimenpiteen aloittamiseksi ennen kuin se on tehnyt edellisen valmiiksi.
- Jokainen mahdollinen toimenpide, jonka prosessori voi kerrallaan tehdä, on jotakin hyvin yksinkertaista --- tyypillisimmillään vain rajatun bittimäärän (sähköjännitteiden) siirtäminen paikasta toiseen ("piuhoja pitkin"), mahdollisesti soveltaen välillä jotakin yksinkertaista, ennaltamäärättyä digitaaliloogista operaatiota (joka perustuu transistori- diodi- ym. komponenttien yhdistelyyn).
- Nykyaikaisimmillaan ja hienostuneimmillaan prosessori voi suorittaa toimenpiteenä myös jonkin pienoisalgoritmin, joka koostuu muutamista edellisen kohdan kuvaamista perustoimenpiteistä (peräkkäin, edelleenkin rajatusti ja ennaltamäärätysti). Näitä hieman pidempiä toimintoja tarvitaan mm. modernin käyttöjärjestelmän ominaisuuksien mahdollistamiseksi. Tärkeätä on se, että tietyn (väkisin hieman perustoimenpidettä monimutkaisemman) kokonaisuuden suoritus tosiaan on **atominen**, eli uusi toimenpide ei pääse alkamaan ennen kuin edellinen on päättynyt. Nämäkin ovat vielä äärimmäisen yksinkertaisia suhteessa siihen, millaisia 3D-pelejä ja projektinhallintajärjestelmiä tietokoneilla näytetään voivan tehdä tai jopa siihen, miltä ohjelmointi näyttää esimerkiksi ensimmäisellä ohjelmointikurssilla (jos siis aloitetaan korkean tason kielestä eikä konekielisestä ohjelmoinnista, joka olisi myös ihan toimiva, joskin erilainen lähtökohta ohjelmoinnin opiskeluun.)

- Toimenpiteisiin sisältyvät bittisiirrot (mahdollisesti sisältäen digitaaliloogisen operaation kuten yhteenlaskun) voivat tapahtua vain lähteen (yksi tai kaksi kpl) ja kohteen (yksi kpl, voi olla sama kuin jompikumpi lähde) välillä. Lähde ja kohde ovat paikkoja, joissa voi säilyttää bittejä (siis sähköjännitteitä) jonkin aikaa, ja siirto edellyttää elektronien virtaamista paikasta toiseen. Niinpä:
  - Prosessorin toimenpiteeseen kuluvalla ajalla on maailmallisesta syystä aiheutuva alaraja.
  - Kuta enemmän siirrettävää toimenpiteeseen tai atomiseen minialgoritmiin sisältyy, sitä pidempi aika sen suorittamiseen menee.
  - Koska siirto prosessorin ja muiden komponenttien (muisti, I/O) välillä tapahtuu kaikille yhteistä väylää pitkin, joutuu jokainen väyläsiirtoa tarvitseva operaatio odottamaan, että väylä vapautuu.
  - Edelleen, ulkoinen väylä on jo suhteellisen "pitkä piuha" ja siksi jos toimenpiteen lähde tai kohde on väylän takana, kestää siirto paljon pidempään kuin jos sekä lähde että kohde ovat prosessorin sisällä.
  - Näistä syistä perusarkkitehtuuriin on lisätty mm. välimuisteja. Tämä olkoon kuitenkin enemmän jatkokurssien asia; perusrakenne on edelleen sama (prosessori, muisti, I/O-laitteet ja näitä yhdistävä väylä) vaikka kunkin komponentin sisäistä toimintaa on tehostettu erilaisin tavoin.
- Em. toimenpiteet ovat samankaltaisia kaikilla prosessoreilla, mutta niissä on joitakin syvällisempiä ja pinnallisempia eroja. Kunkin prosessorin toimenpiteet kuvataan prosessorivalmistajan toimittamassa manuaalissa, jonka tarkoituksena on antaa riittävä tieto minkä tahansa toteutettavissa olevan virtuaalikonehierarkian pystyttämiseen niitä nimenomaisia piisiruja käyttämällä, joita prosessoritehtaasta pakataan ulos.

1.3 Väylän rakenne ja toimenpiteet

- Prosessori ja muut tietokonearkkitehtuurin komponentit yhdistyvät toisiinsa yhden, kaikkiin komponentteihin liitetyn **väylän** avulla.
- Väylä tarkoittaa rinnakkaisia "piuhoja" elektronisten komponenttien välillä. Kussakin piuhassa voi olla yksi bitti, joten 64 bitin siirtäminen kerrallaan vaatii 64 rinnakkaista sähköjohdinta.
- Väylä voi muuttaa piuhoissa olevia arvoja vain kellopulssin lyödessä -- väylän kello on usein hitaampi kuin prosessorin, joten väylän toimenpiteet ovat harvemmassa. Ne kestävät muutenkin pidempään, koska sähkön on kuljettava pidempi matka ja aikaa kuluu ohjauslogiikan toimenpiteisiin. Operaatiot sujuvat nopeammin, jos väylää tarvitaan niihin harvemmin.
- Piuhakokoelmia tarvitaan useita, että väylän ohjaus ja synkronointi voi toimia, erityisesti ainakin ohjauslinja, osoitelinja ja datalinja. Näillä voi olla kaikilla eri leveys. Esimerkiksi osoitelinjan leveys voisi olla 38 piuhaa (bittiä) ja datalinjan leveys 64 bittiä. Ohjauslinjaan ei montaa piuhaa tarvitse (ohjauslogiikasta ei kerrota tässä sen enempää; sanotaan sen verran että ohjauslinjaa käyttäen täytyy saada laitteistotasolla toteutumaan väylään liitettyjen komponenttien välinen synkronointi ja poissulku)
- Osoiteväylän leveys (bittipiuhojen lukumäärä) määrittelee osoiteavaruuden laajuuden eli montako muistiosoitetta tietokoneessa voi olla
- Dataväylän leveys (bittipiuhojen lukumäärä) määrittelee kuinka paljon tietoa väylä korkeintaan voi siirtää yhden väyläkellojakson aikana.
- Väylän kautta pääsee käsiksi moniin paikkoihin, ja osoiteavaruus jaetaan usein (laitteistotasolla) osiin siten, että tietty osoitteiden joukko tavoittaa ROM-muistin (tehtaalla kiinnitetty), osa RAM-muistin (jota ohjelmat käyttävät), osa I/O-laitteet, osa prosessorin rekisterit.

1.4 Digitaalisen järjestelmän rajat

Digitaalinen järjestelmä (jollainen tietokone on) ottaa vastaan koodatun syötteen ja palauttaa koodatun vasteen; koodauksessa voi olla monia abstraktiotasoja, mutta laite itse voi käsitellä vain ykkösiä ja nollia, jotka ilmenevät sähköjännitteinä. Niinpä mm. konekielikäskyt ja kaikki niiden käsittelemä data ilmenee bittijonoina. Tyypillistä nykyään on, että kahdeksan peräkkäisen, toisiinsa kuuluvan bitin ryhmää sanotaan **tavuksi**. Eli peräkkäiset 64 bittiä muodostaisivat kahdeksan tavun kokonaisuuden. Joskus 16-bittistä kokonaisuutta sanotaan kahdesta tavusta koostuvaksi **sanaksi** (word) ja sitten puhutaan 32-bittisistä tuplasanoista ja 64-bittinen kasitavu olisi nelisana "quadword". Muisti organisoidaan usein siten, että yksi **muistipaikka**, johon väylän osoitelinjalle sijoitettava yksikäsitteinen **muistiosoite** viittaa, voi sisältää yhden tavun verran biteiksi koodattua dataa. Muistiosoitteet ovat peräkkäisiä; voisivat olla esimerkiksi 0, 1, 2, 3, ... ja niin edelleen. Peräkkäisiin muistipaikkoihin lienee loogista sijoittaa toisiinsa läheisesti liittyvää dataa. Esimerkiksi suomenkielinen lause voidaan koodata tietokoneeseen siten, että

- sovitaan jokaiselle aakkoselle jokin tavun mittainen koodi (esimerkiksi A==15, B==78, C==88, k==91 ja niin edelleen)
- sijoitetaan lauseen, vaikkapa `"olen äidinkielinen"`, peräkkäiset merkit peräkkäisiin muistipaikkoihin siten, että jokainen merkki koodataan sovitulla numerolla. Sovitaan, että merkit ovat peräkkäisissä muistipaikoissa ja että aina ykkösen verran suuremmasta osoitteesta löytyy seuraava merkki oikealle päin lukien. (Tietokone ei ota mihinkään tällaiseen kantaa, vaan ohjelmien tekijä. Yhtä hyvin voitaisiin sopia, että merkit ovat kymmenen muistipaikan välein eikä peräkkäin ... jos se nyt jonkun sovelluksen kannalta olisi järkevä koodaustapa... tai että jonoa luettaisiin isommasta osoitteesta pienempään päin, jolloin seuraava merkki oikealle olisikin pienemmässä muistiosoitteessa...)

Tässä jää vielä hieman avoimia kysymyksiä: Miten kukaan muu ymmärtää tuollaista ihan itse koodattua dataa? Täytyisi ilmeisesti standardoida merkit jotenkin (mikä voi joskus olla ääkkösten kannalta hankalaa vielä näinä varhaisina Unicode-aikoinakin). Entä teknisen toteutuksen kannalta vaikkapa merkkijonon pituuden tunnistaminen: jos on vain numeroita muistissa... niin mistä mikäkin merkkijono alkaa ja mihin se päättyy? Jo merkkijonon koodaaminen tietokoneelle johtaa siihen, että pitää alkaa rakentamaan virtuaalikonehierarkioita: aliohjelma- tai olioluokkakokoelmia sekä sovellusohjelmakirjastoja, jotka helpottavat aina alemman tason rajapinnan käyttöä. Prosessori, muisti ja muu laitteisto on se kaikkein matalin. Esim. merkkijonojen helppo (ja tietoturvallinen) käsittely on leimallista oikeastaan vasta varsin korkealla abstraktiotasolla oleville virtuaalikoneille. Esim. C-kieli ei tarjoa merkkijonoille mitään käsittelyä vaan on käytettävä jotakin aliohjelmakirjastoa. Niin sanotut C-standardikirjastot tarjoavat jotakin, mutta eivät esim. tietoturvallista syöttöä.

Huomattavaa ja tärkeää on:

- Tässä on kaikki; sen mystisempiä asioita tietokone ei tee. Kaikki hieno syntyy ohjelmista ja ohjelmakokoelmista.
- Sovellusohjelmoija ei kirjoita bittijonoja, vaan lähdekoodia jollakin ohjelmointikielellä. *Käytännössä aina* täytyy olla apuohjelma, joka kääntää lähdekoodin laitteiston ymmärtämäksi koodiksi. (Huomaa silti, että noin periaatteessa tottakai *voisi* tehdä suoritettavan bittijonon käsinkin vaikkei se nykypäivänä järkevää olisikaan; kaikki tarvittava tieto kyllä löytyy prosessorin manuaalista).
- Laitteisto asettaa rajoitteet sille, millaisia ohjelmia voi olla: mitä niillä voi tehdä, ja kauanko tekeminen kestää.

1.5 Erilaisista prosessoreista

Tietty **prosessoriarkkitehtuuri** tarkoittaa niitä tapoja, joilla sen mukaisesti rakennettu fyysinen prosessorilaite toimisi: Mitä toimenpiteitä se voi tehdä, mistä ja mihin mikäkin toimenpide voi siirtää bittejä, ja miten mikäkin toimenpide muunnetaan korkeintaan muutaman tavun mittaiseksi bittijonoksi

(eli operaatiokoodiksi, *opcode*, sekä tiedoksi operandeista). Ohjelmat käskevät prosessoria konekielikäskyillä, jotka saavat aikaan aina tietyn toimenpiteen tietyille operandeille (lähde ja kohde). Prosessoriarkkitehtuurissa kuvataan mahdollisten, prosessorin ymmärtämien käskyjen ja operandiyhdistelmien joukko. Tätä kutsutaan suomeksi nimellä **käskykanta** (engl. *instruction set*).

Prosessoriarkkitehtuureita ja niitä toteuttavia fyysisiä prosessorilaitteita on markkinoilla monta, ja niissä on merkittäviä eroja, mutta kaikissa on jollakin tavoin toteutettu pakolliset piirteet:

- jokin joukko jonkinlaisia **rekisterejä**, nopeita muistipiirejä, joissa voi hetken aikaa pitää muutamia tavuja, jotka kuvaavat mm. muistiosoitteita, ohjelmakoodin palasia tai dataa
- jokin joukko mahdollisia konekielisiä käskyjä (eli **käskykanta**) ja jokin joukko sääntöjä, joiden mukaisesti konekieliohjelma pitää tehdä (operaatiokoodien muodostaminen, sallitut muistinosoitusmuodot, aliohjelmien kutsumiskäytännöt, ...).

2 Alustus tähän lehdykkään

Nyt on esitiedot "kerrattu" jollain tasolla myös kesäkurssin kirjallisessa osuudessa. Tarkempi tietämys on hankittava oma-aloitteisesti tai tietotekniikan laiteläheisillä kursseilla (muistaakseni nimeltään "digitaalilogiikka", "laiteläheinen ohjelmointi" ym.)

Tässä materiaalissa koetetaan esitellä ja valaista prosessorien ja konekielisen ohjelmoinnin yleistä käsitteistöä. Käytännön esimerkkinä sattuu olemaan x86-64, koska THK:ssa sattuu pyörimään sellainen nimeltä Jalava. Yhtä hyvin esimerkkinä voisi olla mikä tahansa, jolla olisi helppo pyöräytellä esimerkkejä. x86-64 -arkkitehtuuri on luentomonisteessa esitellyn Intel 8086 -arkkitehtuurin suora perillinen. 8086-konekieliset ohjelmat toimivat muuttamattomina x86-64 -koneissa, vaikka välissä on ollut useita prosessorisukupolvia teknisine harppauksineen. Huomatkaa, että tosiaan Tietohallintokeskuksen kone `jalava.cc.jyu.fi`, jossa tehdään kurssin harjoituksia (ja harjoitustyö), on malliltaan neliytiminen Intel Xeon, jonka arkkitehtuuri on nimenomaan x86-64. Puolestaan IT-väen työ- ja opiskelukoneeksi suunnattu `charra.it.jyu.fi` on kaksiytiminen AMD Opteron, myös x86-64. Materiaalin päivitys tällä tavoin mahdollistaa uskoakseni paremmat mahdollisuudet toteuttaa käytännön esimerkit tavalla, jonka ensinnäkin kesäopettaja osaa tehdä, ja joka antaa tulevaisuutta ajatellen teorian lisäksi myös käytännön kädentaitoja.

Hieman x86-64:n taustaa: Prosessoriteknologiaan keskittyvä yritys nimeltä Intel on julkaissut mm. toisiaan seuranneet prosessorimallit (ja arkkitehtuurit) nimeltä 8086, 80186, 80286, 80386, 80486 ja Pentium. Intel itse on luonut sittemmin merkittävällä tavoin erilaisen prosessoriarkkitehtuurin nimeltä IA-64, jonka ei voi sanoa enää olevan suora perillinen edellisistä. On mielenkiintoista, että pitkäaikainen kilpailija ja "klooniprosessoreja" valmistanut AMD onkin ensimmäisenä esitellyt arkkitehtuurin, joka perustuu vanhaan Intel-jatkumoon, mutta tuo uusia ominaisuuksia niin paljon, että on pystynyt kilpailemaan markkinoilla Intelin omaa erilaista uutuutta vastaan. Tällä kertaa Intel onkin "kloonannut" AMD64-arkkitehtuurin nimikkeellä Intel 64, ja valmistaa prosessoreja, joissa AMD64:lle käännetty konekieli toimii lähes identtisesti. Koska Intel 64 ja "aito ja alkuperäinen" AMD64 ovat lähes samanlaisia, niille on muodostunut yhteisnimi `x86-64`, joka kuvaa periytymistä x86-sarjasta ja leimallista 64-bittisyyttä (eli sitä, että rekistereissä ja ulkoisen väylän datalinjalla on 64 bittiä rivissä). Joitakin eroja on, mutta lähinnä niillä on merkitystä yhteensopivien kääntäjien valmistajille. Käytettäköön jatkossa siis arkkitehtuurien yhteisnimeä x86-64. Muista erilaisista nykyisistä prosessoriarkkitehtuureista mainittakoon ainakin IBM Cell (mm. Playstation 3:n multimediamylly) sekä ARM-sarjan prosessorit (jollainen löytyy monista sulautetuista järjestelmistä kuten kännyköistä).

Tämän kirjoittamiseen on käytetty lähteenä seuraavaa dokumentaatiota:

- AMD64 Architecture home page:
  http://www.amd.com/us-en/Processors/DevelopWithAMD/0,,30_2252_875_7044,00.html
- Intel 64 Architecture home page:
  http://www.intel.com/technology/intel64/
- System V Application Binary Interface, AMD64 supplement (v 0.98):
  http://www.x86-64.org/documentation/abi-0.98.pdf
- Sekalaisia, jo unohtuneita nettilähteitä mm. hakusanoilla "assembler syntax Intel AT&T", "GNU assembler".

3 Prosessorin toiminnasta yleisesti

Ennen kuin tarvitsee (tai edes voidaan) mennä esimerkkitoteutukseen, pitää hieman kuvailla universaaleja ominaisuuksia ja toimintatapoja, jotka prosessoreihin liittyvät. Tässä luvussa kuvaillaan prosessorin eri toimintatiloja ja sitä, miten ohjelmakoodin suoritus kaikissa prosessoreissa etenee.

Moderneissa koneissa on vanhoihin tai ikivanhoihin verrattuna paljon ominaisuuksia; mm. rekisterejä eri tarkoituksia varten on paljon. Huomattakoon, että esimerkiksi käskyrekisteri (joka tunnetaan kirjallisuudessa nimellä `INSTR`, `IR` tai vastaavalla koodinimellä) on esimerkki **ohjelmoijalle näkymättömästä rekisteristä**. Ohjelmoija ei voi mitenkään tehdä koodia, joka vaikuttaisi suoraan tällaiseen näkymättömään rekisteriin -- sellaisia konekielikäskyjä kun ei yksinkertaisesti ole. Prosessori käyttää näitä rekistereitä sisäiseen toimintaansa. (Niiden olemassaolo ja tarkoitus on hyvä tietää, mutta aihetta ei käsitellä tällä kurssilla enää sen jälkeen, kun "fetch-execute -sykli" ja keskeytykset on käyty läpi.) Jos asia meni luennolla ohi, muistetaan, että `INSTR` on jokaisessa prosessorissa, ja sen tehtävä on ottaa väylän kautta vastaan seuraavan konekielikäskyn bittijono ja tallentaa se väliaikaisesti siihen asti, kun kontrolliyksikkö on valmis suorittamaan sen.

Seuraavaksi suoritusvuoroon tarkoitetun käskyn muistiosoite on yhdessä ohjelmoijalle näkyvässä rekisterissä, jonka nimi on **ohjelmalaskuri** (`PC`, *program counter*), **käskyosoitin** (`IP`, *instruction pointer*) **käskyosoiterekisteri**, tai vastaavaa. Käytetään tästedes nimeä `IP`, koska luentomonisteemme käyttää sitä, suomen kielellä siis käskyosoiterekisteri. Noudetun käskyn suoritus tapahtuu sitten kun kaikki operandit ovat rekistereissä tai väylällä valmiina. Suoritus kestää muutamia kellojaksoja, käskyn monimutkaisuudesta riippuen ehkä hyvinkin monta. Suorituksen tuloksena väylään kytketyn muistiosoitteen osoittaman muistipaikan sisältö voi muuttua, jos kohdeoperandi oli muistipaikka (tai jos käsky käyttää pinomuistia). Rekistereiden sisällöt voivat muuttua, ja ainakin käskyosoiterekisteri `IP` päivittyy seuraavan käskyn muistiosoitteeksi, jotta prosessori osaa noutaa sen `IP`:n osoittamasta paikasta, ja sykli voi jatkua. Tätä nouto-suoritus -sykliä käsitellään kohta tarkemmin, ja siihen lisätään vielä lopulta ns. keskeytyskäsittely. Perusmuodossaan, ilman keskeytyskäsittelyä, "fetch-execute" tarkoittaa tätä:

1. Prosessori **noutaa** (*fetch*) konekielikäskyn tavut `IP`-rekisterin osoittamasta paikasta. `IP`:n sisältämät bitit ovat siis muistiosoite, jonka perusteella väylälle saadaan siirretyksi tietyssä keskusmuistin kohdassa olevat bitit, joihin on koodattu yksi konekielinen käsky.
2. Prosessori **suorittaa** (*execute*) juuri noudetun käskyn.

   Eli konekielisen käskyn bittijono herättää kontrolliyksikössä atomisen toimenpiteen, johon syötetään myös muiden käskyssä tarvittavien rekisterien tai muistipaikkojen arvoja.
3. Käskyn suorituksen tuloksena rekisterien tila on muuttunut tietyin tavoin.

   Yksi, joka aina muuttuu, on `IP`. Miten `IP` muuttuu, riippuu suoritetusta käskystä:

   - Laskutoimitus, datan siirto tai muu *peräkkäissuoritus* ==> `IP`:ssä on juuri suoritettua käskyä seuraavan käskyn muistiosoite.
   - *Ehdoton hyppykäsky* ==> `IP`:n sisällöksi on ladattu juuri suoritetun käskyn yhteydessä kerrottu uusi muistiosoite, esim. silmukan ensimmäinen käsky tms.
   - *Ehdollinen hyppykäsky* ==> `IP`:n sisällöksi on ladattu käskyssä kerrottu uusi osoite, mikäli käskyssä kerrottu ehto toteutuu; muutoin `IP` osoittaa seuraavaan käskyyn samoin kuin peräkkäissuorituksessa. (Ehto tulkitaan koodatuksi FLAGS-lippurekisterin johonkin/joihinkin bitteihin.)
   - *Aliohjelmakutsu* ==> `IP`:n sisältönä on käskyssä kerrottu uusi osoite (jonka tulee olla kutsuttavan aliohjelman ensimmäinen käsky; aliohjelmakutsussa prosessori tekee muutakin, mitä käsitellään kohta tarkemmin)

- *Paluu aliohjelmasta* ==> `IP` osoittaa taas siihen ohjelmaan, joka suoritti kutsun, erityisesti kyseessä on aliohjelmakutsua välittömästi seuraava käsky. (kohtapuoleen nähdään, miten prosessori noutaa paluuosoitteen pinomuistista)

Aliohjelmakutsu ja paluu ovat normaalia käskyä hieman monipuolisempia toimenpiteitä, joissa prosessori käyttää myös pinomuistia (tarkennus tulee kohtapuoleen, ja myöhemmin nähdään aliohjelmakutsun ja keskeytyskäsittelyn samankaltaisuus).

Prosessorien manuaaleissa suoritussykli kuvataan laajemmin. Siinä on mukana nykyaikaisia hienouksia, kuten käskyn muuntaminen ns. mikrokoodiksi eli *decode*-vaihe. (Puhutaankin myös *fetch-decode-execute* -syklistä). Lisäksi on käskyjen ennakkonoutoa (*pre-fetch*), todennäköisen suoritusjärjestyksen ennalta-arvailua, rinnakkaisia liukuhihnoja (*pipeline*) sekä välimuisteihin liittyviä asioita. Nämä toimintaa tehostavat toteutusyksityiskohdat ovat Käyttöjärjestelmät -aihepiirin ulkopuolella, mutta mainittakoon ne nimeltä, etteivät tule yllätyksenä. Jos prosessorimanuaalien kaavioista etsii ylläolevaa perusmallia, se kyllä löytyy sieltä osana. Sovellusohjelmoijan (ja jopa käyttöjärjestelmän tekijän) näkökulmasta nykyiset prosessorit näyttävät ylläkuvatulta, joten vedetään näköpiirimme raja tällä kertaa siihen.

Siitä asti, kun käyttöjärjestelmien kehitys pääsi vauhtiin (jo kauan sitten), prosessoreihin on tullut teknisiä ominaisuuksia nimenomaan käyttöjärjestelmien toteuttamista ajatellen. Yksi suuri tarve on eriyttää normaalit käyttäjän ohjelmat omaan "karsinaansa", jotta ne eivät vahingossakaan sotke toisiaan tai järjestelmää. Tätä tarkoitusta varten prosessorissa on **vain käyttöjärjestelmälle näkyviä rekistereitä** (*system registers*) sekä laitteistotasolla toteutettuja toimintoja, joihin pääsee käsiksi vain käyttöjärjestelmän suoritettavissa olevilla konekielikäskyillä. Käyttäjän ohjelmat saavat sisältää vain sellaisia konekielikäskyjä, jotka käsittelevät **käyttäjän nähtävissä olevia rekistereitä** (*user-visible registers*). Koska sama prosessorilaite suorittaa sekä käyttäjän ohjelmia että käyttöjärjestelmäohjelmaa, joilla on eri valtuudet, täytyy prosessorin voida olla ainakin kahdessa eri toimintatilassa, ja tilan on voitava vaihtua tarpeen mukaan.

Prosessori käynnistyy aina niin sanottuun **käyttöjärjestelmätilaan** (*kernel mode*); toinen nimi tälle olisi suomeksi kai "**todellinen tila**" (engl. *real mode*). Käynnistyksen jälkeen prosessori alkaa suorittaa initialisointiohjelmaa ROM-muistista (kiinteästi asetetusta fyysisestä muistiosoitteesta alkaen). Oletuksena on, että ROM:issa oleva, yleensä pienehkö ohjelma lataa varsinaisen käyttöjärjestelmän joltakin ulkoiselta tallennuslaitteelta. Olet ehkä huomannut, että kotitietokoneiden ROM:issa on yleensä BIOS-asetusten säätöohjelmisto, jolla käynnistyksen yhteydessä voi määrätä fyysisen laitteen, jolta käyttöjärjestelmä pitäisi koettaa löytää (korppu, DVD, CD-ROM, kovalevyt, USB-tikku jne...). BIOS tarjoaa myös muita asetuksia, jotka säilyvät virran katkaisun jälkeen (jos tarkoitusta palvelevassa paristossa on virtaa). Käynnistettäessä tietokone siis on vain tietokone, eikä esim. "Mac OS-X, Windows tai Linux -kone".

Käyttöjärjestelmän latausohjelmaa etsitään hyvin alkeellisilla, standardoiduilla laiteohjauskomennoilla tietystä paikasta fyysistä tallennetta -- puhutaan "käynnistyssektorista" (*boot sector*). Siellä pitäisi olla siis nimenomaiselle prosessorille käännetty konekielinen latausohjelma, jolla on sitten vapaus säädellä kaikkia prosessorin systeemitoimintoja ja toimintatiloja. Sen pitäisi myös alustaa tietokoneen fyysinen muisti tarkoituksenmukaisella tavalla, ladata muistiin tarvittavat ohjelmistot, tehdä koko liuta muitakin valmisteluja sekä vielä lopulta tarjota käyttäjille mahdollisuus kirjautua sisään koneelle ja alkaa suorittamaan hyödyllisiä tai viihteellisiä ATK-sovelluksia. Esimerkiksi Unix-käyttöjärjestelmä jää käynnistyttyään odottamaan "loginia" eli käyttäjätunnuksen ja salasanan syöttöä päätteeltä, minkä jälkeen tunnistetulle käyttäjälle käynnistetään vaikkapa demoissa tutuksi tuleva `tcsh`-shell (tai `bash`, `ksh`, tms., valinnan mukaan). Käyttöjärjestelmä ohjaa päätteen näppäinsyötteet shellille ja shellin printtitulosteet päätteelle. Käyttöliittymä voi toki olla graafinenkin, jolloin puhutaan ikkunointijärjestelmästä. Ikkunointi voi olla osa käyttöjärjestelmää kuten Windowsissa, tai se voi olla erillinen ohjelmisto, kuten Unix-ympäristöissä usein käytetty ikkunointijärjestelmä nimeltä X.

Kirjautumisen jälkeen kaikki käyttäjän ohjelmat toimivat **suojatussa tilassa** (*protected mode*) jolle käytetään myös nimeä **käyttäjätila** (*user mode*). Ensiksi mainittu nimi viitannee siihen, että osa prosessorin toiminnoista on suojattu vahingossa tai pahantahtoisesti tapahtuvaa väärinkäyttöä vastaan.

"Käyttäjätila" lienee vastakohta "käyttöjärjestelmätilalle". Prosessorin tilaa (käyttäjä/käyttöjärjestelmätila) säilytetään (kuten arvannetkin) jossakin yhden bitin kokoisessa sähkökomponentissa prosessorin sisällä. Tämä tila (esim. 0==käyttöjärjestelmä, 1==käyttäjätila) löytyy useimmiten **lippurekisteristä** eli **tilarekisteristä** (kirjallisuudessa esim. `FLG`, `FLGS`, `FLAGS`, `PSW` eli *Processor Status Word*, tai vastaavaa). Käytettäköön tässä luentomonisteen mukaisesti nimeä PSW.

PSW tallentaa myös muut prosessorin tilaan liittyvät on/off -liputukset. Prosessoriarkkitehtuurin määritelmä kertoo, miten mikäkin käsky muuttaa PSW:tä. Kolme tyypillistä esimerkkiä:

- Yhteenlaskussa (bittilukujen "alekkain laskeminen") voi jäädä muistibitti yli, jolloin nostetaan "carry flag" lippu -- se on tietty bitti PSW-rekisterissä, ja sen nimi on usein kirjallisuudessa `CF`
- Vähennyslaskussa ja vertailussa (joka on olennaisesti vähennyslasku ilman tuloksen tallentamista!) päivittyy PSW:ssä bitti, joka kertoo, onko tulos negatiivinen -- nimi on usein "negative flag", `NF` (tai jotain...)
- Jos jonkun operaation tulos on nolla (tai halutaan koodata joku tilanne vastaavasti) asettuu "zero flag", nimenä usein `ZF`.

Liput ovat mukana prosessorin syötteessä aina kunkin käskyn suorituksessa, ja suoritus on monesti erilainen lippujen arvoista riippuen. Monet ohjelmointirakenteet, kuten ehtolauseet ja toistorakenteet perustuvat jonkun testikäskyn suorittamiseen, ja vaikkapa ehdollisen hyppykäskyn suorittamiseen (hyppy tehdään täsmälleen silloin kun tietty bitti PSW:ssä on asetettu). Käyttöjärjestelmälle varatut prosessoriominaisuudet eivät ole käytettävissä silloin kun PSW:n käyttäjätilalippu ei niitä salli. Vakiintunut termi on "käyttäjämaa" ("*userland*"), jossa vallitsevat erilaiset pelisäännöt kuin käyttöjärjestelmätilassa.

Nykyaikaisissa prosessoreissa on myös muita käyttäjän tai käyttöjärjestelmän vaihdeltavissa olevia toimintatiloja, jotka vaikuttavat esimerkiksi siihen, miten suurta osaa rekisterien biteistä käytetään operaatioihin, ollaanko jossakin taaksepäin-yhteensopivuustilassa tai vastaavassa, ja sen sellaista, mutta niihin ei ole mahdollisuutta eikä tarvetta syventyä tällä kurssilla. *Olennaista on ymmärtää käyttöjärjestelmätilan ja käyttäjätilan erilaisuus* fyysisen laitteen tasolla. Siihen perustuu moniajo, virtuaalimuistin käyttö ja suuri osa tietoturvasta.

Prosessori on fyysiseltä kooltaan pieni -- sen pitää olla pieni, koska sähköisen signaalin nopeus on rajoitettu, ja mitä pidempiä matkoja sähkön pitää matkata, sen hitaampaa on toiminta. Pienen pieni prosessori sijoitetaan tyypillisesti suhteellisen pieneen koteloon, joka voidaan lämpöä johtavasta kohdasta yhdistää jäähdytysjärjestelmään (vaikkapa tuuletin ja metallisiili). Nykyinen prosessori kuumenee niin paljon, että ilman jäähdytystä se menisi lähes välittömästi rikki. Pieni kotelo on kiinni isommassa kohtelossa, joka on kätevä asentaa kiinni muuhun laitteistoon. Koteloinnissaan olevan prosessorin kommunikaatio muun laitteiston kanssa voi tapahtua vain sähköjohtimia pitkin, joten johtavasta materiaalista on tehty "piuhat" pienen kotelon sisältä suuremman kotelon ulkopuolelle. Suuremmassa kotelossa jokainen piuha ilmenee kuparisena nastana, joka voidaan liittää emolevyyn. Nastojen sijoittelulle on standardeja, jotta eri prosessorivalmistajat voivat koteloida prosessorinsa yhteensopivasti muiden laitteisto-osien valmistajia varten. Luentomonisteessa sivulla 7 luetellaan 8086-prosessorin nastojen merkityksiä. Modernimmissa prosessoreissa, kuten AMD:n Opteronissa, on nastoja enemmän, ja fyysinen sijoittelu riippuu käytetystä standardista, mutta merkitykset ovat prosessorimerkistä riippumatta useimmiten samat:

- dataväylän jokainen bitti yhdistyy yhteen nastaan
- osoiteväylän jokainen bitti yhdistyy yhteen nastaan
- väylän ohjauksessa käytetyt bitit tarvittavassa määrässä nastoja
- kellopulssi
- keskeytysilmoitukset
- maadoitus

Väylä, väylän ohjaus, keskusmuisti ja kaikki I/O -laitteet tosiaan ovat prosessorin koteloinnin ulkopuolella, jos puhutaan pöytäkoneista tai servereistä. Sulautettuja järjestelmiä varten voidaan prosessoreita valmistaa myös "*system-on-a-chip*" -periaatteella, jolloin koko tietokonearkkitehtuurin toteutus

väylineen päivineen, usein myös ROM-ohjelmistolla varustettuna, prässätään yhdelle sirulle. Tällainen sirusysteemi voidaan suunnitella esim. tiettyä kännykkämallia, MP3-soitinta tai digikameraa varten. Ohjelman suorituksen kannalta kaikki näyttää kuitenkin samalta, olipa prosessori koteloitu erikseen tai yhdessä muiden laitteiden kanssa.

4 Yleistä assemblereista ja notaatioista

Nykyään konekielen bittijonoa on järkevää tuottaa vain kääntäjäohjelman avulla. (Toisenlaista oli esihistorian alussa, kun käännös todella tehtiin käsin ja bitit rei'itettiin lävistimellä reikäkorteille -- kielijärjestelmille ja automaattisille kääntäjäohjelmille on aina ollut varsin ymmärrettävä tarve, ja niitä on ollut olemassa lähes yhtä pitkään kuin tietokoneita.) Sovellusohjelmoija pääsee lähimmäksi todellista konekieltä käyttämällä ns. **symbolista konekieltä** eli "assemblyä"/"assembleria", joka käännetään bittijonoksi **assemblerilla** eli symbolisen konekielen kääntäjällä. Jokainen assemblerkielinen rivi kääntyy yhdeksi konekieliseksi käskyksi, ja jokaisella eri prosessorilla on erilainen, oman käskykantansa mukainen assembler. Käyttöjärjestelmistä (pienehkö) osa on kirjoitettava assemblerilla, joten tällä kurssilla ilmeisesti käsitellään sitä. Se on myös oiva apuväline prosessorin toiminnan ymmärtämiseksi (ja yleisemmin ohjelman suorituksen ymmärtämiseksi... ja myös korkeamman abstraktiotason kielijärjestelmien arvostamiseksi!). Assembler-koodin rivi voi näyttää päällisin puolin tältä:

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

Kyseinen rivi voisi hyvin olla x86-64 -arkkitehtuurin mukaista, joskin yhden rivin perusteella olisi vaikea vetää lopullista johtopäätöstä. Erot joissain yksittäisissä assembler-käskyissä ovat arkkitehtuurien välillä olemattomia. Prosessorivalmistajan julkaisema arkkitehtuuridokumentaatio on yleensä se, joka määrittelee symbolisessa konekielessä käytetyt sanat. Jokaisella konekielikäskyllä on **käskysymboli** (vai miten sen suomentaisi, ehkä "muistike" tjsp., englanniksi kun se on *mnemonic*). Yllä olevan esimerkin tapauksessa symboli on `movq`. Käskyn symboli on tyypillisesti jonkinlainen helpohkosti muistettava lyhenne sen merkityksestä. Jos tämä olisi x86-64 -arkkitehtuurin käsky, `movq` (joka AMD64:n manuaalissa kirjoitetaan isoilla kirjaimilla `MOV` ilman `q`-lisuketta) olisi lyhenne sanoista "Move quadword". Sen merkitys olisi siirtää "nelisana" eli 64 bittiä paikasta toiseen. Tieto siitä, mistä mihin siirretään, annetaan **operandeina**, jotka tässä tapauksessa näyttäisivät x86-64:n määrittelemiltä rekistereiltä `rsp` ja `rbp` (AMD64:n dokumentaatiossa isoilla kirjaimilla `RSP` ja `RBP`). Käskyillä on useimmiten nolla, yksi tai kaksi operandia. Joka tapauksessa osa käskyn suorituksen syötteistä voi tulla muualtakin kuin operandeina ilmoitetusta paikasta -- esim. PSW:n biteistä, tietyistä rekistereistä, tai jostain tietystä muistiosoitteesta. Jos operandina on rekisteri, jossa on muistiosoite, ja käskyn halutaan vaikuttavan muistipaikan sisältöön, puhutaan epäsuorasta osoittamisesta, (*indirect addressing*). Tietyn prosessoriarkkitehtuurin dokumentaation käskykanta-osuudessa kerrotaan aina hyvin täsmällisesti, mitkä kunkin käskyn kaikki syötteet, tulosteet ja sivuvaikutukset prosessorin tai keskusmuistin seuraavaan tilaan ovat. Esimerkin tapauksessa nuo 64 bittiä siirrettäisiin rekisteristä `rsp` rekisteriin `rbp`. Sanotaan, että käskyn **lähde** on rekisteri `rsp` ja **kohde** on rekisteri `rbp`. Koska siirto on rekisterien välillä, ulkoista väylää ei tarvitse käyttää. Siirtokäskyllä ei ole vaikutusta lippurekisteriin. Rekistereiden välinen siirto ei voi myöskään aiheuttaa esim. muistinsuojaukseen liittyvää poikkeusta.

Prosenttimerkki `%` ylläolevassa on riippumaton x86-64:stä; se on osa tässä käytettyä yleisempää assembler-syntaksia, jota kurssillamme tänä kesänä käytettävät GNU-työkalut noudattavat.

Jotta ohjelmoijan maailma olisi tehty vaikeammaksi (tai muista historiallisista syistä) noudattavat jotkut assembler-työkalut ihan erilaista syntaksia kuin GNU-työkalut (GNU-työkalujen syntaksi on nimeltään "AT&T -syntaksi" ja se toinen taas "Intel -syntaksi"). Ylläoleva rivi olisi siinä toisessa syntaksissa jotakuinkin näin:

```
movq    rbp, rsp
```

Erittäin merkittävä ero edelliseen on se, että **operandit ovat eri järjestyksessä!!** Eli lähde onkin oikealla ja kohde vasemmalla puolen pilkkua. Jonkun muinoisen insinöörin mukaan kai asiat olivat loogisempia näin, että siirretään "johonkin jotakin" ja jonkun toisen mielestä taas niin, että siirretään "jotakin johonkin". Tai sitten jommallekummalle oli kätevämpi toteuttaa kääntäjä jollekin muinoiselle prosessoriarkkitehtuurille. Tänä päivänä täytyy aina ensin vähän katsastella assembler-koodia ja dokumentaatiota ja päätellä jostakin, kumpi syntaksi nyt onkaan kyseessä, ja miten päin lähteitä ja kohteita ajatellaan. **Kesäkurssin 2007 kaikissa esimerkeissä ja mm. koko tällä hetkellä lukemasi lehdykän loppuun saakka lähdeoperandi on vasemmalla ja kohde oikealla puolella pilkkua!**

Käytämme siis AT&T -syntaksia, tarkemmin sen GNU-variaatiota, jota `gas` eli GNU Assembler käyttää.

Ja jotta Käyttöjärjestelmät -kesäkurssin 2007 opiskelijoiden elämä tehtäisiin hilpeämmäksi, huomataan, että varsinaisessa luentomonisteessa aina sivulta 9 alkaen kaikki assembler-esimerkit ovat juuri sillä toisella (eli Intel) syntaksilla ja operandijärjestyksellä. Aiemmin työkaluna on ollut MS-DOS -ikkunassa toimiva Borland C++ 3.1 -käännöstyökalun assembler nimeltä Turbo Assembler, joka käyttää Intel-syntaksia.

Olipa syntaksi tuo tai tämä, assembler-kääntäjän homma on muodostaa prosessorin ymmärtämä bittijono symbolisen rivin perusteella. Paljastetaan tässä, että tuo ylläoleva rivi on ohjelmasta, johon se kääntyy seuraavasti:

```
400469:       48 89 e5                movq   %rsp,%rbp

^             ^                       ^
|             |                       |__ assembler-kielinen ilmaus,
|             |                           "mnemonic" ja operandit.
|             |
|             |   Käsky konekieleksi käännettynä.
|             |__ Näköjään kyseisen käskyn bittijonossa on kolme
|                 tavua, jotka heksana ovat   48 89 e5
|                 Siis bitteinä  0100 1000  1000 1001  1110 0101
|                 jos en mokannut päässämuunnosta heksoista.
|                 Tarkista itse (=hyvä harjoitus lukujärjestelmistä).
|
|__ käskyn suhteellinen muistiosoite ohjelma-alueen alusta luettuna,
    virtuaaliosoite.
```

Assembler-käännös taitaa olla ainoa ohjelmointikäännös, joka puolijärjellisellä tavalla on tehtävissä toisin päin: Konekielinen bittijono nimittäin voidaan kääntää takaisin ihmisen ymmärtämälle assemblerille. Sanotaan, että tehdään *disassembly*. Tällä tavoin voidaan tutkia ohjelman toimintaa, vaikkei lähdekoodia olisi saatavilla. Työlästähän se on, ja "viimeinen keino" debuggauksessa tai teollisuusvakoilussa, mutta mahdollista kuitenkin. Assembler-kielinen lähdekoodi sinänsä on kokeneen silmään ihan selkeätä, mutta ilman lähdekoodia tehdyssä disassemblyssä ei ole käytettävissä muuttujien tai muistiosoitteiden kuvaavia nimiä -- kaikki on vain suhteellisia numeroindeksejä suhteessa rekisterien sisältämiin muistiosoitteisiin. Kokonaisuutta on silloin mahdoton hahmottaa. Sillä tavoin tietokone käsittelee ohjelman suoritusta eri tavoin kuin ihminen.

5 Käyttäjän näkemät rekisterit x86-64:ssa

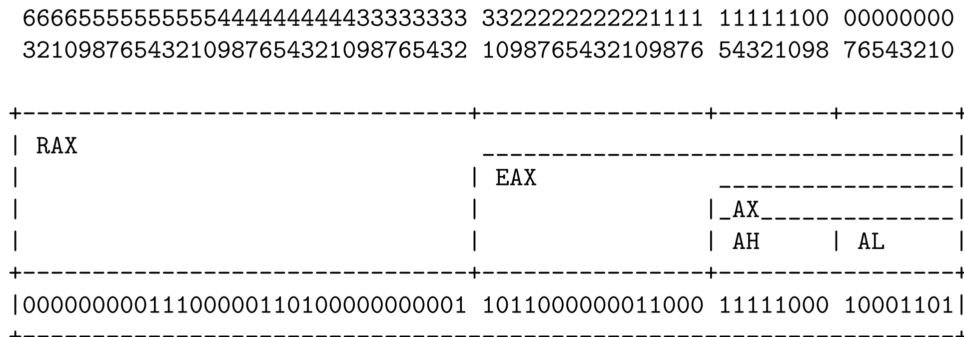
Nyt toivottavasti on riittävästi pohjatietoa, että voidaan vain esimerkinomaisesti listata eräässä prosessorissa käytettävissä olevat rekisterit merkityksineen niillä lyhyillä nimillä, jotka prosessorivalmistaja on antanut. Tässä on ns. yleisrekisterit, joita ohjelmoija voi käyttää Intelin Xeon -prosessorissa (tai muussa x86-64 arkkitehtuurin mukaisessa prosessorissa):

```
Toiminnanohjausrekisterit:

   RIP     - Instruction pointer, "IP"

   RFLAGS  - Flags, "PSW"


Yleisrekistereitä datalle ja osoitteille

   RAX     - Yleisrekisteri; "akkumulaattori"

   RBX     - Yleisrekisteri; "epäsuora osoite"

   RCX     - Yleisrekisteri; "laskuri"

   RDX     - Yleisrekisteri

   RSI     - Yleisrekisteri

   RDI     - Yleisrekisteri

   RBP     - Nykyisen aliohjelman pinokehyksen kantaosoitin

   RSP     - Osoitin suorituspinon huippuun

   R8      - Yleisrekisteri

   R9      - Yleisrekisteri

   R10     - Yleisrekisteri

   R11     - Yleisrekisteri

   R12-15  - Vielä 4 kpl Yleisrekisterejä
```

Huom: Hiukan nopeasti vilkaistu. Toimii varmasti perusidean opetteluun, mutta älä usko kaikkea ennen kuin itse luet speksin... niinhän se toisaalta aina menee

Jokaisessa x86-64:n rekisterissä voidaan säilyttää 64 bittiä. Rekistereistä voidaan käyttää joko kokonaisuutta tai 32-bittistä, 16-bittistä tai jompaa kumpaa kahdesta 8-bittisestä osasta. Seuraavassa on esimerkiksi RAX:n osat ja niiden nimet, bitit on numeroitu siten, että 0 on vähiten merkitsevä ja 63 eniten merkitsevä bitti:

```
bittien numerointi:
```



Esim. yhden 8-bittisen ASCII-merkin käsittelyyn riittäisi `AL`, 32-bittiselle kokonaisluvulle (tai 4-tavuiselle Unicode-merkille) riittäisi `EAX`, ja 64-bittinen kokonaisluku tai muistiosoite tarvitsisi koko rekisterin `RAX`.

Jatkossa keskitytään lähinnä edellä mainittuihin yleiskäyttöisiin kokonaislukurekistereihin. Käsittelemättä jätetään 32 kpl liukulukulaskentaan ja multimediakäyttöön tarkoitettua rekisteriä (`FPR0-FPR7`, `MMX0-MMX7` ja `XMM0-XMM15`) Esimerkiksi siinä vaiheessa, kun on kriittistä tehdä aiempaa tarkempi sääennuste aiempaa nopeammin, saattaa olla ajankohtaista opetella `FPR0-7`-rekisterit ja niihin liittyvä käskykannan osuus. Siinä vaiheessa, kun haluaa tehdä naapurifirmaa hienomman ja tehokkaamman 3D-koneiston tietokonepelejä tai lentosimulaattoria varten, on syytä tutustua multimediarekistereihin. Aika pitkälle "tarpeeksi tehokkaan" ohjelman tekemisessä pääsee käyttämällä liukuluku- ja multimediasovelluksissa jotakin valmista virtuaalikonetta. Joka tapauksessa ohjelman *suoritusnopeus perustuu kaikista eniten algoritmien ja tietorakenteiden valintaan, ei jonkun algoritmin konekielitoteutukseen.* Mutta älä koskaan sano ettei koskaan... voihan sitä päätyä töihin vaikka firmaan, joka nimenomaan toteuttaa noita virtuaalikoneita, jolloin kaikkein alin taso ilman muuta tehdään jonkun prosessoriarkkitehtuurin konekielellä.

Tällaisia rekisterejä siis x86-64 -tietokoneen sovellusohjelmien ohjelmoija voi nähdä ja käyttää ohjelmoimalla assemblerilla, ja niitä jokaisen sovellusohjelman konekielinen käännös aina käyttää. Ne ovat esimerkkiarkkitehtuurimme "user-visible registers". Käyttöjärjestelmäkoodi pääsee käsiksi systeemirekistereihin ja systeemikäskyihin, siis noin 50 muuhun rekisteriin sekä näihin liittyvään käskykannan osaan, joilla muistinhallintaa, laitteistoa ja ohjelmien suojausta hallitaan. Jos käyttäjän ohjelma yrittää jotakin niistä käyttää, seuraa suojausvirhe, ja ohjelma kaatuu saman tien. Käyttäjän ja käyttöjärjestelmän rekisterien lisäksi prosessorissa on sisäisiä rekisterejä väyläosoitteiden ja käskyjen väliaikaisia tallennuksia varten, mutta jätetään ne tosiaan maininnan tasolle.

Tällä kurssilla on syytä rajoittua käyttäjätilan sovelluskoodin tekemiseen assemblerilla. Käyttöjärjestelmätilasta tehdään teoreettisempia huomioita. Syy on lähtökohtaisesti se, että kesäopettajanne ei itse osaa käyttöjärjestelmätilan rekisterien käyttöä siinä määrin, että riittävä tiivistäminen ja olennaisen löytäminen olisi mahdollista. Niinpä asia jätetään meidän jokaisen myöhemmän opiskelun kohteeksi. Tältä kurssilta saadaan kuitenkin toivottavasti perusymmärrys pohjaksi myöhempään opiskeluun; se syntyy hyvin käyttäjäpuolen assemblerin ja konekielen ymmärtämisestä.

6 Käskykanta x86-64 -arkkitehtuurissa

Edellä nähtiin esimerkki konekielikäskystä, `movq %rsp,%rbp`. Mitä muita käskyjä voi esimerkiksi olla? Otetaan muutama poiminta AMD64:n manuaalin käskykantaosiosta, tiivistetään ja suomennetaan

tähän.

6.1 MOV-käskyt

Bittien siirto paikasta toiseen tapahtuu käskyllä, jonka muistike (nimi, assembler-syntaksi) on MOV. Itse asiassa GNU assemblerissa tähän lisätään vielä bittien määrää ilmaiseva kirjain. Esimerkkejä erilaisista tavoista vaikuttaa käskyn lähteeseen ja kohteeseen:

```
movq   %rsp, %rbp        # Rekisterin RSP bitit rekisteriin RBP

movl   %eax, %ebx        # 32 bitin siirto osarekisterien välillä

movq   $123, %rax        # Käskyyn sisällytetyn vakioluvun siirto
                         # rekisteriin RAX; ylin bitti monistuu
                         # siirrossa joten kaikki 64 bittiä
                         # asettuvat vaikka luku 123 mahtuu 8 bittiin

movq   %rax, -8(%rbp)    # Rekisterin RAX bitit väylän kautta
                         # muistipaikkoihin, joista ensimmäisen
                         # (virtuaali)osoite on RBP:n sisältämä
                         # osoite miinus 8. Viimeinen tavu sijoittuu
                         # paikkaan RBP-1. Missä keskinäisessä
                         # järjestyksessä 64-bittisen rekisterin 8 tavua
                         # tallentuvat noihin kahdeksaan muistipaikkaan?
                         # Tarkista itse prosessorimanuaalista
                         # kohdasta "byte order", mikäli haluat tarkan
                         # tiedon ...
                         #
                         # Myöhemmin tutustutaan pinokehysmalliin, jota
                         # noudattaen tuosta osoitteesta, eli RBP:n arvo
                         # miinus kahdeksan, voisi olettaa
                         # löytävänsä ensimmäisen nykyiselle aliohjelmalle
                         # varatun 64-bittisen lokaalin muuttujan...

movq   32(%rbp), %rax    # Rekisteriin RAX haetaan bitit väylän kautta
                         # muistipaikasta, jonka (virtuaali)osoite on
                         # RBP:n sisältämä osoite plus 32.
                         #
                         # Myöhemmin tutustutaan pinokehysmalliin, jota
                         # noudattaen tuosta osoitteesta voisi olettaa
                         # löytävänsä yhden pinon kautta välitetyistä
                         # aliohjelmaparametreista.
```

Esitellään tässä kohtaa vielä yksi tyypillinen käsky, LEA eli “load effective address” eli “lataa lopullinen osoite”:

```
lea    32(%rbp), %rax    # Osoite RPB + 32 lasketaan tässä valmiiksi,
                         # mutta sen sijaan, että siirrettäisiin
                         # osoitetun muistipaikan sisältö, laitetaankin
                         # tässä itse muistiosoite kohderekisteriin.
                         # Osoitteeseen voitaisiin sitten kohdistaa vielä
                         # laskutoimituksia ennen kuin sitä käytetään.
                         # Esimerkiksi voitaisiin ynnätä taulukon indeksin
                         # mukainen luku taulukon ensimmäisen alkion
```

```
                                  # osoitteeseen ...
```

Näin ollen käskypari `lea 32(%rbp), %rdx` ja sen perään `movq (%rdx), %rax` tekisi saman kuin `movq 32(%rbp), %rax`. Ja yksi käyttötarkoitus on siis esim. yhdistelmä:

```
lea   32(%rbp), %rdx       # Taulukon alkuosoite RDX:ään
addq  %rcx, %rdx           # Siirros RCX on laskettu valmiiksi esim.
                           # silmukkalaskurin päivityksen yhteydessä
movq  (%rdx), %rax         # Kohdistetaan haku taulukon sisällä olevaan
                           # muistipaikkaan.
```

6.2 Pinokäskyt

Toinen tapa siirtää bittejä paikasta toiseen on käyttää **suorituspinoa** (engl. *stack*). Silloin siirron lähde tai kohde on aina pinon huippu, jonka muistiosoite on rekisterissä RSP. Kun pinoon laitetaan jotakin tai sieltä otetaan jotakin pois, prosessori tekee automaattisesti siirron väylän kautta keskusmuistiin nimenomaan pinoalueelle tai vastaavasti sieltä johonkin rekisteriin. (Huomaa, että koko ajan tarkoituksella ohitetaan välimuistiin liittyvät tekniset yksityiskohdat!). Samalla se päivittää pinon huippua ylös tai alaspäin. Pari esimerkkiä:

```
pushq $12                   # Ensinnä RSP:n arvo pienenee 8:lla, koska
                            # käskyssä on ``q`` mikä tarkoittaa
                            # 64-bittistä siirtoa. Muistiosoitteethan
                            # ovat aina yhden tavun eli 8 bitin kokoisten
                            # muistipaikkojen osoitteita.
                            #
                            # Siihen kohtaan muistia (osoite uusi RSP)
                            # menee sitten luku 12, eli 64-bittinen luku
                            # jonka heksaesitys on 0x000000000000000c.

pushl %edx                  # RSP:n arvo pienenee 4:lla, koska
                            # käskyssä on ``l`` mikä tarkoittaa
                            # 32-bittistä siirtoa. Siihen kohtaan muistia
                            # menee sitten ne 32 bittiä, jotka ovat
                            # rekisterissä EDX eli RDX:n 32-bittinen puoli.
```

Kun pinoon laitetaan jotain, RSP tosiaan pienenee, koska pinon pohja on suurin muistiosoite, ja pinon huippu kasvaa muistiosoitemielessä alaspäin. Tässäkin on varmaan joku hyvä tekninen perustelu, jota en tässä osaa kertoa... En ole varma syistä, mutta näin se kuitenkin on useimmiten toteutettu. Pinon päältä voi ottaa asioita vastaavasti:

```
popq %rbx                   # Ensinnä prosessori siirtää RSP:n sisältämän
                            # muistiosoitteen kertomasta muistipaikasta
                            # 64 peräkkäistä bittiä rekisteriin RBX.
                            # Sen jälkeen se lisää RSP:n arvoon 8, eli
                            # tuloksena pinon huippu palautuu 64-bittisellä
                            # pykälällä kohti pohjaa.
```

6.3 Aritmeettisia käskyjä

Edellä olevat siirto- ja pinokäskyt vain siirtävät bittejä paikasta toiseen. Ohjelmointi edellyttää usein bittien muokkaamista matkalla. Pari esimerkkiä:

```
addq %rdx, -32(%rbp)        # Hakee muistipaikasta (RBP:n arvo - 32) löytyvät
                            # bitit, laskee ne yhteen rekisterissä RDX olevien
```

```
                         # bittien kanssa ja sijoittaa tuloksen takaisin
                         # muistipaikkaan (RBP:n arvo - 32).
                         # Väylää tarvitaan kolmessa kohtaa: käskyn nouto,
                         # operandin nouto, tuloksen tallennus.

addq $17, %rax           # Usein ynnätään "akkumulaattoriin" eli
                         # RAX-rekisteriin. Tässä luku 17 on mukana käskyn
                         # konekielikoodissa; se lisätään RAX:n arvoon ja
                         # tulos jää RAX:ään. Ylimääräisiä muistipaikkojen
                         # käyttöjä ei käskyn noudon lisäksi tarvita,
                         # joten tämä vaatii vähemmän kellojaksoja kuin
                         # edellä esitelty yhteenlasku

subl 20(%rbp), %eax      # EAX-rekisterin arvosta vähennetään luku, joka
                         # haetaan ensin muistipaikasta RBP+20; tulos
                         # jää EAX-rekisteriin.
```

Prosessorit tarjoavat kokonaislukulaskentaan usein myös MUL-käskyn kertolaskulle ja DIV-käskyn jakolaskulle (tuloksena erikseen osamäärä ja jakojäännös). Näistä on usein erikseen etumerkillinen ja etumerkitön versio. Niin on myös x86-64 -arkkitehtuurissa. Aritmeettiset käskyt vaikuttavat lippurekisterin RFLAGS tiettyihin bitteihin, esimerkkejä:

- Yhteenlaskun muistibitti jää talteen, *Carry flag*
- Jos tulos on nolla, asettuu *Zero flag*
- Jos tulos on negatiivinen, asettuu *Negative flag*

Liukulukujen laskentaan pitää käyttää erillisiä liukulukurekisterejä ja liukulukukäskyjä. Niihin ei mennä tässä.

6.4 Bittilogiikkaa

Moniin tarkoituksiin tarvitaan bittien muokkaamista. Pari esimerkkiä:

```
notq  %rax       # Kääntää RAX:n kaikki bitit nollasta ykkösiksi tai
                 # toisin päin, siis bitittäinen looginen EI-operaatio.

andq  $15, %rax  # Bitittäinen looginen JA-operaatio. Tässä tapauksessa
                 # 15 on bitteinä 000...001111 eli neljä alinta bittiä
                 # ykkösiä ja loput 60 kpl nollia. Lopputuloksena RAX:n
                 # 60 ylintä bittiä ovat varmasti nollia ja puolestaan
                 # 4 alinta bittiä jäävät aiempaan arvoonsa, eli looginen
                 # JA toteuttaa bittien "maskaamisen". (Tämä btw on
                 # hyödyllinen kikka myös korkean tason kielillä
                 # ohjelmoidessa)

testq $15, %rax  # TEST tekee saman kuin AND, mutta ei tallenna tulosta
                 # mihinkään. Miksi näin? Liput eli RFLAGS päivittyy, eli
                 # esim. tässä tapauksessa jos tulos on nolla, Zero flag
                 # kertoisi käskyn jälkeen että mikään RAX:n neljästä
                 # alimmasta bitistä ei ole asetettu.

orq   %rdx, %rcx # Bitittäinen looginen TAI-operaatio
                 # (Tätä voi käyttää bittien asettamiseen: ne jotka
                 # olivat ykkösiä RDX:ssä tulevat ykkösiksi RCX:ään,
```

```
                # ja ne jotka olivat nollia RDX:ssä jäävät ennalleen
                # RCX:ssä).

xorq  %rax, %rax # Bitittäinen looginen JOKO-TAI -operaatio. Esimerkissä
                # molemmat operandit ovat RAX, jolloin JOKO-TAI
                # aiheuttaa RAX:n kaikkien bittien nollautumisen, mikä
                # vastaa luvun nolla sijoittamista rekisteriin, mutta
                # voi olla nopeampi suorittaa (oli aikoinaan 286:ssa ym.
                # mutta en tiedä x86-64 -vehkeistä).
```

6.5 Muita bittien muokkausoperaatioita

Onhan näitä. Joitain esimerkkejä:

```
sarb  $3, %ah    # Siirtää 8-bittisen (''b'', byte) rekisteriosan bittejä
                 # kolmella pykälällä oikealle, eli jos siellä oli bitit
                 # 0110 0101 niin sinne jää käskyn jälkeen 0000 1100.

rolw  %cl, %ax   # Pyörittää 16-bittisen (''w'', word) rekisteriosan
                 # bittejä vasemmalle niin monta pykälää kuin 8-bittisen
                 # rekisteriosan CL viisi alinta bittiä kertovat. Siis
                 # esim. jos CL on 0100 0100 (eli viisi alinta bittiä ovat
                 # lukuarvo 4) ja AX on 1000 0011 0000 1110 niin pyöritetty
                 # tulos olisi 0011 0000 1110 1000
```

Pyörityksiä ja siirtoja on vasemmalle ja oikealle (SAR, SAL, ROR, ROL); näissä pyörityksen voi tehdä ilman Carry-lippua tai sitten voi pyörittää siten, että laidalta pois putoava bitti siirtyy Carry-lipuksi ja toiselta laidalta sisään pyöritettävä tulee vastaavasti Carrystä. Sitten on kokonaisluvun etumerkin vaihto NEG, ja niin edelleen. Tämä ei ole täydellinen konekieliopas, joten jätetään esimerkit tälle tasolle ja todetaan, että on niitä muitakin, mutta että suurin piirtein tällaisia asioita niillä tehdään, yksinkertaisia bittien siirtoja paikasta toiseen...

6.6 Suoritusjärjestyksen ohjaus: mistä on kyse

Tähän asti mainituista käskyistä muodostuva ohjelma suoritetaan **peräkkäisjärjestyksessä**, eli käskystä siirrytään aina välittömästi seuraavaan käskyyn. Tarkemmin: prosessori päivittää IP:n seuraavan käskyn osoitteeksi suorituksen jälkeen, ennen seuraavaa noutoa. Peräkkäisjärjestys on ohjelmoinnin peruste, niinkuin suorassa seisominen ja käveleminen ovat perusteita juoksemiselle, hyppäämiselle ja muulle vaativammalle liikkumiselle.

Ohjelmoinnista tietänet, että algoritmien toteuttaminen vaatii myös muita kuin peräkkäisiä suorituksia, erityisesti tarvitaan:

- **ehtorakenteet**, eli jotain tehdään vain silloin kun joku looginen lauseke on tosi
- **toistorakenteet**, eli jotain toistetaan useaan kertaan, kunnes jokin lopetuskriteeriä kuvaava looginen lauseke muuttaa arvoaan.
- **aliohjelmat** (tai **metodit**), eli suoritus täytyy voida siirtää toiseen ohjelman osioon väliaikaisesti. Myös aliohjelmasta pitää voida sopivaan aikaan palata siihen kohtaan, missä aliohjelmaa kutsuttiin.
- **poikkeukset**, jotka ovat hiukan myöhemmin keksitty lisuke edellä mainittuihin: suoritus täytyy pystyä siirtämään muualle kuin kutsuvaan aliohjelmaan, tai sitä kutsuneeseen tai niin edelleen... itse asiassa täytyy kyetä palaamaan niin kauas, että löytyy poikkeuksen käsittelijä.

Poikkeukset helpottavat ohjelmointia (tai vaikeuttavat, näkökulmasta riippuen...), mutta eivät sinänsä ole välttämättömiä ohjelmien tekemiselle. Kolme ensimmäistä ovat sangen välttämättömiä, ja nyt tutustutaan siihen, millaisilla käskyillä konekielessä saadaan aikaan ehto- ja toistorakenteet sekä aliohjelmien kutsuminen.

6.7 Suoritusjärjestyksen ohjausta: ehdot ja toistot

Konekielikoodi sijaitsee tavuina keskusmuistin muistipaikoissa, joiden muistiosoitteet ovat peräkkäisiä. Ehdollinen suoritus ja silmukat perustuvat ehdollisiin ja ehdottomiin hyppykäskyihin, esimerkkejä:

```
jmp  MUISTIOSOITE      # Ehdoton hyppy "jump". Tämän käskyn suorituksen
                       # kohdalla prosessori lataa uudeksi
                       # käskyosoitteeksi (RIP-rekisteriin) osoitteen,
                       # joka käskyssä kerrotaan. Käännetyssä
                       # konekielessä osoite on tyypillisesti
                       # suhteellinen osoite hyppykäskyn oman muistipaikan
                       # osoitteeseen nähden, eli se on mallia "hyppää
                       # 48 tavua eteenpäin" tai "hyppää 112 tavua
                       # taaksepäin". Ensimmäisessä em. esimerkissä RIP
                       # päivittyisi RIP := RIP + 48 ja toisessa
                       # esimerkissä RIP := RIP - 112.

jz  MUISTIOSOITE       # Ehdollinen hyppy "jump if Zero". Hyppy on kuten
                       # jmp, mutta se tehdään vain silloin kun Zero flag
                       # on asetettu, eli kun edellisen aritmeettisen tai
                       # loogisen operaation tulos oli nolla. Jos RFLAGSin
                       # Zero-bitti ei ole asetettu, hyppyä ei tehdä vaan
                       # käskyn suorituksessa ainoastaan päivitetään RIP
                       # osoittamaan seuraavaa käskyä, ihan kuin
                       # peräkkäisesti suoritettavissakin käskyissä

jnz MUISTIOSOITE       # Ehdollinen hyppy "jump if not Zero". Arvatenkin
                       # hyppy tehdään silloin kun Zero flag -bitti ei ole
                       # asetettu eli edeltävä käsky ei antanut tulokseksi
                       # nollaa.

jg  MUISTIOSOITE       # "Jump if Greater" eli aiemmassa vertailussa (tai
                       # vähennyslaskussa) lähdeoperandi oli suurempi kuin
                       # kohdeoperand [HEP! Tai toisin päin, tämä on aina
                       # yhtä vaikea muistaa, en jaksa katsoa manuaalista]
                       # Ehto selviää tietysti RFLAGSissä olevista
                       # biteistä, kuten kaikissa ehdollisissa hypyissä.

jng  MUISTIOSOITE      # "Jump if not greater"

jle  MUISTIOSOITE      # "Jump if less or equal"

jnle  MUISTIOSOITE     # "Jump if not less or equal"

... ja niin edelleen ... näitä on melko monta variaatiota, jotka
kaikki toimivat samoin ...
```

Korkean tason kielellä kuten C:llä tai Javalla ohjelmoija ei tee itse lainkaan hyppykäskyjä, vaan hän kirjoittaa silmukoita silmukkasyntaksilla (esim. `for..` tai `while..`) ja ehtoja ehtosyntaksilla (kuten `if`

`.. else if..`). Kääntäjä tuottaa kaikki tarvittavat hypyt ja bittitarkistukset. Jos ohjelmoidaan suoraan assemblerilla, pitää hypyt ohjelmoida itse, mutta suhteellisia muistiosoitteita ei tarvitse tietenkään itse laskea, vaan assembler-kääntäjä osaa muuntaa symboliset nimet sopivasti. Esimerkiksi seuraava ohjelma laskisi luvusta 1000 lukuun 0 rekisterissä RAX:

```
ohjelman_alku:                                  # symbolinen nimi muistipaikalle
                 movq   $1000, %rax

silmukan_alku:                                  # symbolinen nimi muistipaikalle
                 subq   $1, %rax
                 jnz    silmukan_alku           # Kääntäjä osaa laskea montako
                                                # tavua taaksepäin on hypättävä
                                                # että uudesta osoitteesta löytyy
                                                # edelläkirjoitettu subq-käsky.
                                                # Tuon miinusmerkkisen luvun se
                                                # koodaa mukaan konekielikäskyyn.
```

Huomaa, että sama asia voidaan toteuttaa erittäin monella erilaisella konekielikäskyjen sarjalla, esim. edellinen voisi olla:

```
ohjelman_alku:
                 movq   $1000, %rax

silmukan_alku:
                 subq   $1, %rax
                 jz     silmukka_ohi
                 jmp    silmukan_alku
silmukka_ohi:
                 ... tästä jatkuisi koodi eteenpäin ...
```

6.8 Suoritusjärjestyksen ohjausta: aliohjelmat

Käydään tässä kohtaa läpi aliohjelmaan liittyvät konekielikäskyt. Lisää aliohjelman suorittamisesta on myöhemmässä kohdassa, jossa käsitellään parametrien ja paluuarvon välittämistä sekä paikallisia muuttujia. Käskyt ovat:

```
call MUISTIOSOITE         # Tämä on ehdoton hyppy, ihan kuin edellä
                          # esitetty jmp-käsky, mutta ennen kuin RIP:n
                          # arvo päivitetään uudeksi osoitteeksi,
                          # seuraavan käskyn osoite (joka
                          # peräkkäissuorituksessa ladattaisiin RIP:hen)
                          # painetaan pinoon. Siis ikäänkuin prosessori
                          # tekisi  " pushq SEURAAVAN_KÄSKYN_OSOITE;
                          #           jmp MUISTIOSOITE "
                          # Kuitenkin molemmat asiat tapahtuvat yhdellä
                          # call-nimisellä käskyllä. Osoitteen laittaminen
                          # pinoon mahdollistaa palaamisen aliohjelmasta

ret                       # Tämä on paluu aliohjelmasta, ihan kuin edellä
                          # esitetty jmp-käsky, mutta RIP:hen laitettava
                          # arvo otetaan pinon päältä, siis muistipaikasta,
                          # jonka osoite on RSP:ssä. Eli ikäänkuin olisi
                          # "popq %rip", mutta käsky tosiaan on "ret".
```

Em. käskyillä siis hoidetaan suoritusjärjestys aliohjelmakutsun yhteydessä, ja tähän tarvitaan pinomuistia. Aliohjelmiin liittyy myös muuttujien käyttö, eli pitää voida välittää parametreja ja paluuarvoja sekä käyttää paikallisia muuttujia. Myöhemmin esitetään tarkemmin ns. pinokehysmalli. Siihen tulee liittymään seuraavat x86-64:n käskyt:

```
enter $540                 # Tämä käsky kuuluisi heti aliohjelman alkuun.
                           # Se loisi juuri kutsutulle aliohjelmalle oman
                           # pinokehyksen, ja tässä tapauksessa varaisi
                           # tilaa 540 tavulle paikallisia muuttujia.
```

Em. ENTER-käsky tekee yhdessä operaatiossa kaikkien seuraavien käskyjen asiat, eli se on laitettu käskykantaan helpottamaan ohjelmointia tältä osin... muuten pitäisi kirjoittaa:

```
pushq %rbp                 # RPB talteen pinoon
movq  %rsp, %rbp           # Merkitään nykyinen RSP uuden pinokehyksen
                           # kantaosoitteeksi eli RBP := RSP
subq  $540, %rsp           # Varataan 540 tavua tilaa lokaaleille
                           # muuttujille.
```

Tähän komentosarjaan (tai ENTER-käskyyn) tullee lisää järkeä, kun luet myöhemmän pinokehyksiä käsittelevän kohdan. Käsky on oikeasti vähän monipuolisempi; siinä voisi olla mukana toinen operandi, joka liittyisi pääsyyn aiempien toisiaan kutsuneiden aliohjelmien pinokehyksiin (eli ns. kutsupinossa ylöspäin...). Mutta ei mennä siihen nyt; riittää kun ajatellaan kerrallaan yhtä aliohjelmakutsua ja sen pinokehystä.

Vastaavasti aliohjelman lopussa voidaan käyttää LEAVE-käskyä:

```
leave                      # Vapauttaa nykyisen pinokehyksen, eli
                           # hukkaa paikallisille muuttujille varatun
                           # tilan pinosta, ja palauttaa pinon huipun
                           # sellaiseksi, että siitä löytyy RET-käskyn
                           # edellyttämä paluuosoite.
```

Tämä LEAVE-käsky tekisi yhdessä operaatiossa seuraavien käskyjen asiat; jos funtsit asiaa hetken, huomannet, että nämä kumoavat kokonaan ENTERin tekemät tilamuutokset:

```
movq %rbp, %rsp            # RBP oli se aiempi RSP ... tilanteessa jossa
                           # oli juuri pinottu edeltävä RBP...
pop  %rbp                  # Niinpä se edeltävä RBP saadaan palautettua
                           # pop-käskyllä. Ja POP-käskyssähän RSP
                           # palautuu yhdellä pykälällä pohjaa kohti.
```

Ilo tästä on, että ENTERin ja LEAVEin välisessä koodissa SP on aina vapaana uuden kehyksen tekemiselle eli seuraavan sisäkkäisen aliohjelmakutsun tekemiselle. Tästä tosiaan lisää myöhemmin.

6.9 Suoritusjärjestyksen ohjausta: keskeytyspyyntö

Käsitellään vielä ohjelmallinen keskeytyspyyntö, joka on prosessorin käsky. Yleisiä nimiä sille on esim. “supervisor call, SVC”, “trap”, “interrupt request”. Keskeytyspyyntö on avain käyttöjärjestelmän käyttöön: kaikki käyttöjärjestelmän palvelut ovat saatavilla vain sellaisen kautta. Eli **käyttöjärjestelmän rajapinta**, engl. **system call interface** näyttäytyy joukkona palveluita, joita käyttäjän ohjelmassa pyydetään ohjelmoidun keskeytyksen avulla. Keskeytyspyyntö näyttäisi x86-64:ssä seuraavanlaiselta:

```
int $10                    # Pyydetään keskeyttämään tämä prosessi
                           # ja suorittamaan keskeytyskäsittelijä
                           # numero 10. Oletus on, että jossain
                           # vaiheessa suoritus palaa tätä käskyä
```

```
# seuraavaan käskyyn, ja
# käyttöjärjestelmäkutsu on toteuttanut
# palvelunsa. Paitsi tietysti, jos pyyntö
# on tämän prosessin lopettaminen eli
# exit() -palvelu. Silloin oletus on, että
# seuraavaa käskyä nimenomaan ei koskaan
# suoriteta.
```

Keskeytyspyynnön toteutus laitetasolla on ehkä mutkikkain, joka käskykannasta löytyy. Jos haluat katsoa esim. AMD64:n manuaalia, löydät sieltä viisi sivua pseudokoodia, joka kertoo kaikki prosessorin toimenpiteet. Tällä kurssilla emme käsittele keskeytyspyyntöä noin yksityiskohtaisesti, vaan todetaan, että jos INT suoritetaan 64-bittisessä käyttäjätilassa (normaalin sovellusohjelman normaali suoritustila x86-64:ssä), sen suorituksen jälkeen on voimassa seuraavaa:

- RSP:hen on ladattu uusi muistiosoitin, eli pinon paikka on eri kuin keskeyttävän prosessin pino, tästä alkaen käytössä taitaa olla siis luentomonisteessakin mainittu “Kernel-pino” (en äärellisessä ajassa ehtinyt opiskella, että mistä tuo uusi pino-osoitin vedetään; siihen liittyy prosessorin laitetasolla käyttämiä tietorakenteita, joissa on jonkin verran sälää... En osaa kertoa äkkiseltään esim. että onko jokaisella prosessilla tosiaan oma Kernel-pino vai onko käyttöjärjestelmällä yksi ja ainoa Kernel-pino... hmm... mutta eipä sitä tarvitse tällä kurssilla oppia eikä opettajankaan tarvitse tietää... mutta joka tapauksessa pino-osoitin muuttuu aina x86-64:n keskeytyksessä joksikin muuksi kuin mikä se oli, kun prosessia suoritettiin käyttäjätilassa.)
- Keskeytyskäsittelijän käyttämän pinon päällä (siis uuden RSP:n osoittamassa pinossa) on tärkein osuus keskeytetyn prosessin kontekstista:
  - RFLAGSin sisältö ennen INTin suoritusta
  - INT-käskyä seuraavan käskyn muistiosoite (eli se joksi RIP olisi päivitetty peräkkäissuorituksessa)
  - keskeytetyn prosessin pino-osoitin (eli RSP:n sisältö ennen INTin suoritusta).

  Muita rekistereitä ei ole laitettu mihinkään; niissä on yhä keskeytetyn prosessin konteksti.
- RFLAGS on päivitetty seuraavin tavoin: Prosessori on käyttöjärjestelmätilassa ja keskeytykset ovat toistaiseksi kiellettyjä (jotta atomista käsittelyä voidaan jatkaa ilman että tulee uusia keskeytyksiä; esim. semaforipalvelu olisi järjetön, jos kellokeskeytys pääsisi aiheuttamaan vuoronnuksen ennen kuin semaforin lukuarvoa on atomisesti muutettu).
- RIP:hen on ladattu muistiosoite, jonka osoittamassa muistipaikassa on hyppykäsky pyydettyyn keskeytyskäsittelijään. Näistä peräkkäisissä muistipaikoissa sijaitsevista hyppykäskyistä muodostuu ns. **keskeytysvektori**; se on käyttöjärjestelmän luoma muistialue, jossa on hypyt myös niihin käsittelijöihin, joilla oheislaitteiden pyytämät keskeytykset hoidetaan. Ohjelmoidussa keskeytyksessä osoite riippuu INT-käskyn 8-bittisestä operandista, eli ohjelma voi pyytää mitä tahansa käsittelijää väliltä 0-255. Käyttöjärjestelmän palvelut löytyvät usein jollain tietyllä numerolla, tai muutamalla eri numerolla palveluiden tyypin mukaan jaoteltuna. Valinta riippuu siis täysin siitä, miten käyttöjärjestelmä on toteutettu.
- Muutakin voi olla, mutta tuossa on tärkeimmät asiat, joiden avulla keskeytys saadaan hoidettua, ja suoritus siirrettyä käyttäjän ohjelmalta käyttöjärjestelmälle.

RIP, RSP ja RFLAGS on välttämätöntä saada tallennettua atomisessa keskeytyskäsittelyssä (first-level interrupt handling), koska ne ovat kaikkein herkimmin muuttuvat rekisterit; esim. RFLAGS muuttuu melkein jokaisen käskyn jälkeen ja RIP ihan jokaisen käskyn jälkeen. Jos keskeytyskäsittelyssä pitää tehdä kontekstin vaihto, käsittelijän pitää erikseen tallentaa kaikkien rekisterien sisältö keskeytetyn

prosessin PCB:hen, ja sen on huolehdittava tarkasti siitä, että se itse ei ole vahingossa muuttanut (tai päästänyt uusia keskeytyksiä muuttamaan) rekisterien arvoja ennen kontekstin tallennusta.

Käyttöjärjestelmän palvelun tarvitsemat parametrit pitää olla laitettuna rekistereihin ennen keskeytyspyyntöä, ja tietenkin käyttöjärjestelmän rajapintadokumentaatio kertoo, mihin rekisteriin pitää olla laitettu mitäkin. Parametrina voi olla esim. muistiosoite tietynlaisen tietorakenteen alkuun, jolloin käytännössä voidaan välittää käyttöjärjestelmän ja sovelluksen välillä mielivaltaisen muotoista dataa. Toinen tyypillinen tapa on välittää kokonaisluvuiksi koodattuja "deskriptoreita" tai "kahvoja" jotka yksilöivät joitakin käyttöjärjestelmän kapseloimia tietorakenteita kuten semaforeja, prosesseja (PID), käyttäjiä (UID), avoimia tiedostoja (tiedostodeskriptori), viestijonoja ym.

Paluu käyttöjärjestelmäkutsusta tapahtuu seuraavasti:

```
iret            # Keskeytyskäsittelijän lopussa pitäisi olla tämä
                # käsky. Se on käänteinen INT-käskylle, eli prosessori
                # ottaa pinosta INTin aikoinaan sinne laittamat asiat
                # (tai siis olettaa että siellä on juuri ne)
                # ja sijoittaa ne asiaankuuluviin paikkoihin. Keskeytetyn
                # prosessin kannalta tämä näyttää siltä kuin mitään ei
                # olisi tapahtunutkaan: koko konteksti, mukaanlukien
                # RFLAGS, RSP, RIP ovat niinkuin ennenkin, paitsi jos
                # käyttöjärjestelmäpalvelu on antanut paluuarvon, se
                # löytyy nätisti dokumentaatiossa kerrotusta rekisteristä,
                # tai se on muuttunut muistialueella, jonka osoite oli
                # kutsun parametrina.
```

Samalla käskyllä palataan sekä ohjelmoidun keskeytyksen että I/O:n aiheuttaman keskeytyksen käsittelijästä. Muistutus: prosessori on jatkuvasti yhtä tyhmä kuin aina, eikä se IRETin kohdalla tiedä, mistä se on siihen kohtaan tullut. Se kun suorittaa yhden käskyn kerrallaan eikä näe muuta. Käyttöjärjestelmän ohjelmoijan vastuulla on järjestellä keskeytyskäsittelyt oikeellisiksi, ja mm. IRET oikeaan paikkaan koodia.

Lopuksi todetaan kaksi käskyä keskeytyksiin liittyen:

```
cli             # Estää keskeytykset; eli kääntää RFLAGSissä olevan
                # keskeytyslipun nollaksi (clear Interrupt flag)

sti             # Sallii keskeytykset; eli kääntää RFLAGSissä olevan
                # keskeytyslipun ykköseksi (set Interrupt flag)
```

Nämä käskyt on sallittu vain käyttöjärjestelmätilassa (käyttäjän ohjelma ei voi estää keskeytyksiä, joten ainoa tapa saada aikaan atomisesti suoritettavia ohjelman osia on pyytää käyttöjärjestelmän palveluja, esimerkiksi MUTEX-semaforia). Kaikkia keskeytyksiä ei voi koskaan estää, eli on ns. "nonmaskable"-keskeytyksiä. Niiden käsittely ei saa kovin kummasti muuttaa prosessorin tai muistin tilaa, vaan keskeytyskäsittelijän on luotettava siihen, että jos keskeytykset on kielletty, niin silloin suoritettava käsittelijä on ainoa, joka voi muuttaa asioita järjestelmässä. Tiettyjä toteutuksellisia hankaluuksia syntyy sitten jos on monta prosessoria: Yhden prosessorin keskeytysten kieltäminen kun ei vaikuta muihin prosessoreihin, ja muistihan taas on kaikille prosessoreille yhteinen... mutta SMP:n yksityiskohtiin ei mennä nyt syvällisemmin; todetaan, että niitä varten tarvitaan taas tiettyjä lisukkeita käskykantaan, että muistinhallinta onnistuu ilman konflikteja. Ja monen prosessorin synkronointi edellyttää kompleksisuutta, joka osaltaan syö prosessoriaikaa, eli kaksi rinnakkaista prosessoria ei ole ihan kaksi kertaa niin nopea kokonaisuus kuin yksi kaksi kertaa nopeampi prosessori.

6.10 Huomio keskeytyskäsittelystä

Edellä kerrottiin, miten käyttöjärjestelmän kutsurajapintaan päästään käsiksi ohjelmoidun keskeytyksen kautta. Täysin samat asiat voivat tapahtua milloin vain joltakin laitteelta tulee I/O -keskeytys. Näitä

keskeytyksiä käyttäjän prosessi ei huomaa millään tavoin. On vaan ymmärrettävä, että keskeytyksiä voi tulla moniajojärjestelmässä milloin vain, ja tästä aiheutuu muutamia seikkoja:

- koskaan ei voi tietää etukäteen kuinka monta nanosekuntia, millisekuntia tai viikkoa esimerkiksi kestää ennen kuin käyttäjän ohjelman seuraava käsky suoritetaan. Jos tarkka ajoitus on välttämätöntä, pitää olla käyttöjärjestelmä ja laitteisto, jotka voivat tarjota riittävän tarkan ajastuksen erityisenä palveluna (puhutaan reaaliaikakäyttöjärjestelmästä). Tai sitten on käytettävä jotakin muuta kuin moniajokäyttöjärjestelmää; sekin on tottakai mahdollista, riippuen rakennettavan järjestelmän vaatimuksista. 1980-luvulla silloiset hienoimmat tietokonepelit toteutettiin kokonaan käyttöjärjestelmätilassa ilman moniajoa; niihin saatiin äärimmäisen tarkka ajoitus laskemalla kuinka monta kellojaksoa minkäkin koodipätkän suorittaminen kesti.

7 Keskeytykset ja lopullisempi visio fetch-execute -syklistä

Nyt on tutustuttu yhden ohjelman ajamiseen ja fetch-execute -sykliin. Sitä prosessori tekee ohjelmalle, ja yhden ohjelman kannalta näyttää ettei mitään muuta olekaan. Mutta nähtävästi koneissa on monta ohjelmaa yhtäaikaa -- miten se toteutetaan? Pelkkä fetch-execute -sykli lehdykän alussa kuvatulla tavalla ei oikein hyvin mahdollista kontrollin vaihtoa kahden ohjelman välillä. Apuna tässä ovat keskeytykset. Niitä käsitellään kurssin varsinaisessa luentomonisteessa enemmän, mutta todetaan tässä kohtaa keskeytysten nivoutuminen prosessorilaitteiston toimintaan.

Tietokonearkkitehtuuriin kuuluva ulkoinen väylä on kiinni prosessorin nastoissa, ja prosessori kokee nastoista saatavat jännitteet. Ainakin yksi nastoista on varattu **keskeytyspulssille** (*interrupt signal*): Kun oheislaitteella tapahtuu jotakin uutta, eli vaikkapa näppäimen painallus päätteellä, syntyy väylälle jännite keskeytyspulssin piuhaan kyseiseltä laitteelta prosessorille. Laite voi olla verkkoyhteyslaite, kovalevy, hiiri tai mikä tahansa oheislaite. Sillä on useimmiten väylässä kiinni oleva sähköinen kontrollikomponentti, jota sanotaan laiteohjeimeksi tai I/O -yksiköksi. Jos vaikka kovalevyltä on aiemmin pyydetty jonkun tavun nouto tietystä kohtaa levyn pintaa, se voi ilmoittaa keskeytyksellä, että se olisi valmis toimittamaan tavun dataväylälle, kunhan prosessori vain seuraavan kerran ehtii. Ja prosessori ehtii usein välittömästi, koska täydennämme aiemmin yhdelle ohjelmalle ajatellun nouto-suoritussyklin seuraavalla versiolla:

1. **Nouto**: Prosessori noutaa dataa `IP`-rekisterin osoittamasta paikasta
2. **Suoritus**: Prosessori suorittaa käskyn
3. Käskyn suorituksen tuloksena rekisterien tila on muuttunut jollain tavoin; myös muistin sisältö voi olla muuttunut riippuen käskystä.

   Yksi, joka aina muuttuu, on `IP`. Miten `IP` muuttuu, riippuu suoritetusta käskystä:

   - Laskutoimitus, datan siirto tai muu peräkkäissuoritus ==> `IP` osoittaa seuraavaan konekielikäskyyn
   - Hyppykäsky ==> `IP` osoittaa käskyssä kerrottua uutta osoitetta, esim. silmukan alkua tms.
   - Ehdollinen hyppykäsky ==> `IP` osoittaa käskyssä kerrottua uutta osoitetta mikäli käskyssä kerrottu ehto toteutuu; muutoin osoittaa seuraavaan käskyyn
   - Aliohjelmakutsu ==> `IP` osoittaa käskyssä kerrottua uutta osoitetta
   - Paluu aliohjelmasta ==> `IP` osoittaa taas siihen ohjelmaan, joka suoritti kutsun, erityisesti kyseessä on aliohjelmakutsua välittömästi seuraava käsky.
4. **Keskeytyskäsittely**: Jos keskeytysten käsittely on kielletty (eli kyseinen tilabitti `FLAGS`-rekisterissä kertoo niin), prosessori jatkaa sykliä kohdasta 1. Muutoin se tekee vielä seuraavaa:

   Jos prosessorin keskeytyspyyntö -nastassa on jännite, se siirtyy keskeytyskäsittelijään:

   - Olennaisesti tällöin tapahtuu samanlaiset operaatiot kuin ohjelmoidussa keskeytyksessä (katso x86-64:n INT-käskyn kuvaus edellä). Yhdellä kertaa suoritettavien toimenpiteiden yleisnimi on **FLIH** eli *First-level interrupt handling.*

   Tämän jälkeen prosessori jatkaa sykliä kohdasta 1, jolloin seuraava noudettava käsky on joko käyttäjän prosessin tai keskeytyskäsittelijän koodia, riippuen edellä mainituista tilanteista (keskeytysten salliminen, keskeytyspyynnön olemassaolo).

Tämän kuvauksen tavoite oli antaa yleistietoa, jonka pohjalta on mahdollisuus ymmärtää paremmin käyttöjärjestelmän osien toimintaa: prosessien vuorottelua, viestinvälitystä ja synkronointia, laiteohjausta sekä I/O:ta. Relevantti kysymys, joka voi herätä, on, voiko pyydetty keskeytys jäädä palvelematta, jos edellinen käsittely kestää pitkään siten että keskeytykset on kielletty. Vastaus on, että

ei, mutta tämä edellyttää laitteistotasolla toteutettua jonotuskäytäntöä, jonka yksityiskohtiin tässä ei mennä. Joka tapauksessa esim. multimedialaitteiden keskeytyksiä on syytä päästä palvelemaan mahdollisimman nopeasti pyynnön jälkeen, jotta median tulostukseen ei tule katkoja. Keskeytyskäsittelijän koodi tulisi olla siten tehty, että mahdollisimman pian käsittelijään siirtymisen jälkeen se suorittaa `sti` -konekäskyn, eli sallii prosessorille uuteen keskeytykseen siirtymisen vaikkei edellinen käsittely olisi kokonaan loppunutkaan.

8 Yksinkertainen esimerkki konekieliohjelmasta

Luentomonistetta mukaillen katsotaan päällisin puolin lyhyttä C-kielistä ohjelmaa ja sen konekielikäännöstä. Ohjelma on muulla tavoin sama kuin luentomonisteen sivu 9, mutta muuttujien tyyppi on käsittelyn yksinkertaistamiseksi koko rekisterin pituus:

```
/*  Varataan tilaa kolmelle 64-bittiselle luvulle (long long int)
 *  ja lasketaan sellaiset yhteen. Ei järkeä - vain esimerkki!
 */

int main(int argc, char** argv){
  long long int lukuA, lukuB, lukuC;   /* C-mäinen esittely */
  lukuA = lukuB + lukuC;               /* Yhteenlasku vain  */
}
```

Tällainen tulee assembler-käännöksestä:

```
        .file   "pikkuohjelma.c"
        .text
.globl main
        .type   main, @function
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -36(%rbp)
        movq    %rsi, -48(%rbp)
        movq    -8(%rbp), %rax
        addq    -16(%rbp), %rax
        movq    %rax, -24(%rbp)
        leave
        ret
```

Mukana on joitakin Assembler-työkalun syntaksin mukaisia määreitä (kuten `.file`, alkavat pisteellä). Sitten on muistipaikan symbolinen nimi (`main:`) ja sitten on varsinaisia x86-64 -käskyjä assemblerilla kuvattuna. Notaatio `-36(%rbp)` tarkoittaa arvoa, joka löytyy muistipaikasta `rbp - 36`. Käsky:

```
movl    %edi, -36(%rbp)
```

siirtää rekisterissä EDI olevat bitit tuohon kyseiseen muistipaikkaan. Jotta ymmärretään, miksi juuri nämä käskyt, rekisterinimet ja ihmeelliseltä vaikuttavat miinusmerkkiset numerot syntyivät, pitää ottaa vielä vähän yleistä teoriaa taustalle. Se tulee heti seuraavassa luvussa.

Alla on vielä esimerkin vuoksi käännetystä ohjelmasta takaisin päin tehty "disassembly", joka näyttää opkoodit (siis konekielen) ja sen perusteella arvaillun assembler-koodin. Huomaa, miten `-36` näkyy sen 64-bittisessä kahden komplementtiesityksessä heksadesimaaleiksi koodattuna eli `0xffffffffffffffdc`. Se on sama kuin negatiivinen kokonaisluku -36:

```
400468:       55                        push   %rbp
400469:       48 89 e5                  mov    %rsp,%rbp
```

```
40046c:       89 7d dc                mov    %edi,0xffffffffffffffdc(%rbp)
40046f:       48 89 75 d0             mov    %rsi,0xffffffffffffffd0(%rbp)
400473:       48 8b 45 f8             mov    0xfffffffffffffff8(%rbp),%rax
400477:       48 03 45 f0             add    0xfffffffffffffff0(%rbp),%rax
40047b:       48 89 45 e8             mov    %rax,0xffffffffffffffe8(%rbp)
40047f:       c9                      leaveq
400480:       c3                      retq
```

Okei; katsotaan lisää esimerkkejä demossa 4 ja jokainen näkee sitten harjoitustyössään itse tehdyn C-ohjelman assembler-käännöksen.

9 Koodi, tieto ja suorituspino; osoittimen käsite

Luodaan katsaus siihen, miten tietokoneen muistin ja prosessorin yhteispeli oikein tapahtuu. Ohjelmaan kuuluu selvästi konekielikäskyjä prosessorin suoritettavaksi. Sanotaan, että tämä on ohjelman **koodi** (code). Lisäksi ohjelmissa on usein jotakin ennakkoon tunnettua tai globaalia **dataa** (*data*) ja vielä **paikallisia muuttujia** useisiin väliaikaisiin tarkoituksiin. Tämä lienee selvää.

Mainitut koodi ja data ladataan usein eri paikkoihin tietokoneen muistissa, ja paikallisille muuttujille varataan vielä ihan oma alue, jonka nimi on **suorituspino** (*stack*). Ohjelman tarvitseman muistin jako koodiin, dataan ja pinoon on perusteltua ja selkeätä ohjelmoijan kannalta; ovathan nuo selvästi eri tarkoituksiin käytettäviä ja erilaista dataa sisältäviä kokonaisuuksia. Lisäksi olisi mukavaa, jos voitaisiin saada tuplavarmistuksia ja turvallisuutta siitä, että data- tai pinoalueelle ei voitaisi koskaan vahingossakaan hypätä suorittamaan niiden bittejä ikään kuin ne olisivat ohjelmakoodia. Pahimmassa tapauksessa pahantahtoinen käyttäjä saisi sijoitettua sinne jotakin haitallista koodia... haluttaisiin myös, että koodialueelle ei vahingossakaan voisi kirjoittaa mitään uutta, vaan siellä sijaitsisi muuttumattomassa tilassa aikoinaan käännetty konekielinen ohjelma.

Useimmissa prosessoriarkkitehtuureissa on erilliset rekisterit, joiden tarkoitus on pitää yllä erillistä muistiosoitetta johonkin kohtaan juuri tiettyä muistialuetta. Ne voidaan myös sijoittaa ns. eri segmentteihin (segmentit voidaan sijoitella eri puolille fyysistä muistia, ja niille voidaan laitteistotasolla määritellä luku/kirjoitus/suoritus -oikeuksia). Luentomoniste antaa esimerkin 8086-toteutuksesta, tämä lehdykkä puolestaan antaa esimerkkejä x86-64-toteutuksesta, ja muillakin prosessoriarkkitehtuureilla on ihan vastaavat käytännöt hyvin pienillä eroilla.

Esimerkkimme x86-64:ssä ei ole käytössä erillisiä segmenttirekisterejä, vaan kaikki muistialueet näkyvät lineaarisen osoiteavaruuden eri lukualueina. Rekistereistä `RIP` osoittaa koodialueelle, kuten arvata saattaa. `RSP` eli pinon huipun osoitin puolestaan pinoalueelle ja yleiskäyttöiset rekisterit voivat osoittaa tarvittaessa data-alueelle. Pinoalue on usein (myös x86-64:ssä) organisoitu ovelasti siten, että pinon "pohja" eli ensimmäisenä tallennettu data on suurimmassa muistiosoitteessa, ja pinoon lisätään dataa sillä tavoin, että "huipun" osoitteesta `SP` vähennetään ensin datan vaatima tavumäärä ja sitten siirretään uusi pinottava data siihen osoitteeseen.

Huomaa, että prosessorin kannalta dataa ei ole missään "nimetyissä muuttujissa" kuten lähdekoodin kannalta, vaan kaikki käsiteltävissä oleva data on rekistereissä, tai se pitää noutaa muistiosoitteen perusteella koodi-, data- tai pinoalueelta (tai dynaamisesti varatulta tai useiden prosessien kesken jaetulta muistialueelta). Muistiosoite on vain numero; useimmiten osoite otetaan jonkun rekisterin arvosta (ns. *register indirect* eli rekisterin kautta tapahtuva epäsuora osoitus), esim. pino-osoitin ja käskyosoiterekisteri ovat aina muistiosoitteita; tai osoite voidaan laskea suhteellisena rekisterissä olevaan osoitteeseen nähden (tapahtuu rekisterin ja käskyyn koodatun vakion yhteenlasku ennen kuin osoite on lopullinen, ns. *base plus offset* eli epäsuora osoitus "kantarekisterin" ja siirrososoitteen avulla). Lisäksi voi olla mahdollista laskea yhteen kahden eri rekisterin arvot (jolloin toinen rekisteri voi olla "kanta" joka osoittaa esim. tietorakenteen alkuun ja toinen rekisteri "siirros" jolle voidaan laskea tarpeen mukaan eri arvoja; näin voidaan osoittaa tietorakenteen kuten taulukon eri osia).

Operaatioiden tulokset pitää tietysti erikseen viedä muistiin osoitteen perusteella vastaavasti. Kääntäjäohjelman tehtävänä on muodostaa numeerinen muoto osoitteille, joissa lähdekoodin kuvaamaa dataa säilytetään.

Assemblerilla muistiosoitteiden käyttö voisi näyttää esim. seuraavalta:

```
movq  $13, %(rbp)      # register indirect
movq  $13, -16%(rbp)   # base plus offset
```

C-ohjelmassa muistiosoitteita voi käyttää tietyllä syntaksilla, esim.:

```
int luku = 2;        /* lokaali kokonaislukumuuttuja nimeltä luku*/
int *osoitin;        /* lokaali muistiosoitin kokonaislukuun */
osoitin = &luku;     /* otetaan luvun muistiosoite ja sijoitetaan se */
tulosta_osoitettu_luku(osoitin);
                     /* annetaan parametriksi muistiosoitin; aliohjelma
                        on tehty siten että se haluaa parametrina
                        osoittimen */
tulosta_luku(*osoitin);
                     /* annetaan parametriksi itse luku
                        eikä osoitetta; tähti on käänteinen et-merkille */
tulosta_osoitin(osoitin);
                     /* tässäkin annettaisiin parametriksi luku, mutta
                        kyseinen luku olisi muistiosoite.*/
lisaa_yksi_osoitettuun_lukuun(osoitin);
                     /* Tällä voitaisiin vaikuttaa paikallisen muuttujan
                        "luku" arvoon */
lisaa_yksi_lukuun(luku);
                     /* Tällä ei tekisi mitään, jos tarkoitettu käyttö
                        olisi seuraavanlainen eikä parametri siis olisi
                        osoitin vaan primitiivimuuttuja: */
luku = lisaa_yksi_lukuun(luku);
                     /* Tällä siis sijoitettaisiin paluuarvo. */
```

Java-ohjelmassa jokainen viitemuuttuja on tavallaan "muistiosoite" olioinstanssiin kekomuistissa. Tai vähintäänkin sitä voidaan abstraktisti ajatella sellaisena. Esimerkki:

```
NirsoKapstyykki muuttujaA, muuttujaB, muuttujaC;
muuttujaA = new(NirsoKapstyykki(57));   /* instantoidaan */
muuttujaB = new(NirsoKapstyykki(57));   /* instantoidaan samanlainen*/
muuttujaC = muuttujaA;                  /* sijoitetaan */

tulosta_totuusarvo(muuttujaA == muuttujaB);  /* false */
tulosta_totuusarvo(muuttujaA == muuttujaC);  /* true  */
tulosta_totuusarvo(muuttujaA.equals(muuttujaB)); /* true, mikäli
                                 NirsoKapstyykki toimii siten kuin oletan
                                 sen toimivan.. */
```

Ylläoleva Java-esimerkki pitäisi olla erittäin hyvin selkärangassasi, jos voit sanoa osaavasi ohjelmoida! Ja jos ei se vielä ole, voit ymmärtää asian yhtä aikaa kun ymmärrät muistiosoitteetkin (ja tulla askeleen lähemmäksi ohjelmointitaitoa): Esimerkissä `muuttujaA`, `muuttujaB` ja `muuttujaC` ovat *viitemuuttujia*, virtuaalikoneen sisäisessä toteutuksessa ehkäpä kokonaislukuja jotka ovat indeksejä johonkin oliotaulukkoon tai muuhun. Viite tottakai eroaa muistiosoittimesta siinä, että se on vähän abstraktimpi käsite, eli se voisi olla jotain muutakin kuin kokonaisluku eikä ohjelmoijan tarvitse eikä pidä välittää niin kovin paljon siitä toteutuksesta... Kuitenkin, kun yllä ensinnäkin instantoidaan kaksi kertaa samalla tavoin `NirsoKapstyykki` ja sijoitetaan viitteet muuttujiin `muuttujaA` ja `muuttujaB`, niin lopputuloksena on kaksi erillistä vaikkakin samalla tavoin luotua oliota. Kumpaiseenkin yksilöön on erillinen viite (sisäisenä toteutuksena esim. eri kokonaisluku). Sijoitus `muuttujaC = muuttujaA` on nyt se, minkä merkitys pitää ymmärtää syvällisesti: Siinä sijoitetaan viite muuttujasta toiseen. Sen jälkeen *viitemuuttujat*

`muuttujaA` ja `muuttujaC` ovat edelleen selvästi eri yksilöitä; nehän ovat Java-virtuaalikoneen suorituspinossa eri kohdissa ja niille on oma tila sieltä varattuna. Mutta se *olioinstanssi, johon ne viittaavat on yksi ja sama.* Eli sisäisen toteutuksen kannalta näyttäisi esimerkiksi siltä, että pinossa on kaksi samaa kokonaislukua eli kaksi samaa "osoitinta" kekomuistiin. Sen sijaan `muuttujaB` on eri viite. Rautalankaesimerkkinä pinossa voisi olla seuraava sisältö:

```
muuttujaA : 57686
muuttujaB : 3422
muuttujaC : 57686
```

Niinpä esim. muuttujien vertailut operaattorilla ja metodilla antavat tulokset siten kuin yllä on kommenteissa. Puhh. Yritän tässä kertoa vielä kerran, että:

- sekä JVM että konkreettiset tietokoneprosessorit ovat tyhymiä vehkeitä, jotka tekevät peräkkäin yksinkertaisia suoritteita
- niissä on pinomuisti, koodialueita, dynaamisesti varattavia alueita
- näiden alueiden käyttö sekä rakenteisessa että olio-ohjelmoinnissa edellyttää "viite" nimisen asian toteutumista jollain tavoin, olipa toteutus sitten muistiosoite tai olioviite. Niiden toiminta ja ilmeneminen ovat monessa mielessä sama asia.

Ohjelmoinnin ymmärtäminen edellyttää abstraktin "viite"-käsitteen ymmärtämistä, missä voi ehkä auttaa että näkee kaksi erilaista ilmenemismuotoa (tai edes yhden) konepellin alla eli laitteistotasolla (Javan tapauksessa laitteisto on virtuaalinen, eli JVM).

10 Virtuaalimuisti ja osoitteenmuodostus

Moderneissa tietokoneissa konekielikäskyjen käsittelemät muistiosoitteet ovat ns. **virtuaaliosoitteita**: jokainen ohjelma näkee oman koodinsa, datansa ja pinonsa omassa muistiavaruudessaan peräkkäisissä muistiosoitteissa. Joissain arkkitehtuureissa voidaan kukin alue pitää omana segmenttinään, puhutaan **segmentoidusta muistista**. Tällöin esimerkiksi `IP`:lle mahdolliset osoitteet alkavat nollasta ja päättyvät osoitteeseen, joka vastaa jotakuinkin ohjelmakoodin pituutta tavuina; tämähän on selkeätä, kun koodi alkaa nollasta ja jatkuu lineaarisesti aina käsky kerrallaan. Puolestaan `SP`:lle mahdolliset osoitteet alkavat nollasta ja päättyvät osoitteeseen, joka vastaa pinolle varattua muistitilaa. Ohjelman alussa pino on tyhjä, ja `SP`:n arvo on suurin mahdollinen; sieltä se alkaa kasvaa alaspäin kohti nollaa. Myös data-alueen osoitteet alkavat nollasta. **Segmenttirekisterit** pitävät silloin huolen siitä, että pinon muistipaikka osoitteeltaan 52 on eri paikka kuin koodin muistipaikka osoitteeltaan 52. Kokonaisen virtuaalimuistiosoitteen pituus on silloin segmenttirekisterin bittien määrä yhdistettynä osoitinrekisterin pituuteen. Virtuaaliosoitteet olisivat siten esim. seuraavanlaisia:

```
CS (koodisegmenttirekisteri)    IP (käskyosoiterekisteri)
+----------------------------+  +------------------------------+
|          11000100          |  |       0000000000110100       |
+----------------------------+  +------------------------------+

Seuraava käsky noudettaisiin virtuaaliosoitteesta:
          11000100 0000000000110100

SS (pinosegmenttirekisteri)     SP (pino-osoitin)
+----------------------------+  +------------------------------+
|          00100000          |  |       0000000000110100       |
+----------------------------+  +------------------------------+

Seuraava pop-käsky noutaisi arvon virtuaaliosoitteesta:
          00100000 0000000000110100
```

Tämä on vaan hatusta vedetty esimerkki, jonka tarkoitus on näyttää, että IP ja SP voisivat segmentoidussa järjestelmässä olla samoja, mutta niiden tarkoittama virtuaaliosoite olisi eri, koska näihin liittyvät segmentit olisivat eri. Segmentin numero nimittäin kuuluu virtuaaliosoitteeseen.

Jätetään kuitenkin segmentoitu malli tuolle maininnan ja esimerkin tasolle. Esimerkkiarkkitehtuurimme x86-64 mahdollistaa segmentoinnin taaksepäin-yhteensopivuustilassa, koska siinä on haluttu pystyä suorittamaan x86-koodia, joka käytti segmenttejä. Kuitenkin 64-bittisessä toimintatilassa x86-64:n kullakin prosessilla on oma täysin lineaarinen muistiavaruutensa, joka käyttää 64-bittisen rekisterin 48 alinta bittiä muistiosoitteena. Loppujen bittien tulee olla samoja kuin varsinaisen osoitteen ylin bitti. Ei ole mitään segmenttejä ja segmenttirekisterejä. Koodi, pino ja tieto sijoittuvat kukin omaan alueeseensa 48-bittisessä virtuaaliosoiteavaruudessa ja suojaus toimii sivukohtaisesti.

x86-64:ssä suoritettavan käyttäjän prosessin (eli ohjelman) näkemä 48-bittinen virtuaaliosoiteavaruus on jotakin seuraavanlaista; sori, mutta en ehtinyt varmistaa tätä vielä ihan täysin. Jotakuinkin näin se kuitenkin menee per prosessi, idea on varmasti tämän kaltainen:

```
Muistiosoite:  Sisältö:
         2^48 ======================


                 kartoittamatonta


                   + dynaamisia
                     alueita


         2^32 ----------------------
              Pinoalue

              ----------------------

                 kartoittamatonta

              ----------------------
              Data-alue

              ----------------------

                 kartoittamatonta

              ----------------------
              Koodialue
         2^23
              ----------------------

                pohjalla paljon
                kartoittamattomia
                osoitteita; tarkoitus
                napata vääriä,
                nollaan eli
                NULL-pointteriin
                tehtyjä viittauksia.

            0 ======================
```

Olipa kyse segmentoidusta tai segmentoimattomasta virtuaaliosoiteavaruudesta, selkeän, lineaarisen (eli peräkkäisistä muistiosoitteista koostuvan) osoiteavaruuden toteutuminen on saavutus, joka helpottaa ohjelmien ja kääntäjien tekemistä kummasti. Muistanet toisaalta, että väylän takana oleva keskusmuisti sekä I/O -laitteet ym. ovat saavutettavissa vain fyysisen, osoiteväylään koodattavan muistiosoitteen kautta. Syystä tai toisesta aiheutuva keskeytys aiheuttaa prosessorin siirtymisen käyttöjärjestelmätilaan, ja tällöin se siirtyy myös käsittelemään fyysistä muistiavaruutta (ilman rajoituksia ja tuplavarmistuksia, jotka estävät normaaleja prosesseja koskemasta muihin kuin omaan alueeseensa). Fyysiset osoitteet kartoittuvat esimerkiksi seuraavasti (tässäkin on pieni epävarmuustekijä, mutta idea on varmasti näin). Fyysisen väylän leveys riippuu varsinaisesta prosessorimallista, meidän Jalavan Intel Xeonissa se on näköjään 36 bittiä; muistin kartoittuminen on jotakin tämän kaltaista:

```
36-bittisen muistiavaruuden kartoittuminen väylässä
(laitteistotasolla):


          2^36 ======================
                                         RAM-muistin loppu
                Käyttöjärjestelmän
                kiinteästi sijoitettu
                osuus
               ----------------------

               ======================
                Kehys 1                  Sivukehykset
               ----------------------    (RAM-muistia)
                Kehys 2
               ----------------------
                Kehys 3
               ----------------------
                ...
               ----------------------

                ...

               ----------------------
                Kehys N
               ======================    .. RAM-muistin alku


                 Paljon
                 kartoittumattomia
                 osoitteita myös...
                 koska fyysistä
                 muistia on paljon
                 vähemmän kuin
                 mahdollisia
                 osoitteita

               ======================
                 ROM-muisti
                 (käynnistysohjelma)
               ======================



               ======================
                 Laitteiden porttien
                 osoitteita ym.
               ======================


             0 ======================
```

Eli "todellinen" muistiavaruus sisältää johonkin kohtaan kartoitettuna fyysisten RAM-muistipiirien muistipaikkojen osoitteet. Johonkin kohtaan tätä muistia ladataan käyttöjärjestelmän koodi koneen käynnistyksen yhteydessä. Käyttöjärjestelmä ottaa lopun muistin hallintaansa. Se jakaa RAM-muistin osoitteiston **kehyksiin**; jokainen kehys on kiinteän mittainen. Jokaiseen kehykseen voidaan sijoittaa ns. **sivu** jonkun prosessin koodia, dataa, pinomuistia tai dynaamisesti varattua muistia. Sivu on kehyksen mittainen peräkkäisistä tavuista muodostuva kokonaisuus. Tässä alkaa hahmottua **sivuttavan virtualimuistin** idea: Prosessien tarvitsema muisti on jaettu sivuihin, joita voidaan pitää jossakin fyysisen muistin kehyksessä. Prosessi itse ei tiedä, missä kehyksessä (eli missä todellisen muistiavaruuden osoitteessa) sen tarvitsema tavu on, vaan se näkee lineaarisen osoiteavaruuden. Käyttöjärjestelmä sen sijaan hallitsee sivuja siten, että se voi sijoittaa niitä prosessien käytettäväksi fyysisen RAM-muistin kehyksiin, tai se voi tarpeen mukaan heittää niitä kovalevylle talteen. Jokaisen prosessin jokaisesta muistisivusta on kovalevyllä tallessa kopio, ja silloin kun niiden tarvitsee, yleensä vähän aikaa kerrallaan, käyttää jotakin sivua, käyttöjärjestelmä ottaa sen fyysiseen RAM-muistiin ns. työskentelykopioksi (*working set*). Kun prosessi ei taas vähään aikaan ole tarvinnut jotakin sivuaan, se voidaan tallentaa kovalevylle. Virtuaalimuistia voi näin olla enemmän kuin fyysistä RAM-muistia. Sivutus on erillinen käsite segmentoinnista: *puhtaasti segmentoiva muistinhallinta* käsittelisi vaihtelevan mittaisia segmenttejä samoin kuin sivuttava muistinhallinta käsittelee sivuja. Sellainen alkaa olla täysin historiaa. *Puhtaasti sivuttava muistinhallinta* käsittelee yksinomaan sivuja. *Segmentoiva ja sivuttava* muistinhallinta käsittelee kiinteän mittaisia sivuja: jokainen segmentti jaetaan tällöin erikseen sivuihin. Eli ero on varsin pieni; joka tapauksessahan koodi jaetaan loogisessa mielessä alueisiin, olivatpa ne osia lineaarisesta osoiteavaruudesta tai olivatpa ne segmenttirekisterien avulla erotettuja.

Prosessoreissa on oltava ominaisuus, joka muuntaa ohjelman virtuaaliset osoitteet todellisiksi osoitteiksi (*real address*) lennosta, aina kun se suorittaa käskyn, joka käyttää muistia. Prosessori tekee tällöin aina **osoitteenmuodostuksen**, eli muunnoksen virtuaalisesta todelliseksi. Ja käskyn noudon yhteydessähän tapahtuu aina osoitteenmuodostus, koska RIP:ssä on virtuaaliosoite. Osoitteenmuodostukseen prosessori tarvitsee käyttöjärjestelmän apua, ja yksi nykyaikaisen käyttöjärjestelmän tärkeä tehtävä onkin **virtuaalimuistin hallinta**, jonka toteutusperiaatetta tällä kurssilla käsitellään. Tämä osuus on kuvattu luentomonisteessa yleisellä tasolla, ja siinä laajuudessa joka tulee tenttiin. Katsotaan tässä täsmäesimerkki osoitteenmuodostuksesta x86-64:n tapauksessa. Arkkitehtuuri tukee muitakin sivukokoja, mutta tässä on esimerkki 4096 tavun (2^12) eli neljän kilotavun kokoisten sivujen käytöstä:

```
Virtuaaliosoite, eli vaikkapa jonkun käyttäjän prosessin RSP:n arvo.

Ylimmät bitit kuuluu olla samoja kuin ylin 48:sta käytetystä bitistä
|
|       Sivukartan indeksointi
|       |
|       |         Sivuhakemistojen hakemiston indeksointi
|       |         |
|       |         |         Sivuhakemiston indeksointi
|       |         |         |
|       |         |         |         Sivutaulun indeksointi
|       |         |         |         |
|       |         |         |         |         12-bittinen
|       |         |         |         |         osoite sivun
|       |         |         |         |         sisällä
|       |         |         |         |         |
|       |         |         |         |         |
63-48   47-39     38-30     29-21     20-12     11-0
11...1  111111111 100000000 000000001 010000000 001010000111
```

Huomataan, että x86-64:ssä virtuaaliosoite jakautuu hierarkkisesti neljään 9-bittiseen indeksiin, joiden perusteella haetaan varsinainen sivutaulu. Muistissa on siis useita taulukoita, joissa on osoittimet aina seuraavaan alemman tason taulukkoon. Kussakin taulukossa on 512 muistiosoitetta (eli 2^9 kpl)

aina seuraavan tason taulukoiden alkuosoitteisiin. Viimeinen taulu on varsinainen sivutaulu, josta löytyy mm. fyysisen sivukehyksen ensimmäisen muistipaikan osoite. *Jokaisen muistiviittauksen kohdalla* x86-64 -prosessori määrittää tämän nelikerroksisen indeksointihierarkian mukaisesti fyysisen muistiosoitteen. Käytännössä tehokas välimuistijärjestelmä mahdollistaa sen, että muistinhallinnan RAMissa sijaitseviin taulukoihin tarvitsee viitata harvoin. Suurimman osan aikaa muistiosoite löytyy "elektronin silmän räpäyksessä" TLB-nimisen erikoismekanismin kautta (TLB=*translation look-aside buffer*). Tämä perustuu tosiasiaan, että useimmissa ohjelmissa muistiviittaukset tapahtuvat suurimman osan aikaa lähes vierekkäisiin, tai ainakin samalla sivulla oleviin osoitteisiin, jolloin muunnostaulukko voidaan pitää ripeässä välimuistissa. Puhutaan "lokaalisuuden periaatteesta". TLB:stä tai ei, kuitenkin lopputulos edellisestä esimerkistä voisi olla seuraavanlainen:

```
Muistipaikan virtuaaliosoite (48 merkitsevää bittiä) on:
11...1  111111111  100000000  000000001  010000000  001010000111

--> Osoitteenmuodostus -->

Muistipaikan fyysinen osoite (36 merkitsevää bittiä) on:
11...               ...1  110100001011100011111011  001010000111
|                         |                         |
|                         |                         12-bittinen
Taas ekat bitit samoja    |                         osoite sivun
kuin eniten merkitsevä    |                         sisällä, SAMA
fyysisen osoiteavaruuden  |                         KUIN virtuaali-
osoitteen bitti.          |                         osoitteessa!
                          |
                          24 bittiä, jotka kertovat, että fyysisen
                          muistin kehyksen ensimmäinen tavu on
                          osoitteessa
                        11...110100001011100011111011  000000000000
                          ja viimeinen osoitteessa
                        11...110100001011100011111011  111111111111

                          Sillä ei ole mitään väliä tämän
                          muistiosoitteen kanssa, että kenelle jos
                          kenellekään kuuluu seuraavan kehyksen eka
                          fyysinen muistiosoite
                        11...110100001011100011111100  000000000000
```

Väylän osoitelinjalle asettuu fyysinen osoite. Ja jos sivun sisältö ei ollut missään kehyksessä, tulee **sivunvaihtovirhe**, joka on keskeytys, jonka käyttöjärjestelmän vastaava keskeytyksenkäsittelijä käsittelee tarkoituksenmukaisesti (eli hakee sivun sisällön kovalevyltä, ja tarvittaessa heittää ensin pois jonkun toisen sivun tiedot, jos kaikki kehykset ovat käytössä; ja sivu- ja kehystaulut on päivitettävä vastaavasti myös).

Huomaa tosiaan, että virtuaalinen osoiteavaruus tarkoittaa sitä, että kun ohjelmaa suoritetaan, sen *ei tarvitse yhtään välittää siitä, onko saman tietokoneen muistissa jossain kohtaa muitakin ohjelmia* tai ovatko virtuaalimuistin sivut lähelläkään toisiaan fyysisessä muistissa. Sen ei tarvitse välittää, onko sen tarvitsema tavu fyysisessä muistissa vai onko se vielä kovalevyllä jemmaan heitettynä. Itse asiassa käyttäjän prosessi ei millään tavoin näe fyysisiä osoitteita; se näkee vain virtuaaliosoitteet. Käyttöjärjestelmän muistinhallinta hoitaa asian yhteistyössä prosessorin kanssa.

HUOM: Ylläkuvattu hakemistotaulujen hierarkia on kuvattu prosessoriarkkitehtuurin dokumentaatiossa, ja kyseiselle prosessorille tehdyn käyttöjärjestelmän täytyy pitää sellaista yllä.

11 Aliohjelman suoritus (== ohjelman suoritus)

Käännös- ja ajokelpoinen C-ohjelma kirjoitetaan aina `main`-nimiseen funktioon, jolla on tietynlainen parametrilista. Käytännössä kääntäjän luoma alustuskoodi kutsuu sitä tosiaan ihan tavallisena aliohjelmana. Ei siis oikeastaan tarvitse tehdä mitään erottelua pää- ja aliohjelman välille prosessorin ja suorituksen näkökulmasta. Minkä tahansa ohjelman suoritusta voidaan ajatella sarjana seuraavista:

- peräkkäisiä käskysuorituksia
- ehdollisia ja ehdottomia hyppyjä `IP`:n arvosta toiseen
- aliohjelma-aktivaatioita.

Sama pätee Javassa: Ohjelma alkaa siten, että joku kutsuu julkista, `main`-nimistä luokkametodia. Aina ollaan suorittamassa jotakin metodia, kunnes ohjelma jostain syystä päättyy.

11.1 Lyhyesti aliohjelmista ja metodeista

Aliohjelman käsite jollain tasolla lienee tuttu kaikille -- olihan "ohjelmointitaito" tämän kurssin esitietovaatimus. Jos ei ole tuttu, niin assembler-ohjelmoinnin kautta varmasti tulee tutuksi, kun alat ymmärtää, miten prosessori suorittaa niitä. Vähintään 60 vuotta vanha käsite **aliohjelma** (*subprogram, subroutine*), joskus nimeltään **funktio** (*function*) tai **proseduuri** (*procedure*) ilmenee ohjelmointiparadigmasta riippuen eri tavoin:

- funktio-ohjelmoinnissa funktiot muodostavat puurakenteen, jonka lehtisolmuista lähtee määräytymään pääfunktion ("juurisolmun" eli koko ohjelman) tulos. Tai sinne päin; en ole ihan asiantuntija; käykää halutessanne kurssi nimeltä Funktio-ohjelmointi, jossa ihminen kuulemma valaistuu lopullisesti.
- imperatiivisessa ohjelmoinnissa aliohjelman avulla halutaan suorittaa jollekin datalle joku toimenpide. Aliohjelmaa kutsutaan siten, että sille annetaan mahdollisesti parametreja, minkä jälkeen kontrolli siirretään aliohjelman koodille, joka operoi dataa jollain tavoin, muuttaa mahdollisesti datan tilaa ja muodostaa mahdollisesti paluuarvoja.
- olio-ohjelmoinnissa olioinstanssille annetaan viesti, että sen pitää operoida itseään tietyllä tavoin joidenkin tarkentavien parametrien mukaisesti. Käytännössä olion luokassa täytyy olla toteutettuna viestiä vastaava metodi eli "aliohjelma", joka saa mahdolliset parametrit, muuttaa mahdollisesti olion sisäistä tilaa, ja palauttaa mahdollisesti paluuarvoja.

Ensiksi mainittuun funktio-ohjelmointiin ei tällä kurssilla kajota, mutta imperatiivisen ja olio-ohjelmoinnin näkökulmille aliohjelman käsitteestä pitäisi löytää yhteys. Olion instanssimetodin kutsu voidaan ajatella siten, että ikään kuin olisi olioluokkaan kuuluvien olioiden sisäistä dataa varten rakennettu aliohjelma, jolle annetaan tiedoksi (yhtenä parametrina) viite nimenomaiseen olioinstanssiin, jolle sen tulee operoida. Jotenkin näin se toteutuksen tasolla tapahtuukin, vaikkei sitä esim. Javan syntaksista huomaa. Luokkametodin kutsu taas on sellaisenaankin hyvin lähellä imperatiivisen aliohjelman käsitettä, koska pelissä ei tarvitse olla mukanaa yhtään olioinstanssia.

Java-ohjelma ilman yhtään olion käyttöä (so. primitiivityyppisille muuttujille) pelkkiä luokkametodeja käyttäen vastaa täysin C-ohjelmointia ilman datastruktuurien (tai taulukoidenkaan) käyttöä. Se on "pienin yhteinen nimittäjä", jolla tavoin ei kummallakaan kielellä tietysti kummoisempaa ilotulitusta pysty toteuttamaan. Ilotulitukset tehdään Javassa luomalla olioita ja C:ssä luomalla tietorakenteita sekä operoimalla niille, Javassa suorittamalla metodeja ja C:ssä suorittamalla aliohjelmia. Sekä olioista että tietorakenteista käytetään englanniksi joskus nimeä *object*, objekti.

11.2 Aliohjelma-aktivaatio (eli kutsu) prosessorin toimenpiteenä

Nyt haetaan toteutuksesta ja arkkitehtuurista riippumattomia linjoja. "Ohjelman suoritus konekielitasolla" on tämän kurssin yksi oppimistavoite. Ohjelman suoritus Java virtuaalikoneen eli JVM:n kone-

kielitasolla on samankaltaista kuin ohjelman suoritus x86-64:n konekielitasolla tai kännykässä olevan ARM-prosessorin konekielitasolla.

Ymmärretään toivottavasti, että jos kerran jokainen ohjelma on aliohjelma (tai yhtä hyvin metodi), niin ohjelmaa suoritettaessa ollaan suorittamassa aina aliohjelmaa. Aliohjelmalla taas pitää olla mahdollisuus seuraaviin ominaisuuksiin:

- se on saanut jostakin parametreja; ne pitää nähdä muuttujina aliohjelmassa, jotta niihin pääsee käsiksi
- se tarvitsee suorituksensa aikana paikallisia muuttujia
- sen pitää pystyä palauttamaan tietoja kutsujalleen
- sen pitää pystyä kutsumaan muita aliohjelmia.

Aliohjelmat (eli ohjelmat...) suoritetaan normaalisti käyttämällä kaikkeen ylläolevaan suorituspinoa (lineaarinen nollasta alkava muistialue, joka useimmiten täyttyy ovelasti osoitemielessä alaspäin). Yksi varsin siisti tapa hoitaa asia on käyttää aina (ali)ohjelman suoritukseen perinteistä käsitettä **pinokehys** (*stack frame*) -- toinen nimi tälle on **aktivaatiotietue** (*activation record*). Rakenteen käyttöön tarvitaan pinoalue ja kaksi rekisteriä, jotka osoittavat pinoalueelle. Toinen on pinon huipun osoitin (`SP`), ja toinen pinokehyksen/aktivaatiotietueen kantaosoitin (joskus `BP`, *base pointer*).

Idea on seuraavanlainen. Aliohjelmassa oltaessa pinon huippuosa (pienimmät muistiosoitteet) sisältää seuraavaa:

```
.                  ...
         |                      |
         |   (kutsuvan aliohjelman
         |    paikalliset muut- |
         |    tujat)            |
         |                      |
         |    -------------     |
         |----------------------|
N+2W(%bp)| Parametrien tavu N   |
         |                      |
         |                      |
         |      ...             |      edellinen kehys
         |                      |
         |                      |
  2W(%bp)| Parametrien tavu 0   |
---------|----------------------|----------------------
         |                      |
         | Paluuosoite          |
   W(%bp)|                      |
         |----------------------|
         |                      |
         | Edellisen kehyksen   |
   0(%bp)| kantaosoite          |
         |----------------------|      nykyinen kehys
  -Z(%bp)| paikallinen muuttuja A|
         |                      |
         |   (tämän aliohjelman |
         |    paikalliset muut- |
         |    tujat)            |
         |                      |
   0(%sp)|    -------------     |


   W on osoitteen leveys, esim. 8086:ssa 4 tavua, AMD64:ssä 8 tavua

   Z on paikallisen muuttujan "A" leveys.
```

Nyt, kun suoritus on nykyisessä kehyksessä ja rekisterit `BP` ja `SP` on asetettu oikein, pätee seuraavaa:

- Parametrit ovat pinoalueella muistissa. Niiden muistipaikkojen osoitteet saadaan lisäämällä kantaosoitteeseen `BP` sopivat arvot; yleensä prosessorikäskyt mahdollistavat tällaisen osoitusmuodon eli "rekisteri+lisäindeksi". Parametrien arvot saadaan esim. laskutoimituksia varten rekistereihin siirtokäskyillä, joissa osoite tehdään tällä tavoin indeksoimalla.
- Paluuosoite on tallessa tietyssä kohtaa pinokehystä.
- Edellisen aliohjelma-aktivaation käyttämä kantaosoitin `BP` on tallessa tietyssä kohtaa pinokehystä. Huomaa, että pinokehyksiä voidaan ajatella linkitettynä listana: Jokaisesta on linkki kutsuvan aliohjelman kehykseen ja sen lisäksi oman aktivaation parametrit sekä lokaalit muuttujat. Rakenteesta käytetään usein nimeä kutsupino (*call stack*).

- Paikallisia muuttujia voidaan varailla ja vapauttaa tarpeen mukaan pinosta ja `SP` voi rauhassa elää `PUSH` ja `POP` -käskyjen mukaisesti tai uuden aliohjelma-aktivaation tekemiseen.

Homma toimii siis aliohjelman sisällä, vieläpä siten, että on tallessa tarvittavat tiedot palaamiselle aiempaan aliohjelmaan. Miten sitten tähän tilanteeseen päästään, eli miten aliohjelman kutsuminen (aktivointi) tapahtuu konekielisen käännöksen ja prosessorin yhteispelinä? Prosessorin käskyt tarjoavat siihen apuja, ja hyvätapaisen ohjelmoijan assembler-ohjelma tai C-kääntäjän tulostama konekielikoodi osaavat hyödyntää käskyjä oikein. Tyypillisesti kutsumisen yhteydessä luodaan uusi pinokehys seuraavalla tavoin:

- kutsujan käskyt laittavat parametrit pinoon käänteisessä järjestyksessä (lähdekoodissa ensimmäiseksi kirjoitettu parametri laitetaan viimeisenä pinoon) juuri ennen aliohjelmakutsun suorittamista.
- Yleensä prosessori toimii siten, että `CALL` -käsky tai vastaava, joka vie aliohjelmaan, toteuttaa seuraavan käskyn osoitteen tallentamisen `IP`:n sijasta pinon huipulle. `IP`:hen puolestaan sijoittuu aliohjelman ensimmäisen käskyn osoite.
- Seuraavassa prosessorin *fetch* -toimenpiteessä tapahtuu varsinaisesti suorituksen siirtyminen aliohjelmaan. Sanotaan, että kontrolli siirtyy aliohjelmalle; ehkä siksi, että kyseinen aliohjelma kontrolloi eli hallitsee prosessorin suoritusta.
- Aliohjelman ensimmäisen käskyn pitäisi ensinnäkin painaa nykyinen `BP` eli juuri äsken odottelemaan jääneen aktivaation kantaosoitin pinoon.
- Sen jälkeen pitäisi ottaa `BP`-rekisteri tämän uuden, juuri alkaneen aktivaation käyttöön. Kun siihen siirtää nykyisen `SP`:n, eli pinon huippuosoitteen, niin se menee juuri niin kuin pitikin, ja ylläolevassa kuvassa oli esitelty.
- Ja siten `SP` vapautuu normaaliin pinokäyttöön.

Kuten edellisessä osiossa nähtiin, pinokehyksen käyttöön on tarjolla jopa prosessorin käskykannan käskyt, x86-64:ssä ENTER ja LEAVE, joilla pinokehyksen varaaminen ja vapauttaminen voidaan kätevästi tehdä.

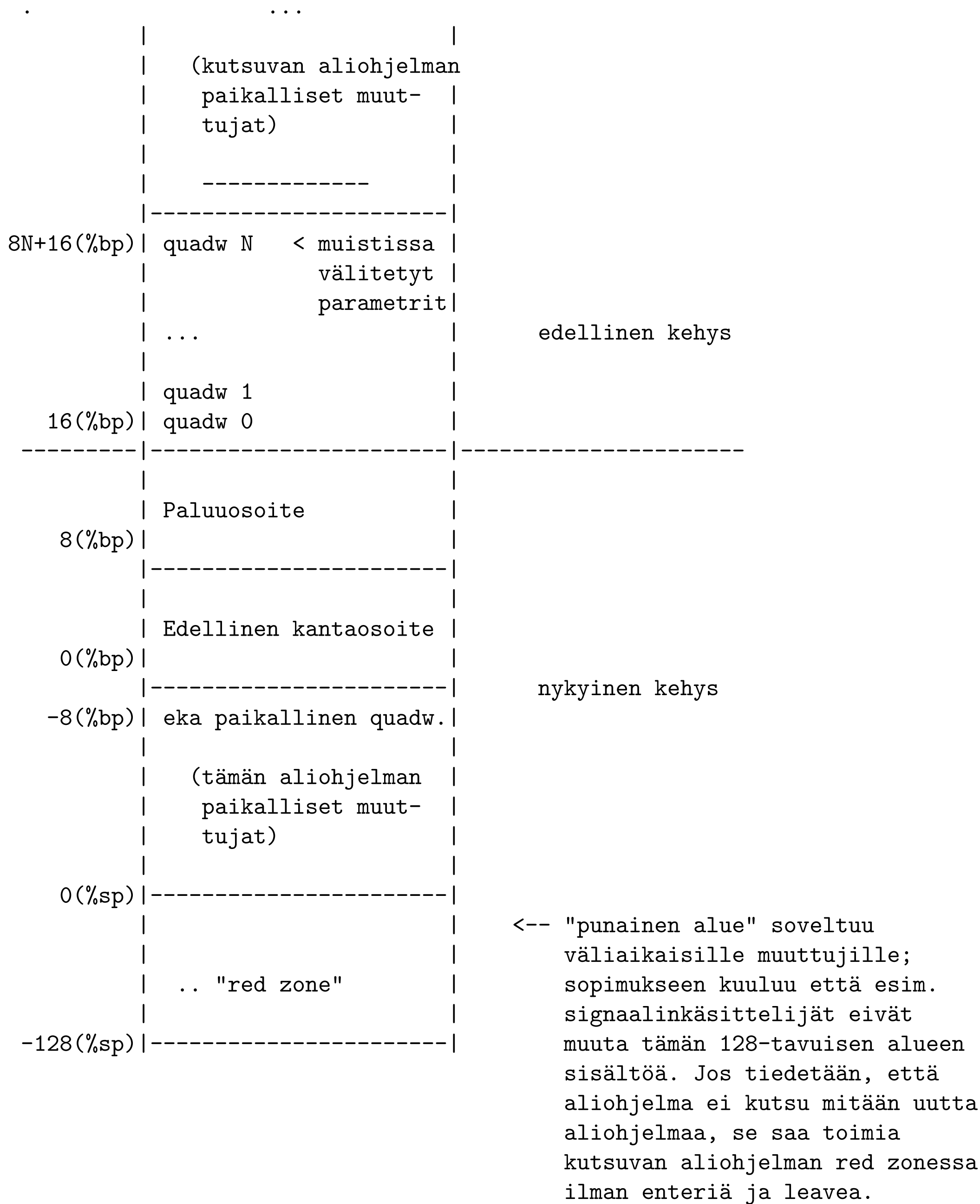
11.3 Moderni laajennos: System V ABIn C-kutsumalli x86-64:lle

Knoppeja: System V on 1980-luvulla tehty versio Unixista. Sitä voidaan pitää eräänlaisena standardina myöhempien Unix-variaatioiden tekemiselle, erityisesti sen versiota 4.0, jota sanotaan SVR4:ksi. **ABI** eli *Application Binary Interface* on osa käyttöjärjestelmän määrittelyä; se kertoo mm. miten käännetty ohjelmakoodi pitää sijoitella tiedostoon, ja miten se tullaan suoritettaessa lataamaan muistiin. ABI määrittelee myös, miten aliohjelmakutsu tulee toteuttaa. Tämän asian standardointi on tarpeen, jotta eri kirjoittajien tekemät ohjelmat voisivat tarvittaessa kutsua toistensa aliohjelmia. Erityisesti voidaan tehdä yleiskäyttöisiä valmiiksi käännettyjä aliohjelmakirjastoja (virtuaalikoneita). Tämä ns. **kutsumalli** (*calling convention*) määrittelee mm. parametrien ja paluuarvon välitysmekanismin. Malli voi vaihdella eri laitteistojen, käyttöjärjestelmien ja ohjelmointikielten välillä. Se on erittäin paljon sopimuskysymys. Siirrettävän ja yhteensopivan koodin tekeminen on vaikeaa, jos ei tiedä tätä asiaa, ja osaa varoa siihen liittyviä sudenkuoppia. Mikä on se kutsumalli, jonka mukaista konekieltä kääntäjäsi tuottaa? Voitko vaikuttaa siihen jollakin syntaksilla tai kääntäjän argumentilla? Minkä kutsumallin mukaisia kutsuja aliohjelmakirjastosi olettaa? Mitä teet, jos työkalusi ei ole yhteensopiva, mutta haluat ehdottomasti käyttää löytämääsi binääristä kirjastoa?

Edellä esitettiin perinteinen pinokehysmalli aliohjelman kutsumiseen. Nykyaikainen prosessoriteknologia mahdollistaa tehokkaamman parametrinvälityksen: idea on, että mahdollisimman paljon parametreja viedään prosessorin rekistereissä eikä pinomuistissa; rekisterien käyttö kun on reilusti nopeampaa

kuin väylän takana sijaitsevan pinomuistin. GNU-kääntäjä, jota Jalavassa käytämme tällä kurssilla toteuttaa kutsumallin, joka on määritelty dokumentaatiossa nimeltä “System V Application Binary Interface - AMD64 Architecture Processor Supplement”. Olen tiivistänyt tähän olennaisen kohdan em. dokumentin draftista, joka on päivätty 27.9.2006.

Pinokehys ilmenee tällä tavoin:




```
x86-64:ssä osoitteen pituus on 8 tavua; tässä kuvassa muuttujat
ovat kasitavuja eli quadwordeja.
```

Eli ihan samalta näyttää kuin yleinen pinokehysmalli. Kuitenkin nyt parametreja välitetään sekä muistissa että rekistereissä. Sääntöjä on useampia kuin tähän mahtuu, mutta todetaan, että esimerkiksi,

jos parametrina olisi pelkkiä 64-bittisiä kokonaislukuja, juuri aktivoitu aliohjelma olettaa, että kutsuja on sijoittanut ensimmäiset parametrit rekistereihin seuraavasti:

```
RDI == ensimmäinen integer-parametri
RSI == toinen integer-parametri
RDX == kolmas integer-parametri
RCX == neljäs integer-parametri
R8  == viides integer-parametri
R9  == kuudes integer-parametri
```

Jos välitettävänä on enemmän kokonaislukuja, ne menevät pinon kautta. Jos välitettävänä on rakenteita, joissa on tavuja enemmän kuin rekisteriin mahtuu, sellaiset laitetaan pinoon -- tai on siellä jotain muitakin sääntöjä, joiden mukaan parametri voidaan valita pinon kautta välitettäväksi vaikkei rekisterit olisi vielä sullottu täyteen.

Paluuarvoille on vastaava säännöstö. Todetaan, että jos paluuarvona on yksi kokonaisluku, niin se palautetaan RAX:ssä kuten x86:n C-kutsumallissa aina ennenkin.

Näillä eväillä pitäisi pystyä tekemään harjoitustyö. Yritän tehdä aiheet sellaisiksi, että eksoottisempia säännöstöjä ei tarvitsisi käyttää. Parametreina olisi joko 64-bittisiä kokonaislukuja tai muistiosoitteita, jolloin em. kuvaus on riittävä.

12 Liite: Entä jos keskeytyksiä ei olisi?

Edellä kerrottiin, mitä ovat keskeytykset. Ehkä pari sanaa kannattaisi kertoa siitä, miksi ne ovat. Jos keskeytyksiä ei olisi, moniajo ainakin ilmeisesti edellyttäisi sovellusten tekijöiltä jotakin tällaista säännöstöä:

- Ohjelmasi käynnistetään Käyttöjärjestelmän kautta siten, että main() -aliohjelmaa kutsutaan.
- kun main() -aliohjelmaasi on kutsuttu, se saa pyöriä korkeintaan 1000-1400 kellojakson ajan, jonka jälkeen ohjelman tulee tallentaa tiedot väliaikaisesti sinulle varattuun muistialueeseen ja palauttaa kontrolli Käyttöjärjestelmälle.
- main() -aliohjelmaa tullaan kutsumaan uudelleen sitten, kun kaikkien muiden ajossa olevien ohjelmien vastaavia on kutsuttu kertaalleen.

Ja sitten käyttöjärjestelmä pallottelisi kontrollia eri prosessien välillä aina määräajoin. (Paitsi jos jossain ohjelmassa olisi virhe, eikä kontrolli palaisikaan Käyttöjärjestelmälle... kaikki pysähtyisi ja se olisi ikävää...)

Näin se moniajo tehtiin ehkä joskus, mutta sitten ihminen keksi, että ei tämä käy, ja insinööri keksi, että mitä sitten tehdään. Tehtiin aikaviipaleet, kellokeskeytys ja automaattinen kontekstin vaihto!

Toinen perustelu keskeytyssysteemille on syöttö ja tulostus. Tunnetusti I/O -laitteet toimivat paljon hitaammin kuin prosessori, ja jotkut (kuten näppäimistö, hiiri) jopa käyttäjän mielivallan mukaisina ajanhetkinä. Niinpä interaktiiviset ohjelmat ja kovalevylle kirjoittavat/lukevat ohjelmat joutuisivat odottelemaan pitkiä aikoja ilman että mitään tapahtuisi. Esimerkiksi pseudo-assemblerilla:

```
odotus_silmukka:
    Tarkasta, onko näppäimistöllä painettu jotain
    Jos ei ollut, hyppää kohtaan odotus_silmukka
```

Tai näin:

```
luku_silmukka:
    Kerro kovalevylle, mistä kohtaa luetaan tavu

odotus_silmukka:
    Tarkasta, onko kovalevy lukenut jo
    Jos ei ollut, hyppää kohtaan odotus_silmukka

    Tallenna luettu tavu muistiin
    Jos ei ollut viimeinen merkki, hyppää kohtaan luku_silmukka
```

Ja selvästi tässä menisi hukkaan tuhansia tai miljoonia kellojaksoja, joiden aikana voisi hyvin suorittaa paljon kaikkea muuta, vaikka juuri se I/O:ta odottava ohjelma ei voisikaan edetä ennen kuin merkki saapuu. Päätteeltä tulevaa merkkiä saattaisi joutua odottamaan 12 tuntia, jos päätteen ääressä istuva nörtti on nukahtanut tai muuten vain ei paina mitään nappia. Kunnollinen moniajo **prosessorin koko kapasiteetin hyödyntämiseksi** on ollut tavoiteltava tilanne niin kauan kuin tietokoneella on voinut tehdä rahanarvoisia tuloksia ja toisaalta koneen hankinnasta ja käyttöajasta on ollut kustannuksia.

Joitakin tavoitetilanteita moniajossa:

- **Kapasiteetin hyödyntäminen**: jos ohjelma odottelee I/O-operaatiota, kuten että nörtti painaa näppäintä päätteellä, olisi hyvä että muut ohjelmat voisivat laskea sääennusteita tai dekoodata DVD-elokuvaa monitorille sillä välin.
- **Vastaaminen kaikkiin pyyntöihin**: jos joku ohjelma dekoodaa DVD-elokuvaa monitorille täyttä vauhtia ja käyttää melkein jokaisen kellojakson, olisi hyvä, että kuitenkin kun nörtti painaa näppäintä päätteellä, hänelle melko pian kaiutettaisiin painettu merkki takaisin printtinä, ettei rupea ihmettelemään, onko verkkoyhteys poikki.

- **Tasapuolisuus**: jos sekä Juha että Liisa laskevat serverillä paperikoneen nestevirtausmallia, olisi hyvä, että molemmille jaettaisiin noin 50 % suoritusajasta. Elleivätpä toisaalta halua eksplisiittisesti jakoa vaikkapa 25/75% ...
- **Reaaliaikavaatimukset** ja muut erityistarpeet: samalla kun pakkaan DVD:ltä kovalevylle dekoodattua informaatiota DivX:ksi, olisi hyvä pystyä kuuntelemaan MP3-musiikkia ilman, että ääni ratisee tai pätkii

Tavoitteet ovat usein keskenään ristiriitaisia, ja niiden toteutuksessa joudutaan painottamaan erilaisia asioita. Mm. vuoronnusperiaatteiden erilaisuuden vuoksi erilaiset käyttöjärjestelmät soveltuvat syvällisesti erilaisiin käyttötarkoituksiin.