Esimerkki 2000-luvun prosessoriarkkitehtuurista: AMD64 (x86-64)

[Ensimmäinen versio, jossa on puutteita ja virheitä. Jotakuinkin ehkä tulee toimeen tällä aluksi... korjauksin varustettu versio jaetaan monisteena ja netissä myöhemmin.]

Tämä lehdykkä korvaa kesän 2007 Käyttöjärjestelmät -kurssilla luentomonisteen sivut 6-21. Voi olla hyödyllistä lukea sama asia myös luentomonisteesta, koska viimeistään silloin huomaa, että samat perusasiat toteutuvat useissa prosessoreissa samalla tavoin. On myös niin, että tässä esiteltävä AMD64 -arkkitehtuuri on luentomonisteessa esitellyn Intel 8086 -arkkitehtuuri suora perillinen. 8086-konekieliset ohjelmat toimivat muuttamattomina AMD64:ssä, vaikka välissä on ollut useita prosessorisukupolvia teknisine harppauksineen. Huomatkaa, että Tietohallintokeskuksen kone `jalava.cc.jyu.fi`, jossa tehdään kurssin harjoituksia, on malliltaan kaksiytiminen AMD Opteron, jonka arkkitehtuuri on nimenomaan AMD64. Materiaalin päivitys mahdollistaa uskoakseni paremmat mahdollisuudet toteuttaa käytännön esimerkit tavalla, jonka kesäopettaja osaa tehdä.

Esitietoina edellytetään tietokonelaitteiston ymmärrys sillä tasolla, joka kesän luennoilla on tähän asti käsitelty (pieni osajoukko Tietotekniikan Perusteet -monisteen alkupään sisällöstä; sivunumerot annettu kurssin nettisivulla). Tässä annetaan käytännön esimerkki nykyaikaisesta prosessoriarkkitehtuurista. Vastaavia on markkinoilla monta, ja niissä on merkittäviä eroja, mutta kaikissa on jollakin tavoin toteutettu pakolliset piirteet: jokin joukko jonkinlaisia rekisterejä, jokin joukko mahdollisia konekielisiä käskyjä (eli **käskykanta**) ja jokin joukko sääntöjä, joiden mukaisesti konekieliohjelma pitää tehdä (operaatiokoodien muodostaminen, muistinosoitusmuodot, aliohjelmien kutsumiskäytännöt, ...). Tässä koetetaan myös jossain määrin esitellä ja valaista prosessorien ja konekielisen ohjelmoinnin yleistä käsitteistöä. Käytännön esimerkki sattuu olemaan AMD64, koska THK:ssä sattuu pyörimään sellainen nimeltä Jalava. Yhtä hyvin esimerkkinä voisi olla mikä tahansa, jolla olisi helppo pyöräytellä esimerkkejä.

AMD64:n taustaa: Prosessoriteknologiaan keskittyvä yritys nimeltä Intel on julkaissut mm. toisiaan seuranneet prosessorimallit (ja arkkitehtuurit) nimeltä 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 väylällä on 64 bittiä rivissä). Joitakin eroja on, mutta lähinnä niillä on merkitystä yhteensopivien kääntäjien valmistajille. Käytettäköön jatkossa siis arkkitehtuurien yhteisnimeä x86-64.

Tämän kirjoittamiseen on käytetty lähteenä seuraavaa dokumentaatiota:

- AMD64 Architecture home page
- Intel 64 Architecture home page

Linkit kesäkurssin kotisivulla luennon 4 kohdalla.

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 noudettavan käskyn osoite puolestaan on ohjelmoijalle näkyvässä rekisterissä, jonka nimi on **ohjelmalaskuri** (`PC`, *program counter*) **käskyosoitin** (`IP`, *instruction pointer*) **käskyosoiterekisteri**, tai vastaavaa. Sisäinen rekisteri `INSTR` tarvitaan yksinkertaisesti siksi, että sähköjännitteet säilyisivät jossakin sen aikaa kun kaikki käskyn suorituksessa tarvittavat tiedot saadaan noudettua väylältä -- nekin tietty rekistereihin. Käskyn suoritus tapahtuu sitten, kun kaikki on rekistereissä valmiina. Suoritus kestää sitten muutamia kellojaksoja, käskyn monimutkaisuudesta riippuen ehkä hyvinkin monta. Suorituksen tuloksena rekistereiden tila muuttuu; ainakin ohjelmalaskuri `PC` päivittyy seuraavan käskyn muistiosoitteeksi, minkä jälkeen prosessori noutaa käskyn `PC`:n osoittamasta paikasta, ja sykli jatkuu. Tätä nouto-suoritus -sykliä käsitellään kohta tarkemmin, ja siihen lisätään vielä lopulta ns. keskeytyskäsittely. Perusmuodossaan "fetch-execute" tarkoittaa tätä:

1. Prosessori noutaa dataa `PC`-rekisterin osoittamasta paikasta
   Noudettu data sijoitetaan sisäiseen `INSTR`-rekisteriin
2. Prosessori suorittaa käskyn
   Eli `INSTR`-rekisterissä oleva bittijono herättää kontrolliyksikössä toimenpiteitä, joihin syötetään myös muiden käskyssä tarvittavien rekisterien arvoja.
3. Käskyn suorituksen tuloksena rekisterien tila on muuttunut jollain tavoin
   Yksi, joka aina muuttuu, on `PC`. Miten `PC` muuttuu, riippuu suoritetusta käskystä:
   - Laskutoimitus, datan siirto tai muu peräkkäissuoritus ==> `PC` osoittaa seuraavaan konekielikäskyyn
   - Hyppykäsky ==> `PC` osoittaa käskyssä kerrottua uutta osoitetta, esim. silmukan alkua tms.
   - Ehdollinen hyppykäsky ==> `PC` osoittaa käskyssä kerrottua uutta osoitetta mikäli käskyssä kerrottu ehto toteutuu; muutoin osoittaa seuraavaan käskyyn
   - Aliohjelmakutsu ==> `PC` osoittaa käskyssä kerrottua uutta osoitetta (jonka tulee olla kutsuttavan aliohjelman ensimmäinen käsky; aliohjelmakutsussa prosessori tekee paljon muutakin, mitä käsitellään kohta tarkemmin)
   - Paluu aliohjelmasta ==> `PC` osoittaa taas siihen ohjelmaan, joka suoritti kutsun, erityisesti kyseessä on aliohjelmakutsua välittömästi seuraava käsky. (kohtapuoleen nähdään, miten prosessori noutaa paluuosoitteen pinomuistista)

Prosessorien manuaaleissa suoritussykli kuvataan laajemmin. Siinä on mukana nykyaikaisia hienouksia, kuten `INSTR`-käskyn muuntaminen ns. mikrokoodiksi eli *decode*-vaihe. (Puhutaankin usein *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 kaavioista etsii ylläolevaa perusmallia, se kyllä löytyy sieltä. Sovellusohjelmoijan 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 suorittaa vain sellaisia konekielikäskyjä, jotka käsittelevät **käyttäjän nähtävissä olevia rekistereitä** (*user-visible registers*).

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 Unixeissa käyttöjärjestelmä odottaa "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.

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 kiikkukomponentissa 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 voi jäädä muistibitti yli, jolloin nostetaan "carry flag" lippu -- se on joku bitti PSW:ssä, ja sen nimi on usein kirjallisuudessa `CF`
- Vähennyslaskussa ja vertailussa (joka on olennaisesti vähennyslasku!) 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). Ja 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 sekä käyttäjän että 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.

Prosessori on fyysiseltä kooltaan pieni -- sen pitää olla pieni, koska sähköisen signaalin nopeus on rajoitettu, ja mitä pidempiä matkoja sähkön pitää matkata, sen hitaampi 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 sähkönjohtimia pitkin, joten hyvin sähköä 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, joka herättää prosessoria jatkuvasti
- keskeytysilmoitukset
- maadoitus

Väylä, väylän ohjaus, keskusmuisti ja kaikki laitteet tosiaan ovat prosessorin koteloinnin ulkopuolella, jos puhutaan pöytäkoneista. 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.

Yleistä assemblereista ja notaatioista

Nykyään vain kääntäjäohjelmilla on järkeä tuottaa konekieltä bittijonona. (Toisenlaista oli esihistorian alussa, kun käännös todella tehtiin käsin ja 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", joka käännetään bittijonoksi **assemblerilla** eli symbolisen konekielen kääntäjällä. Käyttöjärjestelmistä (pienehkö) osa on kirjoitettava assemblerilla, joten tällä kurssilla ilmeisesti käsitellään sitä. Se on myös oiva apuväline prosessorin toiminnan ymmärtämiseksi. 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 `MOVQ`) olisi lyhenne sanoista "Move quadword". Sen merkitys olisi siirtää ("move") neljä sanaa ("quadword") eli 64 bittiä paikasta toiseen. Mistä mihin siirretään, annetaan **operandeina**, jotka tässä tapauksessa näyttäisivät x86-64:n määrittelemiltä rekistereiltä `rsp` ja `rbp` (AMD64:n dokumentaatiossa `RSP` ja `RBP`). Käskyillä on useimmiten nolla, yksi tai kaksi operandia (ja joka tapauksessa osa käskyn suorituksen syötteistä voi tulla muualtakin kuin operandeista -- esim. PSW:n biteistä tai jostain muistiosoitteesta; tietyn prosessoriarkkitehtuurin dokumentaation käskykanta-osuudessa kerrotaan aina hyvin täsmällisesti, mitkä kunkin käskyn kaikki syötteet, tulosteet ja sivuvaikutukset prosessorin 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`.

Prosenttimerkki `%` ylläolevassa puolestaan on riippumaton x86-64:stä; se on osa tässä käytettyä yleisempää assembler-syntaksia, jota kurssilla 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 [FIXME: kesäope ei vielä ole Googlettanut näiden kahden syntaksin nimeä, jotka kyllä helposti löytyisivät]. 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!**

Ja jotta Käyttöjärjestelmät -kesäkurssin 2007 opiskelijoiden elämä tehtäisiin hilpeämmäksi, huomataan, että varsinaisessa luentomonisteessa aina sivulta 9 alkaen kaikki assembler-esimerkit ovat juuri sillä toisella syntaksilla ja operandijärjestyksellä.

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               mov    %rsp,%rbp

^             ^                      ^
|             |                      |__ assembler-kielinen ilmaus
|             |
|             |__ 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.
|
|__ käskyn suhteellinen muistiosoite ohjelma-alueen alusta luettuna
```

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.

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 AMD:n Opteron 246 -prosessorissa (tai muussa x86-64 arkkitehtuurin mukaisessa prosessorissa):

```
Toiminnanohjausrekisterit:

   RIP     - Instruction pointer, "PC"

   RFLAGS  - Flags, "PSW"


Yleisrekistereitä datalle ja osoitteille

   RAX     - Yleisrekisteri; "akkumulaattori"

   RBX     - Yleisrekisteri; "epäsuora osoite"

   RCX     - Yleisrekisteri; "laskuri"

   RDX     - Yleisrekisteri

   RSI     - Yleisrekisteri

   RDI     - Yleisrekisteri

   RBP     - Yleisin käyttö nykyisen pinokehyksen osoitin

   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

Jatkossa keskitytään lähinnä näihin rekistereihin, joita sovellusohjelmien ohjelmoija voi nähdä ja käyttää ohjelmoimalla. Käsittelemättä jätetään liukulukulaskentaan ja multimediakäyttöön tarkoitetut rekisterit (`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. Mutta älä koskaan sano älä koskaan... voihan

sitä päätyä töihin vaikka firmaan, joka nimenomaan toteuttaa noita virtuaalikoneita, jolloin kaikkein alin taso ilman muuta tehdään konekielellä.

Käyttöjärjestelmäkoodi pääsee käsiksi noin 50 muuhun rekisteriin sekä näihin liittyvään käskykannan osaan, joilla muistinhallintaa, laitteistoa ja ohjelmien suojausta hallitaan.

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.

Käskykanta

[Tämä osuus voisi sisältää muutamia poimintoja käskykannasta aivan kuten luentomonisteen sivu 8. Olennaisia eroja ei olisi paljonkaan, paitsi rekisterien nimet ja käskyjen käsittelemien bittien määrä sekä operandien järjestys notaatiossa.]

Yksinkertainen esimerkki

Luentomonistetta mukaillen katsotaan ensin päällisin puolin muutamaa C-kielistä ohjelmaa ja niiden konekielikäännöstä. Ensimmäinen 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`. Prosessori näkee sen suoritusmielessä noin, eikä disassembler lähde arvailemaan, tarkoittiko alkuperäinen ohjelmoija positiivista kokonaislukua `0xffffffffffffffdc` (aika iso) vai negatiivista kokonaislukua -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
```

Koodi, tieto ja suorituspino; virtuaalimuisti

(Tämä kohta on tynkä. Se toteaa luentomonisteen sivun 7 lyhyen asian vieläkin lyhyemmin ja perustelemattomammin. Tarkoitus kirjoittaa tähän jotain järkevää ennen lopullista materiaalia.)

Ohjelmassa on 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.

Todetaan ykskantaan, että mainitut koodi ja data ladataan usein eri paikkoihin tietokoneen muistissa, ja paikallisille muuttujille varataan vielä ihan oma alue, jonka nimi on **suorituspino** (*stack*). Useimmissa prosessoriarkkitehtuureissa on erilliset rekisterit, joiden tarkoitus on pitää yllä suhteellista osoiteindeksiä kuhunkin näistä kolmesta muistialueesta. Luentomoniste antaa esimerkin 8086-toteutuksesta, tämä lehdykkä tulee antamaan esimerkkejä AMD64-toteutuksesta ja muillakin prosessoriarkkitehtuureilla on vastaavat käytännöt. Tämä on perusteltua (kunhan kirjoittaisin perustelun tähän) ja selkeätä ohjelmoijan kannalta. `PC` osoittaa koodialueelle, `SP` (Stack pointer) eli pinon huipun osoitin pinoalueelle ja yleiset datan indeksointirekisterit data-alueelle. Vältän käyttämästä alueista sanaa "segmentti", koska segmentoiva muistinhallinta alkaa olla historiaa. (Historiakin on kyllä tarpeellista tietoa... mutta itselläni sitä ei aivan tarpeeksi ole tästäkään asiasta, että osaisin kylmiltään kirjoittaa.)

Pinoalue on usein 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 kohtaan.

Huomaa, että prosessorin kannalta dataa ei ole missään "nimetyissä muuttujissa" kuten lähdekoodin kannalta, vaan kaikki käsiteltävissä oleva data on rekistereissä, tai se pitää ensin noutaa rekisteriin muistiosoitteen perusteella koodi-, data- tai pinoalueelta. Tulokset pitää erikseen viedä osoitteen perusteella vastaavasti. Kääntäjäohjelman tehtävänä on muodostaa numeerinen muoto osoitteille, joissa lähdekoodin kuvaamaa dataa säilytetään.

Moderneissa tietokoneissa konekielikäskyjen käsittelemät muistiosoitteet ovat ns. **virtuaaliosoitteita**: jokainen ohjelma näkee oman koodinsa, datansa ja pinonsa siten, että niille varattu ensimmäinen tavu on osoitteessa 0, seuraava varattu tavu on osoitteessa 1 ja niin edelleen aina viimeisen varatun tavun osoitteeseen saakka. Ohjelman näkemä ja konekielikäskyissä käytetty muistiosoite (datalle, koodille ja pinolle) on siis indeksi nollasta siihen maksimiin, mitä ohjelma tarvitsee. Tämän 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, kiinteän, osoiteväylään koodattavan muistiosoitteen kautta. Prosessoreissa on siis ominaisuus, joka muuntaa ohjelman virtuaaliset osoitteet todellisiksi osoitteiksi (*real address*). Tähän prosessori tarvitsee käyttöjärjestelmän apua, ja yksi nykyaikaisen käyttöjärjestelmän tärkeä tehtävä onkin **virtuaalimuistin hallinta**, jonka toteutusperiaatetta myöhemmällä luennolla käsitellään. Virtuaalimuistiin liittyy edellä mainitun kätevän ja selkeän **lineaarisen, nollasta alkavan osoiteavaruuden** (*linear flat addressing*) lisäksi muutakin mukavaa; katsotaan kaikkea sitä tosiaan sitten kun puhutaan muistinhallinasta. Jatketaan nyt vielä ohjelman suorittamisesta konekielellä. Huomaa kuitenkin, että lineaarinen nollasta alkava osoiteavaruus tarkoittaa sitä, että kun ohjelmaa suoritetaan, sen ei tarvitse yhtään välittää siitä, onko saman tietokoneen muistissa jossain kohtaa muitakin ohjelmia. Vielä ei ole nähty mitään, mikä sovelluksen kannalta antaisi vihiä moniajosta (tai mahdollisuuksia siihen). Tulossa on, mutta pysytään yhdessä ohjelmassa vielä.

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 `PC`:n arvosta toiseen
- aliohjelma-aktivaatioita.

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** (*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 versioille aliohjelman käsitteestä pitäisi löytää yhteys. Olion instanssimetodin kutsu voidaan ajatella siten, että ikään kuin olisi olioluokan jäsenten sisäistä dataa varten rakennettu aliohjelma, jolle annetaan tiedoksi (yhtenä parametrina) viite nimenomaiseen olioinstanssiin, jolle sen tulee operoida. Silloin luettaisiin imperatiivinen aliohjelmakutsu sillä tavoin että "data==olion sisäinen data" ja "toteutettava operaatio == metodiviestin mukainen operaatio". Jotain tällaista (muistaakseni) näet mm. Javan tavukoodin disassemblyssä. 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.

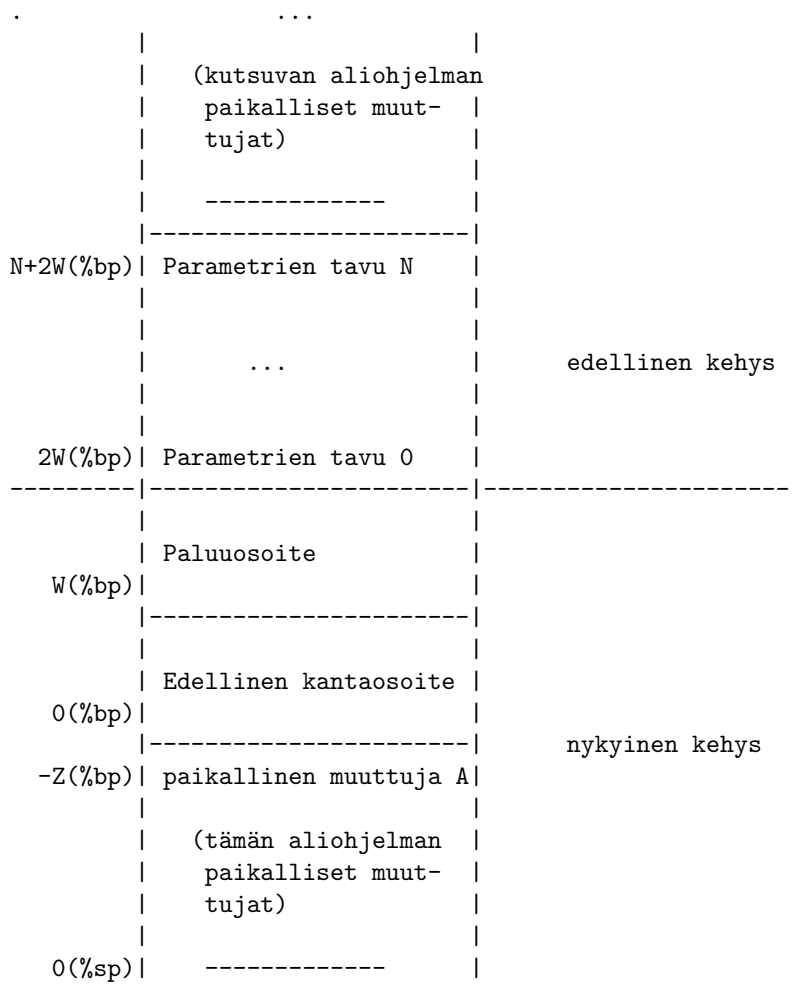
Aliohjelma-aktivaatio (eli kutsu) prosessorin toimenpiteenä

Nyt haetaan toteutuksesta ja arkkitehtuurista riippumattomia linjoja. Ymmärretään siis, että jos kerran jokainen ohjelma on aliohjelma, 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

Aliohjelmat (eli ohjelmat) suoritetaan normaalisti käyttämällä kaikkeen ylläolevaan suorituspinoa (lineaarinen nollasta alkava muistialue, joka 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 **aktivaatiorakenne** (*activation record*). Rakenteen käyttöön tarvitaan pinoalue ja kaksi rekisteriä, jotka osoittavat sinne. Toinen on pinon huipun osoitin (`SP`), ja toinen pinokehyksen/aktivaatiorakenteen kantaosoitin (joskus `BP`, *base pointer*).

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



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:

- parametrien osoitteet saadaan lisäämällä kantaosoitteeseen BP sopivat arvot; yleensä prosessorikäskyt mahdollistavat tällaisen osoitusmuodon eli "rekisteri+lisäindeksi". Parametrien arvot saadaan tarvittaessa rekistereihin siirtokäskyillä, joissa osoite tällä tavoin.
- Paluuosoite on tallessa tietyssä kohtaa pinokehystä;
- Edellisen aliohjelma-aktivaation käyttämä kantaosoitin BP on tallessa tietyssä kohtaa pinokehystä
- Paikallisia muuttujia voidaan varailla ja vapauttaa tarpeen mukaan pinosta ja SP voi rauhassa elää PUSH ja POP -käskyjen mukaisesti.

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 juuri ennen aliohjelmakutsun suorittamista.
- Yleensä prosessori toimii siten, että `CALL` -käsky tai vastaava, joka vie aliohjelmaan, toteuttaa seuraavan käskyn osoitteen tallentamisen `PC`:n sijasta pinon huipulle. `PC`:hen puolestaan sijoittuu aliohjelman ensimmäisen käskyn osoite.
- Seuraavassa prosessorin *fetch* -toimenpiteessä tapahtuu suorituksen siirtyminen aliohjelmaan
- Aliohjelman ensimmäisen käskyn pitäisi ensinnäkin painaa nykyinen `BP` eli juuri äsken odottelemaan jääneen aktivaation kantaosoitin pinoon.
- Sen jälkeen pitäisi ottaa `BP` tämän uuden, juuri alkaneen aktivaation käyttöön. Siihen kun siirtää nykyisen `SP`:n, eli pinon huipun osoitteen, niin se menee juuri niin kuin pitikin.
- Ja siten `SP` vapautuu normaaliin pinokäyttöön.

Mitenkäs moniajo sitten?

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 edellä kuvatulla tavalla ei oikein hyvin mahdollista kontrollin vaihtoa kahden ohjelman välillä. Periaatteessa kai pitäisi edellyttää 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 sitten, kun kaikkien muiden ajossa olevien ohjelmien maineja 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...)

Näin se moniajo tehtiin ehkä joskus, mutta sitten ihminen keksi, että ei tämä käy, ja insinööri keksi, että mitä sitten tehdään. Tai jotain... alkaa väsyttää. [FIXME: korjaa lopulliseen jotain järkevää tekstiä tähän ehkä].

Eritoten moniajo-ominaisuuksia vaatii prosessoritehon hyödyntäminen, jos ajetaan I/O:ta käyttäviä ohjelmia. Käskyn suorittamiseen tarvittavan tavun noutaminen kovalevyltä saattaisi kestää tuhansia kellojaksoja, joiden aikana voisi hyvin suorittaa paljon kaikkea muuta, vaikka juuri se kirjainta odottava ohjelma ei voisi edetä ennen kuin kirjain saapuu. Päätteeltä tulevaa merkkiä saattaisi joutua odottamaan 12 tuntia, jos päätteen ääressä istuva nörtti on nukahtanut tai muuten vain ei paina mitään nappia. 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.

Tavoitetilanteita:

- 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.
- 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.
- samalla kun pakkaan DVD:ltä kovalevylle dekoodattua informaatiota DivX:ksi, olisi hyvä pystyä kuuntelemaan MP3-musiikkia ilman, että ääni ratisee tai pätkii

Keskeytykset ja lopullisempi visio fetch-execute -syklistä

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. Prosessori noutaa dataa `PC`-rekisterin osoittamasta paikasta
   Noudettu data sijoitetaan sisäiseen `INSTR`-rekisteriin
2. Prosessori suorittaa käskyn
   Eli `INSTR`-rekisterissä oleva bittijono herättää kontrolliyksikössä toimenpiteitä, joihin syötetään myös muiden käskyssä tarvittavien rekisterien arvoja.
3. Käskyn suorituksen tuloksena rekisterien tila on muuttunut jollain tavoin
   Yksi, joka aina muuttuu, on `PC`. Miten `PC` muuttuu, riippuu suoritetusta käskystä:
   - Laskutoimitus, datan siirto tai muu peräkkäissuoritus ==> `PC` osoittaa seuraavaan konekielikäskyyn
   - Hyppykäsky ==> `PC` osoittaa käskyssä kerrottua uutta osoitetta, esim. silmukan alkua tms.
   - Ehdollinen hyppykäsky ==> `PC` osoittaa käskyssä kerrottua uutta osoitetta mikäli käskyssä kerrottu ehto toteutuu; muutoin osoittaa seuraavaan käskyyn
   - Aliohjelmakutsu ==> `PC` osoittaa käskyssä kerrottua uutta osoitetta (jonka tulee olla kutsuttavan aliohjelman ensimmäinen käsky; aliohjelmakutsussa prosessori tekee paljon muutakin, mitä käsitellään kohta tarkemmin)
   - Paluu aliohjelmasta ==> `PC` osoittaa taas siihen ohjelmaan, joka suoritti kutsun, erityisesti kyseessä on aliohjelmakutsua välittömästi seuraava käsky. (kohtapuoleen nähdään, miten prosessori noutaa paluuosoitteen pinomuistista)
4. Jos keskeytysten käsittely on kielletty (sellainen tilabitti “FLAGS“issä on asetettu), prosessori jatkaa sykliä aina kohdasta 1. Muutoin se tekee vielä kohdan 5.
   Inline literal start-string without end-string.
5. Prosessori tarkastaa keskeytyspyyntö -nastan jännitteen. Jos siellä on jännite, prosessori suorittaa keskeytyksen:
   Unexpected indentation.
   - [Nämä kohdat täydennettään sitten, kun on aikaa ja pienempi
     Bullet list ends without a blank line; unexpected unindent.
     väsymys ja jumi päällä.]
   - Tässä kuvattaisiin tyypillinen FLIH. Mutta sama asia on kyllä esitetty luentomonisteessa ihan hyvin.

Tarkemmin laitteiden ja laiteajurien toimintaa käsitellään loppupuolella kurssia, otsikolla I/O.