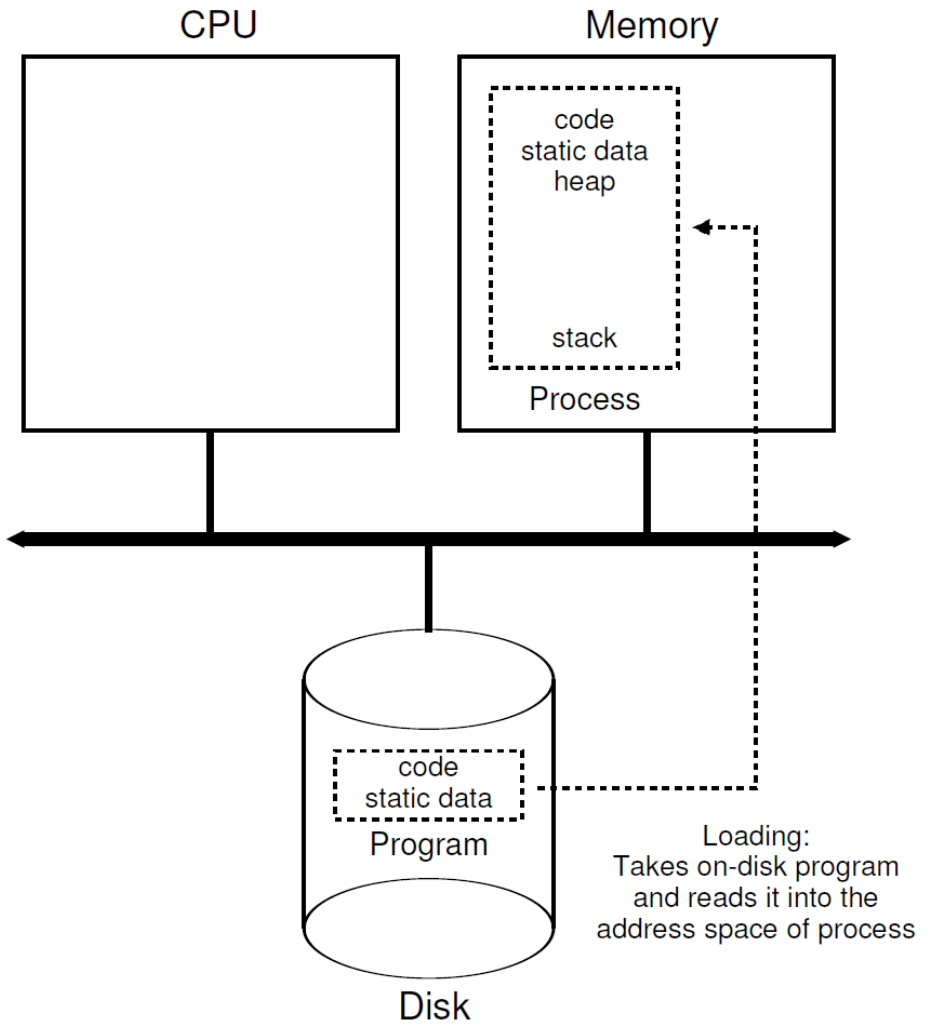
LUT
University

CT30A3370 - KÄYTTÖJÄRJESTELMÄT JA SYSTEEMIOHJELMOINTI 6 OP

Jussi Kasurinen (etu.suku@lut.fi)

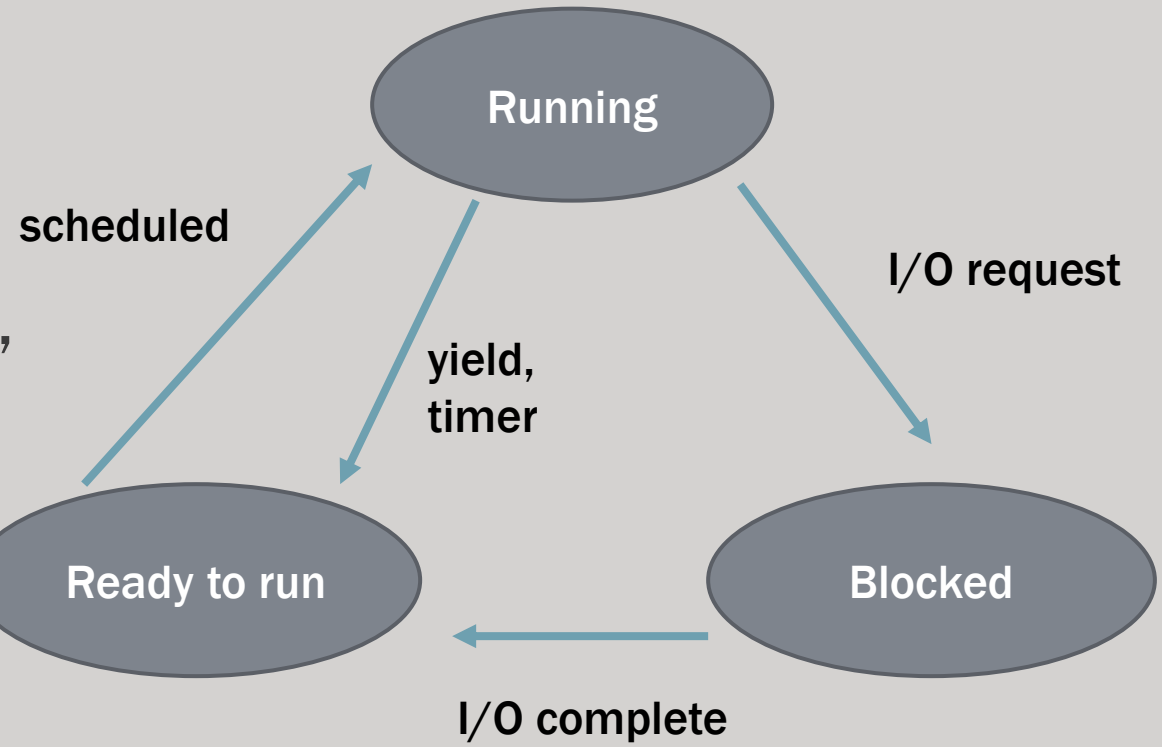
Osa kalvoista Timo Hynnisen 2016 materiaaleista

OHJELMA VS. PROSESSI



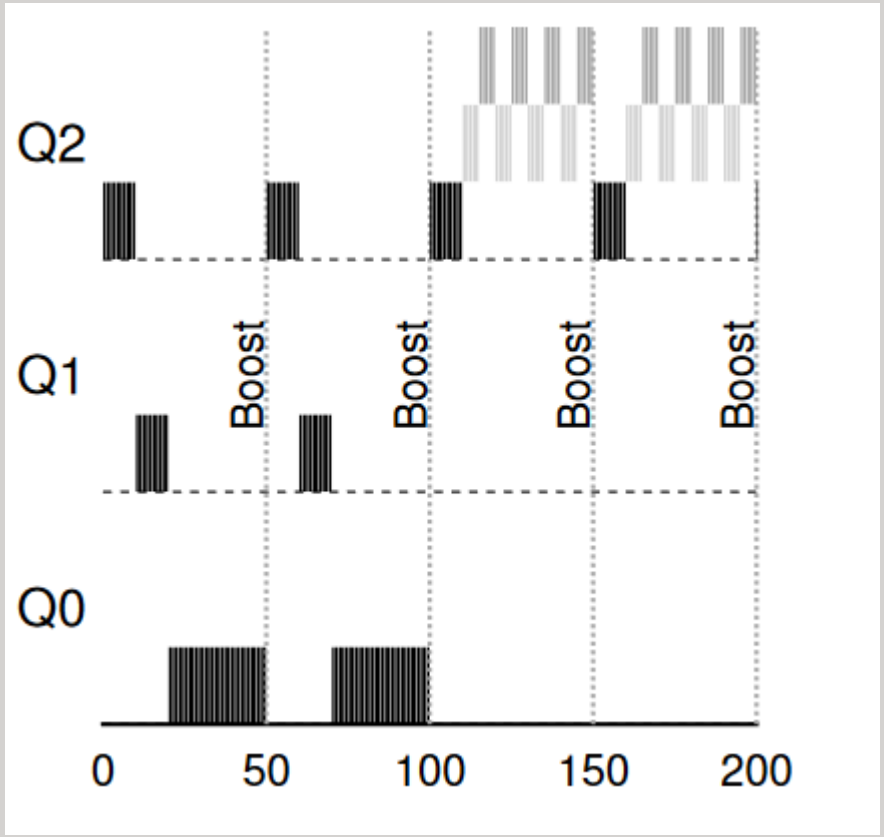
PROSESSI VOI OLLA KOLMESSA TILASSA:

- Running
  - Suorituksessa suorittimella nyt
- Blocked
  - Odottaa jotain muuta resurssia, vaikkapa I/O-laitetta
- Ready to run
  - Vuorontajan listalla, odottamassa suoritusta



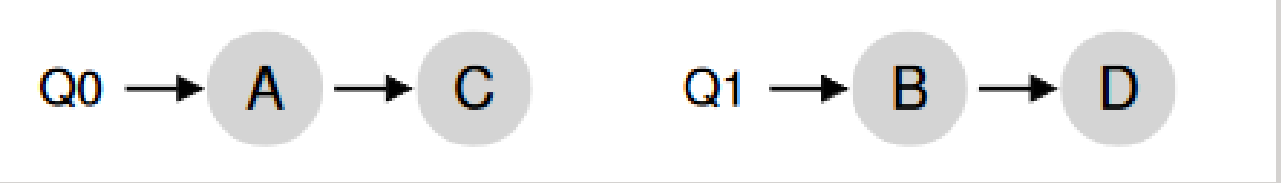
MULTI-LEVEL FEEDBACK QUEUE (SE MITÄ BSD LINUX JA WINDOWS KÄYTTÄÄ)

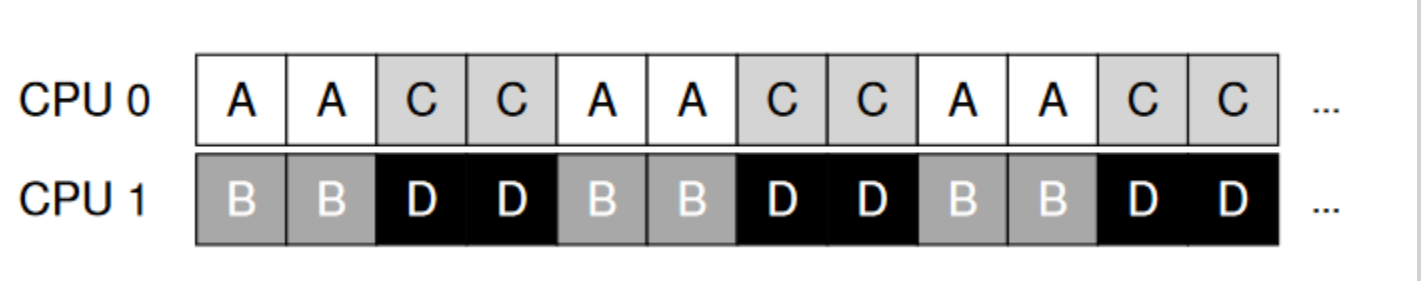
- Interaktiivisuus pysyy korkeana.
- Laskentaintensiiviset prosessit vajoaa alaspäin “taustalle” ja niitä suoritetaan aina kun ei ole kiireellisempää tiellä.
  - Nostoaika varmistaa että kaikki kuitenkin saa välillä prosessoriaikaa.



VUORONNUS MONELLE PROSESSORILLE?

- Jos suorittimia on useita, päättää vuorontaja siitä, mihin suoritinjonoon mikäkin prosessi menee.





ASSEMBLER (SYMBOLINEN KONEKIELI)

CT30A3370 - Käyttöjärjestelmät ja systeemiohjelmointi

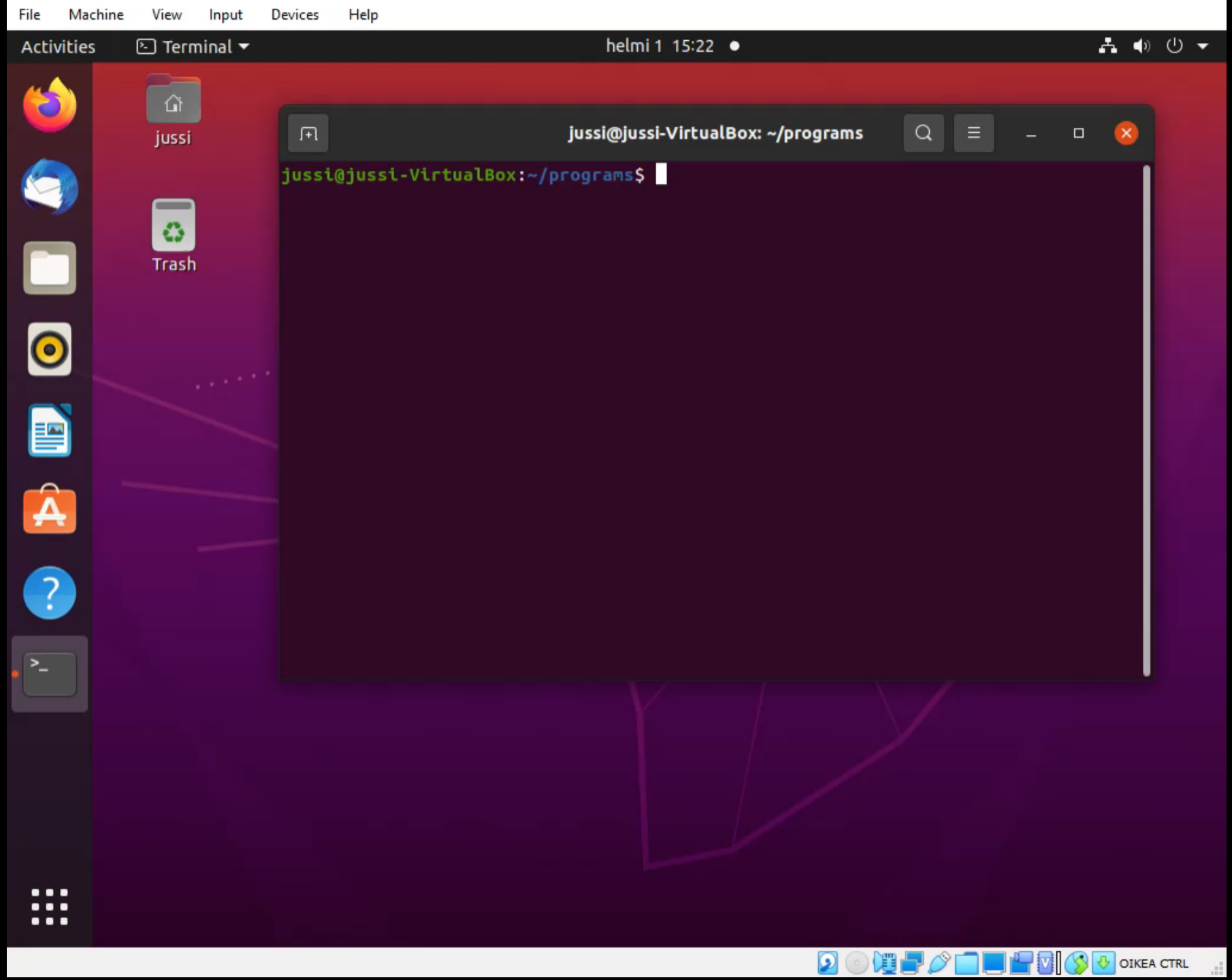
ASSEMBLER JA VIRTUAALIKONEET

- ...Eli tässä vaiheessa meillä on
  - Joku idea siitä miten tietokone rakentuu
  - Joku idea siitä miten tietokone suorittaa asioita.
  - Joku idea siitä miten kone valitsee sen operaation mitä seuraavaksi tehdään.
- Puhutaan hetki siitä, miten ylemmän tason koodi oikeasti muuntuu prosessissa suoritettavaksi käskysarjaksi.

PROSESSEISTA ETEENPÄIN

- …Eli tässä vaiheessa meillä on viereisen kaltaista Assembly-koodia.
  - Mitä tälle sitten tehdään?

```
                .file   "cpu.c"
                .text
                .section        .rodata
.LC0:
                .string "common.h"
.LC1:
                .string "rc == 0"
                .text
                .globl  GetTime
                .type   GetTime, @function
GetTime:
.LFB5:
                .cfi_startproc
                pushq   %rbp
                .cfi_def_cfa_offset 16
                .cfi_offset 6, -16
                movq    %rsp, %rbp
                .cfi_def_cfa_register 6
                subq    $48, %rsp
                movq    %fs:40, %rax
                movq    %rax, -8(%rbp)
                xorl    %eax, %eax
                leaq    -32(%rbp), %rax
                movl    $0, %esi
                movq    %rax, %rdi
                call    gettimeofday@PLT
                movl    %eax, -36(%rbp)
                cmpl    $0, -36(%rbp)
                je      .L2
                leaq
__PRETTY_FUNCTION__.2816(%rip), %rcx
                movl    $10, %edx
                leaq    .LC0(%rip), %rsi
                leaq    .LC1(%rip), %rdi
                call    __assert_fail@PLT
.L2:
                movq    -32(%rbp), %rax
                cvtsi2sdq       %rax, %xmm1
```


File
Machine
View
Input
Devices
Help
Activities
Terminal
helmi 1 15:22
jussi
Trash
jussi@jussi-VirtualBox: ~/programs
jussi@jussi-VirtualBox:~/programs$
OIKEA CTRL

ASSEMBLER

- Miten me saadaan korkean tason ohjelmakoodista väännettyä lopulta konekieltä, mitä tietokone pystyy suorittamaan
  - Ja miten se, mitä ohjelmakoodissa on määrätty, oikeasti sitten päätyy käyttäjän ruudulle.

ASSEMBLER

- assembly-kielet eli symboliset konekielet
  - Aloitetaan miettimään ohjelmistohierarkiaa alhaalta ylöspäin - assembler on ensimmäinen askel siinä
  - Niin laiteläheistä ohjelmointia, kuin mihin ohjelmoija oikeastaan ikinä tulee törmäämään
  - Ehkä osin jopa “turhaa tietoa”? Harva tulee käyttämään?
  - Mutta! Toisaalta myös, jos mietitään tietokoneen toimintaa, ja sitä ympäristöä, missä käyttöjärjestelmä (ja osin systeemiohjelmat) joutuvat toimimaan, niin nyt aletaan olla oikealla tasolla.

ASSEMBLER

- Puhutaan tässä välissä ihan vähän siitä, miten nämä sähköiset piirisirut nivoutuu ohjelmoinnin kanssa yhteen.
- Kaikille lie selvää, että väylät on käytännössä vain sähköjohtoja, joissa liikkuu virtaa?

ASSEMBLY

- Sähkövarauksella on helpohko esittää lukuja => saadaan äkkiä rakennettua ‘laskukone’
  - Tämän takia tietojenkäsittelyn perusteista puolet on binääriluvuilla laskemista
- Käskymuisti on piirisiru, joka työntää binäärisiä toimintaohjeita suorittimelle yksi kerrallaan.
  - TAI tietysti jos ihan tarkkoja ollaan, niin suoritin hakee käskyt muistista yksi kerrallaan.
  - .. ja tässä kohtaa ei ole syytä mennä tarkemmin siihen, miten piirisarja oikeasti realisoituu tai miten syöttölaitteet ja sensorit toimii.
  - Riittää, että ymmärretään (tai ollaan aikaisemmin opittu), kuinka alkeissiruista pystytään rakentamaan erilaisia muistipiirejä. (Tai oikeastaan, ei sitäkään tarvitse ymmärtää. Voidaan ottaa tämä itsestäänselvyytenä.)
  - Muuttujien kartoittaminen muistipaikkoihin tapahtuu *symbolitaulun* avulla

ASSEMBLY

- Ok, käskymuistissa (ROMmissa) on konekielen komentoja binäärimuodossa.
  - Koska suoritin (suoritin-piirisiru) ymmärtää nämä binääriset ohjauskoodit, niin oikeita asioita tapahtuu
  - Binäärinen konekieli on nyt (taikuuden avulla) vain kokoelma sovittuna, säännönmukaisia käskyjä, jotka osaamalla suorittimen voi valjastaa tekemään tietojenkäsittelyä
    - Jälleen, meidän ei tarvitse tässä kohtaa juuri välittää siitä, millainen piirisiru suoritin on. Riittää, että käsitämme sen kasana erilaisia portteja, kiikkuja, kytkin- ja ohjauslaitteita.

ASSEMBLY

- Binääriluvuilla ohjelmointi olisi kai aika tuskallista.
  - Esim. tyypillinen ohjauskomento
  - Toki helpohko selvittää ohjekirjan kanssa…

- 1010 0001 0010 1011 → ADD R1, R2, R3

- Konekieli on kuitenkin määrämuotoista
  - Ehkä binäärimuotoista konekieltä kannattaisi tulkata kielestä, jossa samat käskyt on sidottu ihmisläheisempään esitysmuotoon

ASSEMBLY

- **Symbolinen konekieli** on siis konekielen havainnollisempi ja ihmisläheisempi esitysmuoto, joka määrittelee konekielen käskyille kirjoitetun kielen kaltaisen ulkoasun.
  - Assembly-kielessä on siten lähes sama rakenne ja komennot kuin varsinaisessa konekielessä, mutta tekstimuotoisuus helpottaa ohjelman kirjoittamista ja ymmärtämistä.
  - Lisäksi eri muistiosoitteisiin viitataan assemblyssa usein nimin ja itse muistiosoite voidaan antaa assemblerin määriteltäväksi.

ASSEMBLER

- Huomioitavaa vielä: Konekieli ja siis myös assembly ovat sidonnaisia tiettyyn suoritimeen.
- Ainoa oikea haaste symbolisen konekielen kääntämisessä: Assembly-kielet yleensä sallivat muistiosoitteisiin symbolisen viittauksen
  - “Muuttujien” käytön (lainausmerkeissä)
  - varsinaisia muuttujia sanan varsinaisessa merkityksessä ei Assemblyssä ole, muuttujat on kirjaimellisesti vain viittauksia johonkin muistipaikkaan.
  - Muuttujien kartoittaminen muistipaikkoihin tapahtuu *symbolitaulun* avulla

SYMBOLINEN KONEKIELI

- Symbolinen konekieli muutetaan konekieleksi assemblerilla (eli assembler-nimisellä kääntäjällä).
  - Tästä siis tulee ero nimissä, assembler-kääntäjä ja assembly-kieli
  - Assembler on ensimmäinen askel tietokoneen ohjelmistohierarkiassa!
    - Voidaan sanoa, että myös yksinkertaisin, matalatasoisin jne.
    - samalla ensimmäinen kerros laitteiston yläpuolella
    - Assemblerin yläpuolella on paljon softaa eri tasoilla ohjelmistoarkkitehtuurin hierarkiassa, mutta assembleri on se ensimmäinen kerros, minkä päälle kaikki rakentuu.

SYMBOLINEN KONEKIELI

- Assembler on käytännössä vain kone, joka sääntöjä noudattaen muuttaa symboliset (eli ihmiskieliset) käskyt konekielisiksi (eli binäärisiksi)

- Assembler lukee tekstitiedoston, johon symboliset konekäskyt on kirjoitettu
  - Ja tuottaa siitä binäärisen vastakappaleen.
  - … minkä voi sitten lyödä yleispätevään tietokoneeseen kiinni, ja suorittaa kirjoitetun ohjelman suoraan
    - Avainsana tässä, suoraan. Assemblerilla käännetty ohjelma voidaan suorittaa suoraan tietokoneessa, eikä se tarvitse mitään ohjelmistoalustaa alleen.

SYMBOLINEN KONEKIELI

- No Miksi symbolinen konekieli, assembly, on ylipäätään olemassa?
  - Ohjelmoidaan konekielellä, kuitenkin siten, että käskyt ovat ihmiselle ymmärrettäviä
  - Assemblyä kirjoittaessa ymmärretään täsmällisesti, mitä laitteistotasolla tapahtuu, kuitenkin ilman että tarvitsisi ohjelmoida ykkösillä ja nollilla suoraan
  - Assembly-kielet on edelleen hyvin, hyvin matalan tason kieliä ja kuten todettua, täysin verrattavissa (binääriseen) konekieleen.

MISSÄ ASSEMBLERIA AJETAAN?

- Millainen ohjelma Assembler on?
  - Okei, oletetaan että ollaan juuri rakennettu joku tietokone
  - Tässä vaiheessa ainoa tapa hallita sitä olisi ohjelmoida sitä suoraan binäärisellä konekielellä... mikä saattaa pidemmän päälle käydä vähän ärsyttäväksi.
  - Joten ratkaisu: Kirjoitetaan toisella tietokoneella, omalla läppärillä, Assembler-ohjelma, joka osaa kääntää Assemblyä binäärikoodiksi.
- Paras tapa ajatella asiaa on, että me emme ole rakentamassa maailmankaikkeuden ensimmäistä tietokonetta, vaan ehkä toista tai kolmatta
  - Jos meillä on jo joku tietokone, jota voi ohjelmoida korkean tason ohjelmointikielellä, tällaisen Assembly -> konekieli -kääntäjä on suhteellisen triviaali tehtävä toteutettavaksi.
  - Konekielisen ohjelman tässä ajatusmallissa tuottaa siis eri kone, kuin se, millä ajattelimme tuotetun ohjelman ajaa.

ASSEMBLER-OHJELMAN TOIMINTA

- Assembler on oikeasti tosi yksinkertainen ohjelma
- Se toistaa vaan tällaista perussilmukkaa kunnes on saanut lähdekooditiedoston luettua
  - Lue Assembly-kielinen komento tiedostosta
  - Erotetaan käskyn osat toisistaan
  - Etsitään osakäskyjen binäärikoodit
  - Yhdistetään binääriset osakäskyt
  - Kirjoitetaan binäärinen ohjauskoodi

ASSEMBLER-OHJELMAN TOIMINTA

- Miten seuraavan komennon lukeminen tiedostosta tapahtuu?
  - Luetaan tiedostosta rivi
    - tämä melkein riittää, ainoa asia mikä pitää ottaa huomioon, on tyhjien merkkien huomiotta jättäminen
    - Koska on tärkeä lukea nimenomaan se seuraava komento, eikä esimerkiksi kommenttia tai rivinvaihtoja
- Sovitaan, että meillä on esimerkiksi vaikka tällainen LOAD R1, 18 -komento
  - Tämä on muuten täysin mielikuvituksellista kieltä, sovitaan vaikka, että komento lataa luvun 18 ensimmäiseen rekisteriin
- Tässä vaiheessa ei ole tärkeää tietää, mitä komento tekee, riittää, että saadaan tuo komento talteen esimerkiksi johonkin taulukkoon

ASSEMBLER-OHJELMAN TOIMINTA

- Seuraava askel on ottaa tämä luettu merkkijono, ja katkaista se oikeista kohtaa, jotta erotetaan mikä osa komennosta on käskyä ja mitkä käskyn parametrejä
  - Eli LOAD, R1 ja 18 on kaikki erillisiä asioita, kaikki yhtä merkityksellisiä

ASSEMBLER-OHJELMAN TOIMINTA

- Kun yksittäiset käsky osat on selvillä, pitää muuntaa jokainen niistä vastaavaksi binäärikoodiksi
  - Tämän muunnoksen määrää konekielen speksi
  - Eli käytännössä meillä olisi tällainen taulu, jossa speksataan, mikä binäärikoodi vastaa mitäkin komentoa
  - Luvut, rekisterit ym. vakiot: niiden esitysmuoto riippuu ihan täysin konekielen speksistä. Voi olla esim. että tuo luku 18 esitetään täsmällisesti vain binäärimuodossa - tai sitten ei.
    - Mutta tämä kaikki on aina konekielestä riippuvaista
  - Tämä on ainoa kohta koko käännöksen aikana, jolloin meidän pitää tietää täsmällisesti, miten symboliset käskyt vastaavat binäärisiä. Muulloin siitä ei tarvitse välittää ollenkaan.

ASSEMBLER-OHJELMAN TOIMINTA

- Kun yksittäiset käsky osat on selvillä, pitää muuntaa jokainen niistä vastaavaksi binäärikoodiksi
  - Tämän muunnoksen määrää konekielen speksi
  - Eli käytännössä meillä olisi tällainen taulu, jossa speksataan, mikä binäärikoodi vastaa mitäkin komentoa
  - Luvut, rekisterit ym. vakiot: niiden esitysmuoto riippuu ihan täysin konekielen speksistä. Voi olla esim. että tuo luku 18 esitetään täsmällisesti vain binäärimuodossa - tai sitten ei.
    - Mutta tämä kaikki on aina konekielestä riippuvaista
  - Tämä on ainoa kohta koko käännöksen aikana, jolloin meidän pitää tietää täsmällisesti, miten symboliset käskyt vastaavat binäärisiä. Muulloin siitä ei tarvitse välittää ollenkaan.

ASSEMBLER-OHJELMAN TOIMINTA

- Sitten lopuksi vain liitetään nämä käskyn eri osat yhteen
  - Mahdollisesti kaikki binääriset komennot yhdessä eivät muodosta tarpeeksi pitkää riviä - tällöin tarvitaan ehkä ylimääräistä ‘paddingiä’ eli lisää nollia binääriluvun esitykseen johonkin kohtaan.
- Ok, assembly-koodissa on kyllä muutama monimutkaisuus, esimerkiksi ohjelmalohkojen symboleiden (eli labeleiden) käyttö, sekä muuttujat, mutta ne ei nyt tässä ole niin tärkeitä

HUOMIOITA

- Miten asiat oikeassa maailmassa toimii?
- Yksinkertainen konekieli ei vaadi monimutakaista assembleria
  - Toki voidaan hyppiä myös järjen rajamailla…
  - Ja totta kai vastaavasti, laajempien konekielten assemblerit ovat monimutkaisempia
- Joissain assemblereissa on sisäänrakennettuna symbolinen “muuttujien” käsittely vakioiden aritmetiikkaa varten
  - Esimerkiksi array+5 => viides muistipaikka siitä, mihin ‘array’ viittaa

HUOMIOITA

- Assemblereissa on yleensä mahdollisuus kirjoittaa makroja
  - Esim. printf merkkijonon_osoite => movq merkkijonon_osoite, %eax; call printf
  - Makrot helpottavat assemblyn kirjoittamista käsin, erityisesti tällaista usein käytettyjä operaatioita. makrojen käyttökustannus on lisäksi pienehkö.

HUOMIOITA

- Standalone -assemblereja ei käytännössä oikeasti käytetä. Assembleri on yleensä jonkun muun kääntäjän mukana
  - Esim. GCC:n as..
  - Assembly-ohjelmia harvemmin kirjoitetaan käsin, sen sijaan kääntäjät kirjoittavat ne.
    - Koska kääntäjä on kone, sen ei välttämättä edes tarvitse vaivautua kirjoittamaan symbolista konekieltä - suoraan binäärisen konekielen kirjoittaminen voi olla koneelle helpompaa (lue: nopeampaa)
  - Toisaalta, monet matalat korkean tason kielet mahdollistavat inline-assemblyn kirjoittamisen
    - Ts. assemblykoodin kirjoittamisen toisen ohjelmointikielen sekaan
    - Tämä on yleistä varsinkin C-kielessä
    - Antaa C-ohjelmalle valtavat optimointimahdollisuudet, kun ohjelmoija pääsee suoraan käsiksi laitteistotason ohjelmointiin.

VIRTUAALIKONE JA PINOARITMETIIKKA

CT30A3370 - Käyttöjärjestelmät ja systeemiohjelmointi

MISSÄ OLLAAN

- Ollaan rakentamassa ohjelmistohierarkiaa ohjelmoitavaan tietokoneeseen
- Ohjelmistohierarkiassa jossain korkean tason kielen ja käyttiksen sekä assemblyn välimaastossa.

MISSÄ OLLAAN

- Motivaatio tämän aiheen tutkimiseen on siis edelleenkin siinä, miten korkean tason ohjelmointikielestä saadaan käännettyä konekielistä koodia ja minkä askeleiden avulla.
- Tämän viikon aiheet ovat taustaa
  - “Hyviä ajatuksia” jotka helpottavat elämää tietotekniikan näkökulmasta

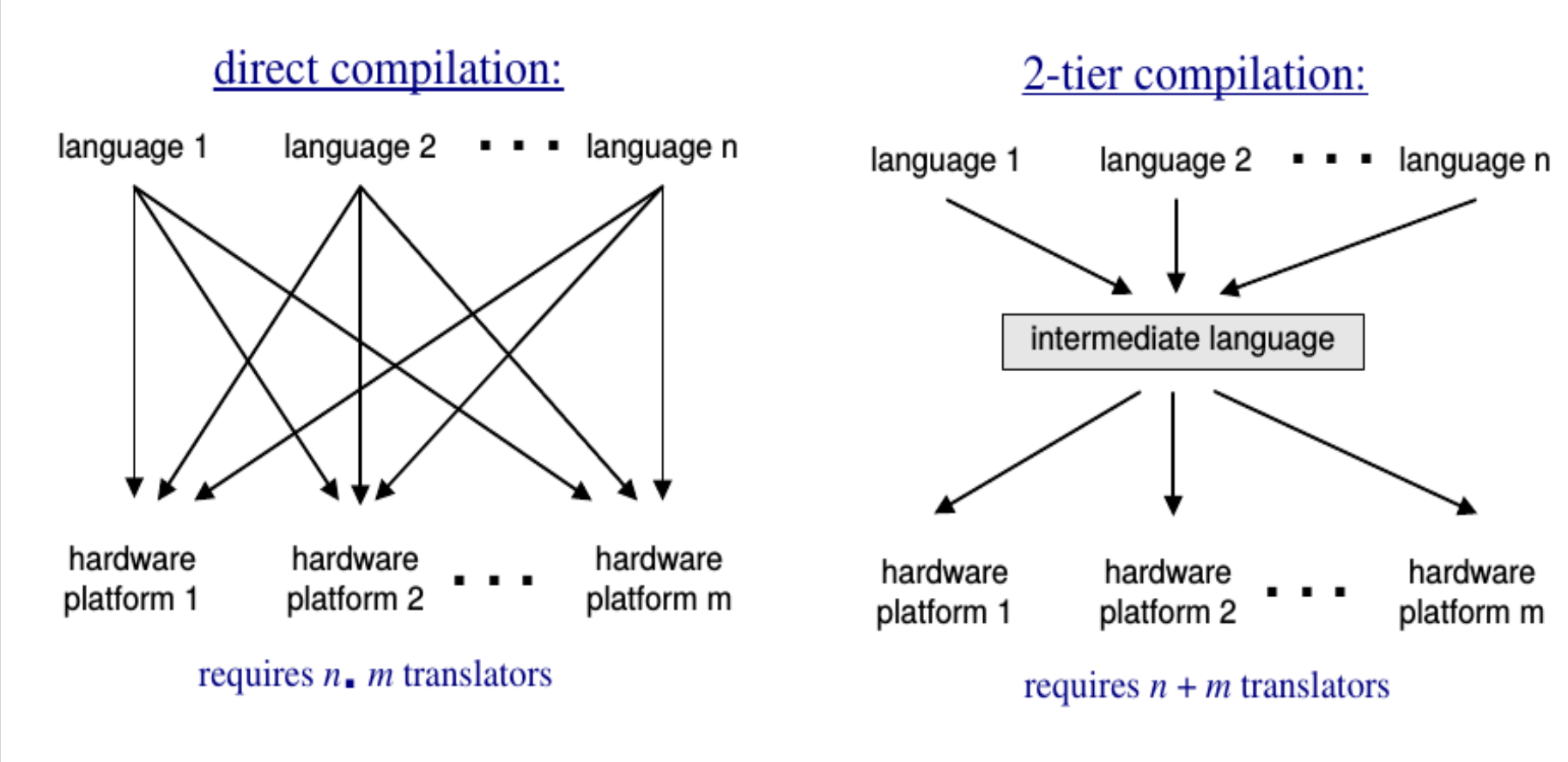
KAKSIOSAINEN KÄÄNTÄMINEN

- Tarvittavia askeleita, mitä korkean tason ohjelmointikielen suorittaminen loppupelissä laitteistolta ja ohjelmistohierarkian alaportailta vaatisi
  - Jotta homma ei paisuisi liian massiiviseksi, ratkotaan tätä ongelmaa pienemmissä osissa
    - Ensinnäkin korkean tason ohjelmointikielen kääntäminen välikielelle, mikä itse asiassa enemmän tai vähemmän jätetään tarkastelun ulkopuolelle.
    - Toisekseen välikielen tulkkaus konekieleksi

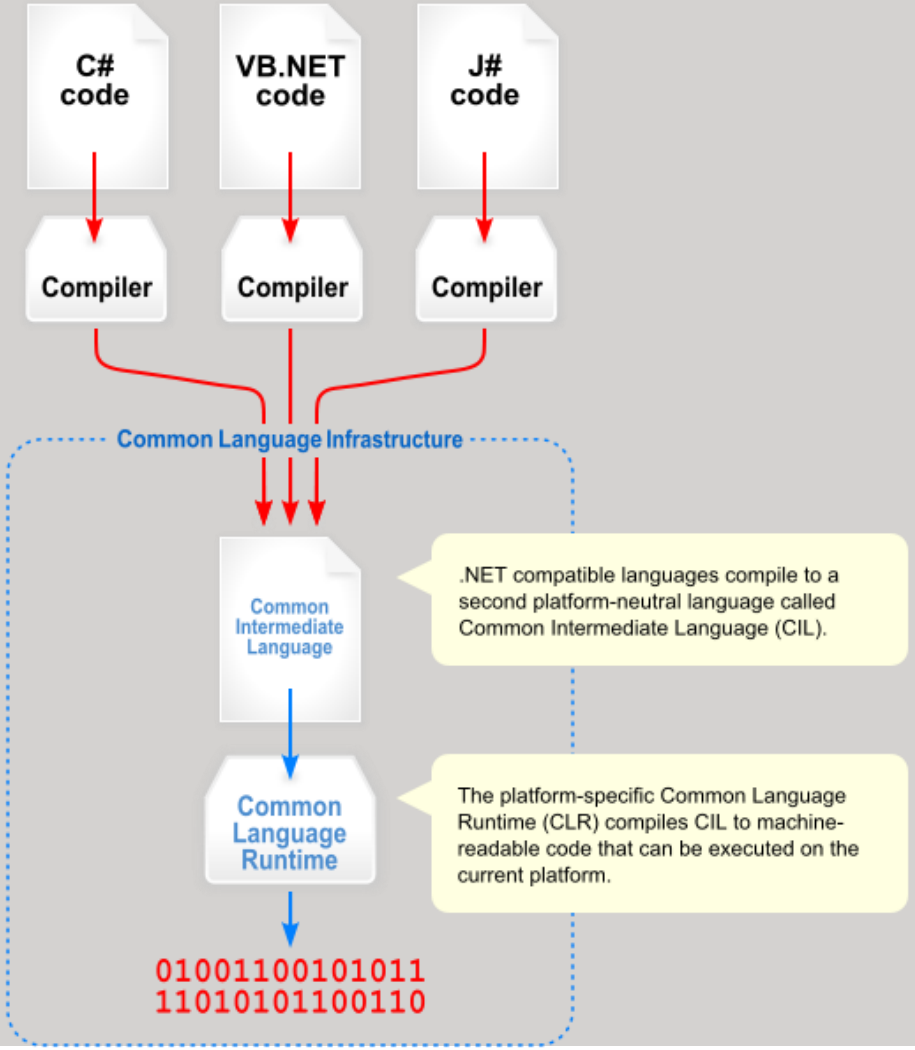
KAKSIOSAINEN KÄÄNTÄMINEN

- Tämä konsepti itsessään on nimeltään kaksitasoinen tai kaksikerroksinen kääntäminen (two-tier translation)
  - Idea suhteellisen vanha, ensimmäistä kertaa tällaista kääntämismallia käytettiin jo 70-luvulla ohjelmointikielissä
  - 90-luvulla kerroksittainen kääntäminen teki comebackin
    - Sellaiset isot ohjelmistoalustat tai ekosysteemit, kuten Java ja .NET käyttävät tämän näköistä arkkitehtuurimallia.

KAKSIOSAINEN KÄÄNTÄMINEN



KAKSIOSAINEN KÄÄNTÄMINEN



KAKSIOSAINEN KÄÄNTÄMINEN

- Kaksitasoisen kääntämisen idea on varsin yksinkertainen
  - Sen sijaan, että suoritettaisiin koodia oikealla laitteistolla, tulkataan välikielistä koodia *virtuaalikoneessa*
  - Virtuaalikone on abstrakti tietokoneen malli, mitä ei ole oikeasti (fyysisesti) rakennettu, vaan se on toteutettu pelkästään ohjelmistotasolla
  - *Virtuaalikone*-ohjelmiston voi sitten kirjoittaa kullekin fyysiselle (=oikealle) tietokonelaitteistolle sopivaksi

KAKSIOSAINEN KÄÄNTÄMINEN

- Virtuaalikoneissa on järkeä monestakin syystä
  - Ensinnäkin se on helpohko tapa mahdollistaa ohjelmien siirtäminen koneesta toiseen
    - Virtuaalikoneen toteutus eri laitteistoarkkitehtuureille ei ole erityisen vaikea tehtävä, jolloin virtuaalikoneen päällä suoritettavat ohjelmistot voidaan toteuttaa ilman, että laitteiston ominaisuuksiin tarvitsee ottaa kantaa.
  - Toisekseen virtuaalikoneen toteutukseen on erilaisia lähestymistapoja
    - Voidaan tehdä pelkästään ohjelmistotasolla
    - Voidaan rakentaa johonkin sulautettuun järjestelmään käytännössä laitteistotasolla
    - Voidaan tulkata virtuaalikoneen konekieltä suoraan laitteiston konekielelle

KAKSIOSAINEN KÄÄNTÄMINEN

- Virtuaalikone ja välikieli ideoina edustavat useita tärkeitä keksintöjä tietotekniikan ja tietojenkäsittelyn saralla
  - Ensinnäkin, tietokoneen emuloiminen tietokoneella on Alan Turingin ajatus 1930-luvulta!
    - Vuosien varrella ajatusta on hyödynnetty eri näköisiin sovelluskohteisiin
    - Vanhoja tietokoneita on emuloitu uudemmilla, jotta vanhoja ohjelmistoja pystytään edelleen käyttämään

KAKSIOSAINEN KÄÄNTÄMINEN

- Toinen tärkeä aihe, mistä puhutaan, ja mikä virtuaalikoneeseen liittyy, on sellainen tietorakenne kuin *pino* ja sen hyödyntäminen.
  - Se on yksinkertainen ja yksinkertaisuudessaan monipuolinen tietorakenne, jota käytetään monissa laitteissa ja algoritmeissa.
    - Teknisesti kykenevä ja riittävä Turing-täydellisyyden toteuttamiseen mutta ei tällä kertaa läpikäydä aihetta enempää.
  - Virtuaalikoneet ovat pohjimmiltaan pinoihin perustuvia
- Ennen kuin korkean tason ohjelmointikielellä kirjoitettu ohjelma voidaan ajaa kohdelaitteistolla, se pitää kääntää / tulkata laitteiston konekieleksi

KAKSIOSAINEN KÄÄNTÄMINEN

- Ohjelmointikielen kääntäminen on jokseenkin monimutkainen prosessi
  - Kääntäminen ei ole “tarkkaa tiedettä” vaan siihen on eri lähestymistapoja
- Yleensä tarvitaan siis oma kääntäjä kullekin ohjelmointikielen ja tietokoneen yhdistelmälle
  - Tästä tietysti seuraa aivan hillitön määrä eri kääntäjiä
  - Jos laitteita on n kpl ja ohjelmointikieliä m kpl n*m

KAKSIOSAINEN KÄÄNTÄMINEN

- Yksi virtuaalikone -> N kääntäjää + M laitealustaa
  - Ja bonuksena voidaan käyttää eri kieliä eri asioiden tekemiseen!

MITÄ TÄSTÄ LUENNOSTA PITÄÄ MUISTAA?

- Assembler ja Assembly-kuvauskieli ja mikä niiden ero on
- Virtuaalikoneet
- Kaksiosainen kääntäminen

LUT University