Tulkinta ja emulointi

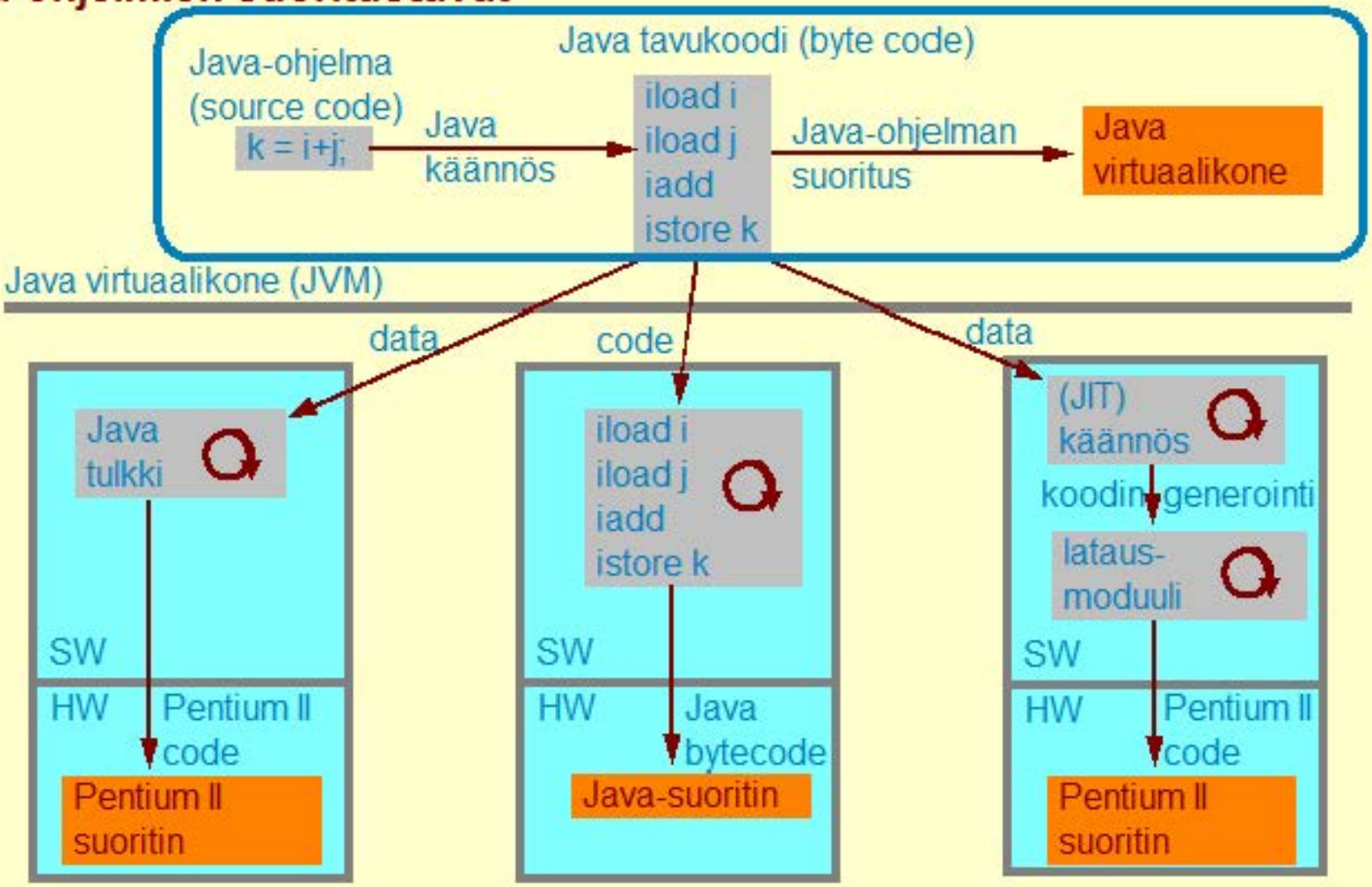
Java-ohjelman suoritus
Java tavukoodi, Java-virtuaalikone

Java-ohjelmien emulointi ja käännös
Java-suorittimet

C#
ttk-91, Crusoe

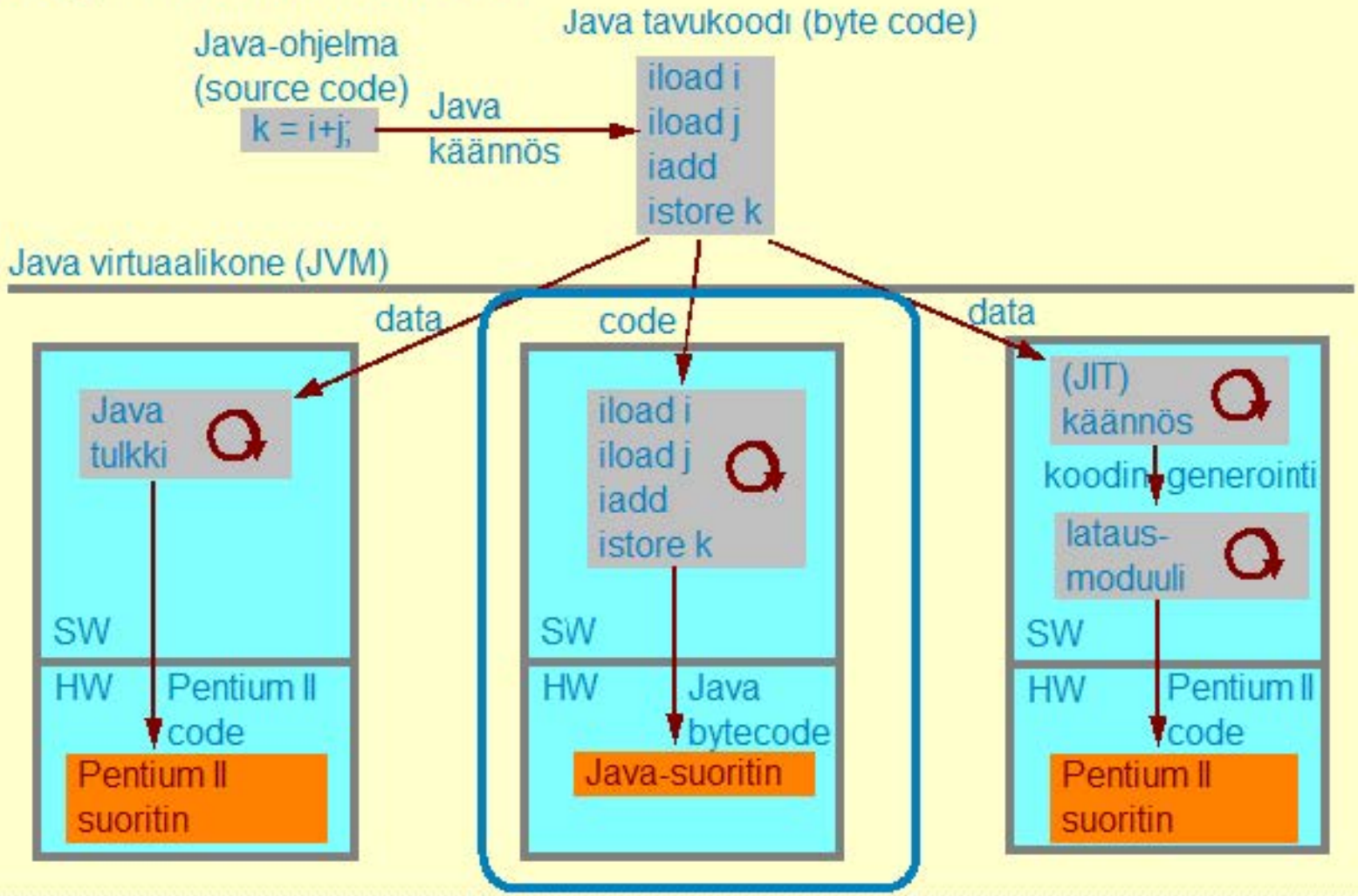
Tällä luennolla tutustumme eri tapoihin, joilla Java-ohjelmia voidaan suorittaa. Esittelemme ensin taustalla olevan Javan tavukoodin ja tavukoodia suorittavan hypoteettisen Java-virtuaalikoneen rakenteen. Näytämme sitten, kuinka Java-virtuaalikone voidaan toteuttaa sekä laitteiston että ohjelmiston avulla. Esittelemme myös lyhyesti C#'n, joka on Microsoftin kilpailija Javalle. Näytämme, kuinka ttk-91 koneen emulointi on hyvin samankaltainen suoritus Java-tavukoodin emuloinnin kanssa. Esittelemme myös Transmetan Crusoe-suorittimen, joka perustuu laitteistotuen avulla toteutettuun Intelin arkkitehtuurin emulointiin.

Java-ohjelmien suoritustavat



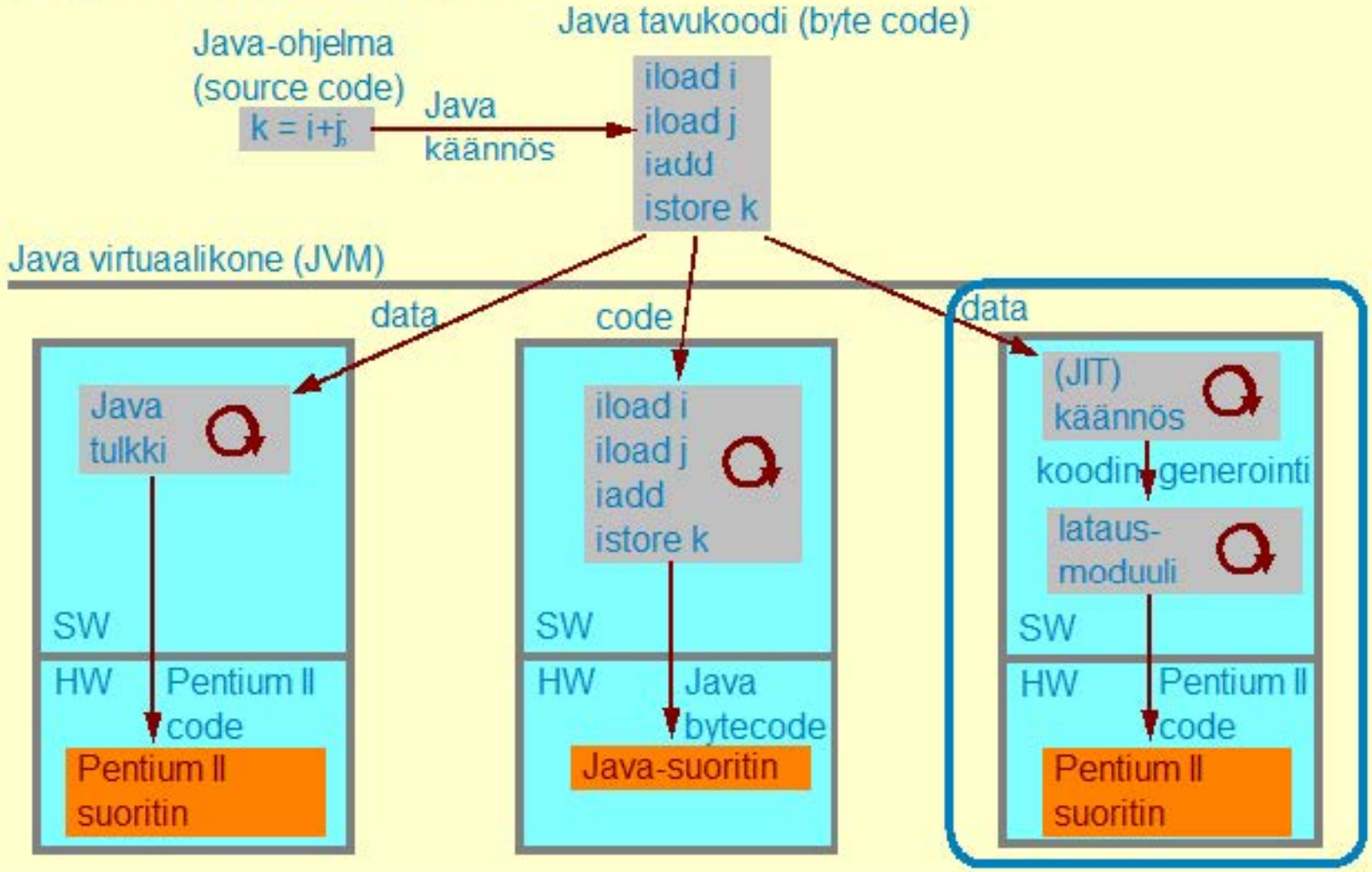
Java-ohjelmien suoritustapa poikkeaa aika lailla tavanomaisilla ohjelmointikielillä, esimerkiksi Fortranilla tai C:llä kirjoitettujen ohjelmien suoritustavasta. Java-ohjelmat käännetään aina Java-tavukoodiksi eli bytekoodiksi. Tavukoodi ei yleensä ole todellisen suorittimen konekieltä, vaan hypoteettisen Java-virtuaalikoneen konekieltä. Tavukoodiset ohjelmat suoritetaan siis aina Java-virtuaalikoneessa, jonka toteutustapoja on moninaisia. Java-ohjelmien käännetty esitysmuoto on kuitenkin varsinaisesta suorituslaitteistosta riippumaton tavukoodi. Java-kääntäjän tuottama tavukoodimoduuli ei siis ole tavanomainen linkitettävä tai ladattava objektimoduuli.

Java-ohjelmien suoritustavat



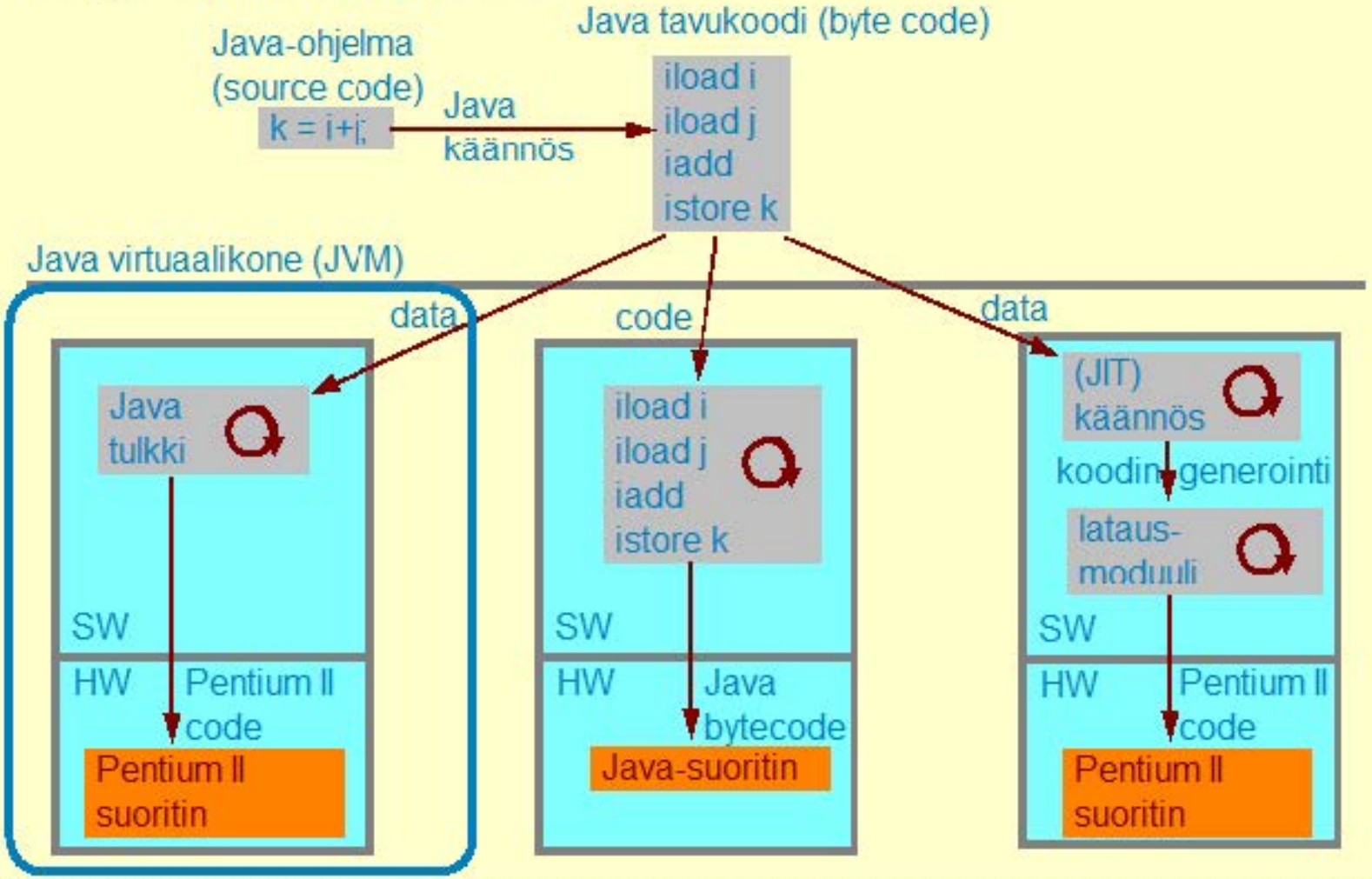
On olemassa erityisarkkitehtuureja, joiden konekieli on suoraan Javan tavukoodi. Näitä laitteistoja ei yleensä käytetä pöytäkoneina. Ne ovat yleensä jotain erikoislaitteistoja, jotka on nimenomaan suunniteltu Java-sovelluksia varten. Näille arkkitehtuureille tavukoodiset moduulit ovat suoraan suoritettavaa koodia, joten näissä tapauksissa Java-ohjelmien suoritus muistuttaa ehkä kaikkein eniten tavanomaista ohjelmien suoritustapaa. Yleensä kuitenkin tavukoodi on liian tehotonta käyttöjärjestelmän toteuttamiseen, joten Java-suorittimilla konekielinen käskykanta sisältää myös muita konekäskyjä kuin Java-tavukoodin käskyt.

Java-ohjelmien suoritustavat



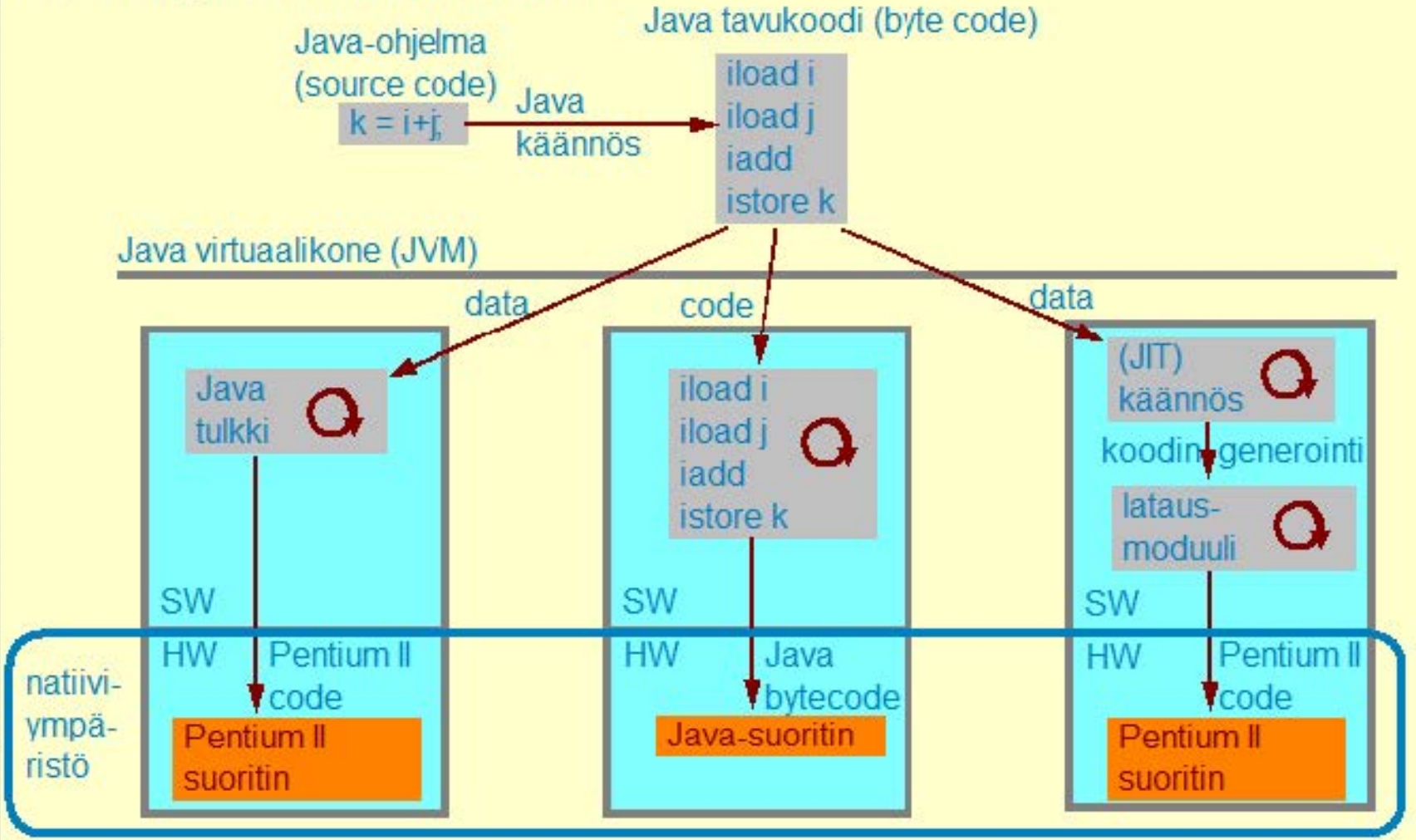
Toinen tavallisen ohjelman suoritusta vastaava Java-ohjelmien suoritustapa perustuu tavukoodin kääntämiseen kohdelaitteiston konekielelle. Tässä käännöksessa luodaan normaali ladattava ja linkitettävä moduuli, jota voidaan sitten suorittaa tavanomaiseen tapaan. On myös mahdollista, että käännös ja linkitys tehdään vasta dynaamisesti viime hetkellä eli siis silloin, kun uuteen Java-moduuliin tulee ensimmäinen viittaus. Tällaista käännöstä kutsutaan Just-In-Time -käännökseksi. Suorituksessa on nyt (joko samaan tai eri aikaan) kääntäjä ja varsinainen alkuperäistä Java-ohjelmaa vastaava prosessi. Kääntäjä lukee tavukoodin pelkkänä datana ja generoi kohdearkkitehtuurin mukaisia linkitys- ja latausmoduuleja.

Java-ohjelmien suoritustavat



Hyvin yleinen Java-ohjelmien suoritustapa perustuu tulkitsemiseen, jossa Java-virtuaalikone on toteutettu emulaattorina samaan tapaan kuin Titokone-ohjelmisto toimii ttk-91 -koneen emulaattorina. Java-tulkki on tavallinen, esimerkiksi C:llä kirjoitettu sovellus. Se lukee tavukoodin käskyjä ja emuloi niitä yksi kerrallaan. Varsinainen suorituksessa oleva ohjelma on siis Java-tulkki ja se lukee Java-ohjelmantavukoodia datana yksi konekäsky kerrallaan. Tulkin suoritus on hyvin samanlainen kuin Titokoneenkin - paitsi, että Java-virtuaalikonetta emuloivan Java-tulkin toiminta on täydellistä, sisältäen kaiken normaalin I/O:n, verkon ja tiedostojen käytön.

Java-ohjelmien suoritustavat

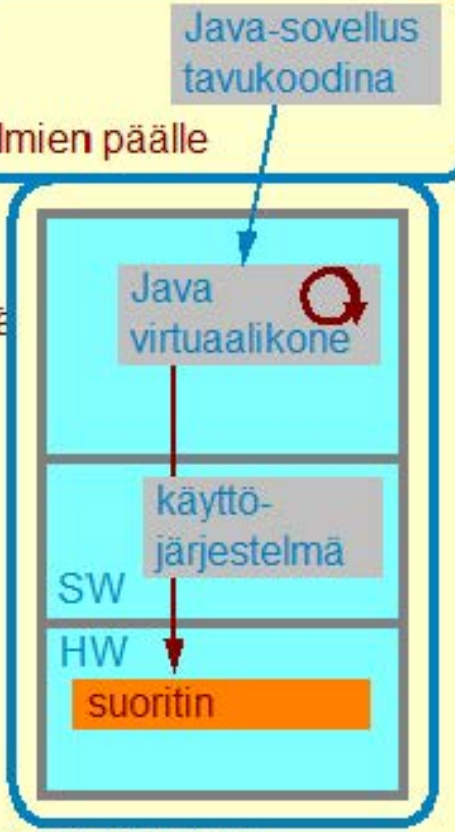


Java-ohjelmien suoritustapoja on siis 3 (tai 4, jos JIT-käännös erotetaan tavallisesta käännöksestä). Natiiviympäristö vaihtelee suoritustavasta toiseen. Java-suorittimen tapauksessa natiiviympäristönä eli varsinaisena suoritusaikaisena laitteistona on erikoisarkkitehtuuri, joka käsittelee tavukoodin käskyjä suoraan konekäskyinä. Yleensä kuitenkin natiiviympäristönä on tavanomainen geneerinen laitteisto, jonka konekielelle tavukoodi pitää joko etukäteen tai suoritusaikana kääntää, tai kuin Titokoneenkin - paitsi, että Java-virtuaalikonetta emuloivan Java-tulkin toiminta jolla suorittava tulkki-ohjelma emuloi tavukoodia käsky kerrallaan.

Java virtuaalikone (JVM, Java Virtual Machine)

Hypoteettinen suoritin, toteutus eri tavoin (HW ja/tai SW)

- geneerinen, suunniteltu emulointia varten
- helppo toteuttaa useimpien laitteistoalustojen ja/tai käyttöjärjestelmien päälle



Useita säikeitä voi olla suorituksessa samanaikaisesti

- emuloi moniprosessori suoritinta ja/tai moniajo käyttöjärjestelmää
- käytännössä suoritus yleensä silti vain yksi säie kerrallaan

Laitteisto ja tietorakenteet

- rekisterit, suorituksen aikaiset muistialueet (esim. pino)
- käskyjen suoritus (emulointi, käännös tai HW-toteutus)

Konekäskyt

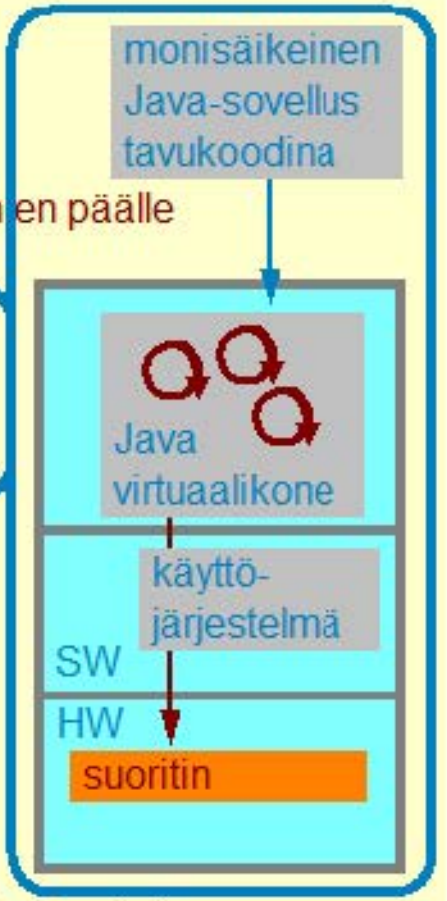
- 226 käskyä á 32 bittiä
- pinokone - kaikki käskyjen operandit yleensä pinossa eli käytännössä muistissa
- emulointi usein hidasta, koska natiivilaitteiston rekistereitä vaikea käytttää tehokkaasti
- tehokkaan natiivikonekielen koodin generointi vaikeata

Java virtuaalikone on siis hypoteettinen kone, tai eräänlainen tietokoneen määrittely. Määrittely on tarpeeksi täsmällinen ja kattava, jotta sille voidaan tehdä täydellisiä ohjelmia. Virtuaalikone voidaan sitten toteuttaa eri tavoin ja kaikki määrittelyt täyttävät ohjelmat toimivat eri toteutuksissa samalla tavalla. Määrittely on lähinnä suunniteltu emulointia varten sillä tavoin, että emuloinnin toteuttaminen olisi helpohkoa mille tahansa natiiviympäristölle ja mille tahansa käyttöjärjestelmälle. Tämä tavoite on aika hyvin toteutunutkin. Java virtuaalikoneen tavukoodilla kirjoitetut sovellukset toimivat liki kaikissa ympäristöissä samalla tavalla, useimmiten verkkoselaimiin integroidun emulaattorin avulla.

Java virtuaalikone (JVM, Java Virtual Machine)

Hypoteettinen suoritin, toteutus eri tavoin (HW ja/tai SW)

- geneerinen, suunniteltu emulointia varten
- helppo toteuttaa useimpien laitteistoalustojen ja/tai käyttöjärjestelmien päälle



Useita säikeitä voi olla suorituksessa samanaikaisesti

- emuloi moniprosessori suoritinta ja/tai moniajo käyttöjärjestelmää
- käytännössä suoritus yleensä silti vain yksi säie kerrallaan

Laitteisto ja tietorakenteet

- rekisterit, suorituksen aikaiset muistialueet (esim. pino)
- käskyjen suoritus (emulointi, käännös tai HW-toteutus)

Konekäskyt

- 226 käskyä á 32 bittiä
- pinokone - kaikki käskyjen operandit yleensä pinossa eli käytännössä muistissa
- emulointi usein hidasta, koska natiivilaitteiston rekistereitä vaikea käytttää tehokkaasti
- tehokkaan natiivikonekielen koodin generointi vaikeata

Virtuaalikoneen määrittelyssä on mukana monisäikeisyys. Tämän avulla Java-ohjelmat voivat aina olla monisäikeisiä riippumatta siitä, tukeeko käyttöjärjestelmä säikeitä, tai siitä, onko laitteistossa yksi tai useampi suoritin. Jos Java-sovellusta suoritetaan emuloimalla, niin useimmissä ympäristöissä emulaattorissa on vain yksi käyttöjärjestelmän tuntema säie ja Java-sovelluksen säikeitä suoritetaan todellisuudessa vain yksi kerrallaan. Toisaalta, on myös ympäristöjä ja laitteistoja, joissa monisäikeisen Java-sovelluksen usea säie on ihan oikeasti samaan aikaan suorituksessa.

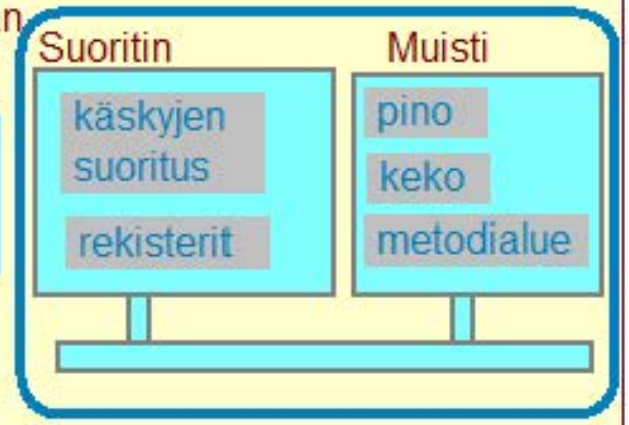
Java virtuaalikone (JVM, Java Virtual Machine)

Hypoteettinen suoritin, toteutus eri tavoin (HW ja/tai SW)

- geneerinen, suunniteltu emulointia varten
- helppo toteuttaa useimpien laitteistoalustojen ja/tai käyttöjärjestelmien päälle

Useita säikeitä voi olla suorituksessa samanaikaisesti

- emuloi moniprosessori suoritinta ja/tai moniajo käyttöjärjestelmää
- käytännössä suoritus yleensä silti vain yksi säie kerrallaan



Laitteisto ja tietorakenteet

- rekisterit, suorituksen aikaiset muistialueet (esim. pino)
- käskyjen suoritus (emulointi, käännös tai HW-toteutus)

Konekäskyt

- 226 käskyä á 32 bittiä
- pinokone - kaikki käskyjen operandit yleensä pinossa eli käytännössä muistissa
- emulointi usein hidasta, koska natiivilaitteiston rekistereitä vaikea käytttää tehokkaasti
- tehokkaan natiivikonekielen koodin generointi vaikeata

Virtuaalikone koostuu kuten kaikki muutkin koneet suorittimen määrittelystä, muistista ja oheislaitteista. Oheislaitteisiin emme puutu tässä enempää. Suorittimella on pieni joukko rekistereitä, jotka ovat virtuaalikonetta suorittavissa emulaattoreissa toteutettu omina tietorakenteinaan muistissa. Virtuaalikoneella on muutama erityismuistialue samaan tapaan kuin ns. normaalikoneiden esim. aktivointitietuepinot. Käskyjen suoritus tapahtuu emulaattorissa em. tietorakenteita manipuloimalla samalla tavalla kuin ttk-91 koneen simulointi tapahtui. Käskyjen suoritus on määritelty vain muistipaikan ja rekistereiden sisällön tasolla, joten esim. MMU:n tai väylän toteutus voi olla mitä vain. Tämä helpottaa JVM:n toteutusta ohjelmistolla ja laitteistolla.

Java virtuaalikone (JVM, Java Virtual Machine)

Hypoteettinen suoritin, toteutus eri tavoin (HW ja/tai SW)

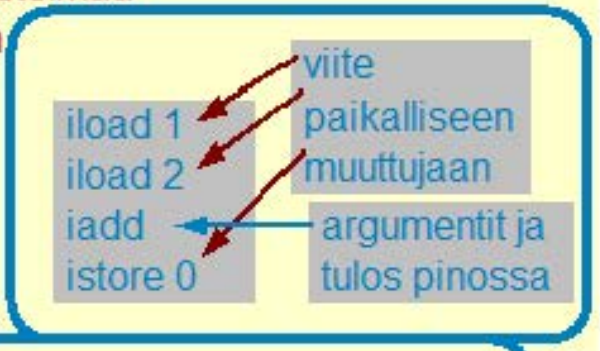
- geneerinen, suunniteltu emulointia varten
- helppo toteuttaa useimpien laitteistoalustojen ja/tai käyttöjärjestelmien päälle

Useita säikeitä voi olla suorituksessa samanaikaisesti

- emuloi moniprosessori suoritinta ja/tai moniajo käyttöjärjestelmää
- käytännössä suoritus yleensä silti vain yksi säie kerrallaan

Laitteisto ja tietorakenteet

- rekisterit, suorituksen aikaiset muistialueet (esim. pino)
- käskyjen suoritus (emulointi, käännös tai HW-toteutus)



Konekäskyt

- 226 käskyä á 32 bittiä
- pinokone - kaikki käskyjen operandit yleensä pinossa eli käytännössä muistissa
- emulointi usein hidasta, koska natiivilaitteiston rekistereitä vaikea käytttää tehokkaasti
- tehokkaan natiivikonekielen koodin generointi vaikeata

Virtuaalikoneessa on 226 32-bittistä konekäskyä. JVM on ns. pinokone, missä kaikki laskennan tilapäistieto pidetään pinossa ja kaikki laskenta kohdistuu aina pinon pinnalla oleviin alkioihin. Tällä on etuna se, että käskyn operandeja ei tarvitse yleensä nimetä, koska niiden sijainti on oletusarviosesti pinon pinnalla. Huonona puolena tässä on se, että useimmat tavalliset nykyiset tietokonelaitteistot ovat rekisterikoneita ja niiden nopeus perustuu paljolti suorittimella olevien nimetteyjen rekistereiden tehokkaaseen käyttöön. Java-ohjelmia suoritettaessa näitä rekistereitä on vaikea käyttää, kun ne eivät ole mukana Java-ideologiassa.

Java virtuaalikone (JVM, Java Virtual Machine)

Hypoteettinen suoritin, toteutus eri tavoin (HW ja/tai SW)

- geneerinen, suunniteltu emulointia varten
- helppo toteuttaa useimpien laitteistoalustojen ja/tai käyttöjärjestelmien päälle

Useita säikeitä voi olla suorituksessa samanaikaisesti

- emuloi moniprosessori suoritinta ja/tai moniajo käyttöjärjestelmää
- käytännössä suoritus yleensä silti vain yksi säie kerrallaan

Laitteisto ja tietorakenteet

- rekisterit, suorituksen aikaiset muistialueet (esim. pino)
- käskyjen suoritus (emulointi, käännös tai HW-toteutus)

Konekäskyt

http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

- 226 käskyä á 32 bittiä
- pinokone - kaikki käskyjen operandit yleensä pinossa eli käytännössä muistissa
- emulointi usein hidasta, koska natiivilaitteiston rekistereitä vaikea käytttää tehokkaasti
- tehokkaan natiivikonekielen koodin generointi vaikeata

Java virtuaalikone on Sun Microsystems'in määrittelemä ja sen tarkemmat speksit löytyvät Sunin omilta sivuilta. Voitte katsella niitä, jos haluatte täsmällistä tietoa. Tämän kurssin yhteydessä noin tarkkoja speksejä ei kuitenkaan tarvita.

JVM pino (JVM Stack)

Koostuu kehyksistä (frame) ja operandipinosta (operand stack)

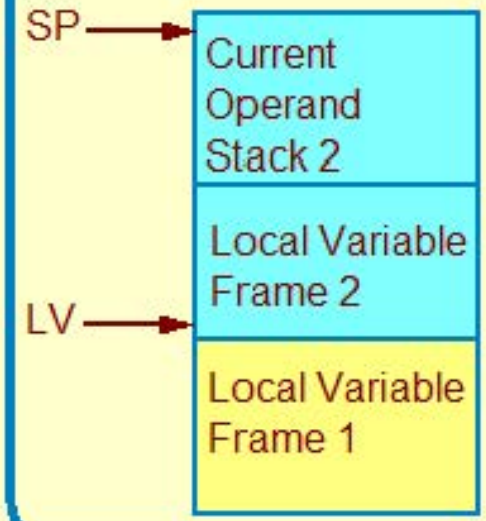
- kuten aktivointitietuepino
- nykykehys + operandipino = nyky-aktivointitietue + rekisterit
- kehyksessä parametrit, paikalliset muuttujat ja linkitys aikaisempaan kehykseen
- operandipinossa välitulokset ja kutsuttavan metodin kehys, metodin paluuarvo
- kutsussa operandipinosta tulee kutsuttavan metodin kehys

Käyttö

- kokonaisille kehyksille kerrallaan (push/pop)
- operandipino puretaan/rakennetaan alkioittain (push/pop)
  - kuten aktivointitietueen purku/rakentaminen

Allokoidaan (Java-ympäristön) omasta keosta

- ei tarvitse toteuttaa yhtenäisenä muistialueena
  - kukin kehys kuitenkin yhtenäinen
- rajallinen tai dynaamisesti laajennettavissa
- tila loppu ⇨ StackOverflowError, OutOfMemoryError



Java virtuaalikoneen pino vastaa aika lailla tavallista aktivointitietuepinoa. Eroja kuitenkin on. Pinossa on kahdenlaisia alkioita: kehyksiä, jotka vastaavat tavallista aktivointitietuetta, ja operandipino, johon talletetaan kaikki laskennan välitulokset. Operandipinossa pidetään siis kaikki se tieto, mikä tavallisissa järjestelmisä pidetään rekistereissä. Useimmissa Java virtuaalikoneen toteutuksissa koko pino ja siten myös siinä oleva operandipino ovat keskusmuistissa, joten laskennan välituloksiin viittaminen tapahtuu aina vähän tehottomasti muistin kautta. Rekisteri SP osoittaa operandipinon pinnalle ja rekisteri LV nykyisen kehyksen alkuun.

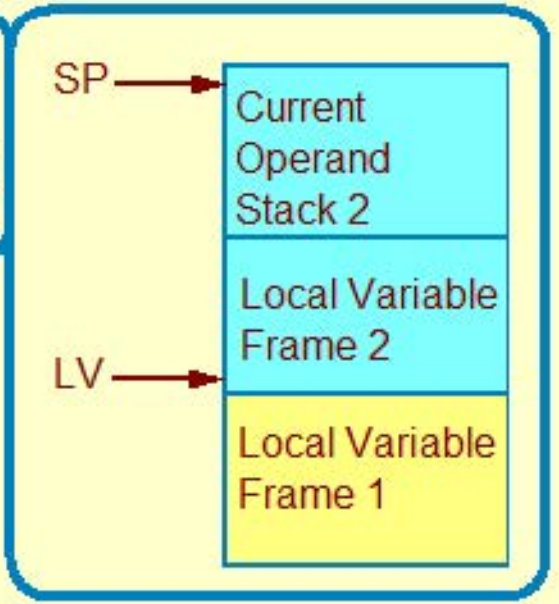
JVM pino (JVM Stack)

Koostuu kehyksistä (frame) ja operandipinosta (operand stack)

- kuten aktivointitietuepino
- nykykehys + operandipino = nyky-aktivointitietue + rekisterit
- kehyksessä parametrit, paikalliset muuttujat ja linkitys aikaisempaan kehykseen
- operandipinossa välitulokset ja kutsuttavan metodin kehys, metodin paluuarvo
- kutsussa operandipinosta tulee kutsuttavan metodin kehys

Käyttö

- kokonaisille kehyksille kerrallaan (push/pop)
- operandipino puretaan/rakennetaan alkioittain (push/pop)
  - kuten aktivointitietueen purku/rakentaminen



Allokoidaan (Java-ympäristön) omasta keosta

- ei tarvitse toteuttaa yhtenäisenä muistialueena
  - kukin kehys kuitenkin yhtenäinen
- rajallinen tai dynaamisesti laajennettavissa
- tila loppu ⇨ StackOverflowError, OutOfMemoryError

Kehyksiä varataan ja vapautetaan kokonaisuuksina metodien kutsujen ja niistä paluiden yhteydessä. Myös tavallisissa arkkitehtuureissa esim. EXIT-käskyn yhteydessä aktivointitietuetta purettiin, mutta JVM:n RETURN-käsky tekee enemmän ja vapauttaa kokokehyksen. Metodin kutsukäsky (invokevirtual) rakentaa melkein koko kehyksen, kun tavallisten arkkitehtuurien aktivointitietue rakennetaan pikkuhiljaa sana kerrallaan. Operandipinoa sen sijaan rakennetaan ja puretaan yksi sana kerrallaan push-, pop- ja muilla käskyillä. Pääosa tästä työstä on normaalia laskentaa, mutta osa on myös valmistautumista uuden metodin kutsuun.

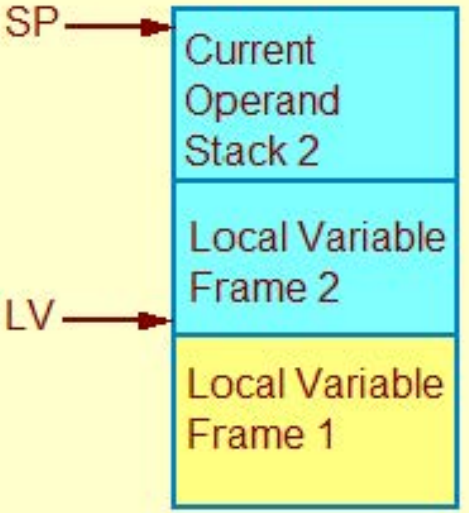
JVM pino (JVM Stack)

Koostuu kehyksistä (frame) ja operandipinosta (operand stack)

- kuten aktivointitietuepino
- nykykehys + operandipino = nyky-aktivointitietue + rekisterit
- kehyksessä parametrit, paikalliset muuttujat ja linkitys aikaisempaan kehykseen
- operandipinossa välitulokset ja kutsuttavan metodin kehys, metodin paluuarvo
- kutsussa operandipinosta tulee kutsuttavan metodin kehys

Käyttö

- kokonaisille kehyksille kerrallaan (push/pop)
- operandipino puretaan/rakennetaan alkioittain (push/pop)
  - kuten aktivointitietueen purku/rakentaminen



Allokoidaan (Java-ympäristön) omasta keosta

- ei tarvitse toteuttaa yhtenäisenä muistialueena
  - kukin kehys kuitenkin yhtenäinen
- rajallinen tai dynaamisesti laajennettavissa
- tila loppu ⇨ StackOverflowError, OutOfMemoryError

Java virtuaalikoneen pino allokoidaan Java-ympäristön hallitsemasta omasta keosta, joka siis on osa käyttöjärjestelmän Java-ympäristölle antamaa keskusmuistia. Pinon ei tarvitse muodostaa yhtenäistä muistialuetta, koska kehykset on linkitetty toisiinsa osoitteiden avulla. Suuremmissa laitteissa Java-suoritusympäristö voi allokoida lisää muistia JVM-pinolle, jos tila loppuu. Näin ei tietenkään voi aina tehdä, esimerkiksi rajallisesti muistia sisältävissä Java-kämmenmikroissa. Tilan loppumista hallitaan valmiiksi määriteltyjen Java-keskeytysten avulla, joten Java-sovellus voi reagoida tilan loppumiseen myös itse.

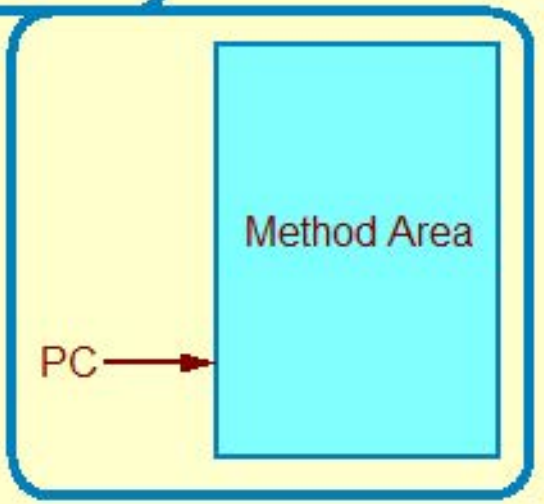
JVM metodialue (JVM Method Area)

Vastaa tavallista kääntäjän tuottamaa koodisegmenttiä

- metodialue on virtuaalikoneella suorittavan prosessin resurssi, ja siten yhteinen kaikille sen suorituksen yksiköille eli säikeille
- säikeen oma PC seuraavaan tavukoodiseen konekäskyyn

Allokoidaan (Java-ympäristön) omasta keosta

- JVM huolehtii tilanvarauksesta automaattisesti
- rajallinen tai dynaamisesti laajennettavissa
- tila loppu ⇨ OutOfMemoryError



Java virtuaalikoneen ohjelmakoodi talletetaan metodialueelle, mikä vastaa täysin tavallisen järjestelmän muistissa olevaa koodisegmenttiä. Java virtuaalikone suorittaa vain yhtä Java-sovellusta ja metodialue on tuota sovellusta toteuttavan prosessin kaikkien säikeiden yhteiskäytössä. Tottakai prosessin jokaisella säikeellä on oma paikanlaskurinsa metodialueelle. Kun säie saa suoritusvuoron, niin JVM:n rekisteri PC ladataan tuon säikeen paikanlaskurin arvolla. Tavanomaiseen tapaan PC osoittaa siis seuraavaksi suoritettavaan (tavukoodiseen) konekäskyyn. Huomatkaa, että PC ei vahingossakaan voi osoittaa data-alueelle, vaan ainoastaan koodiin metodialueelle.

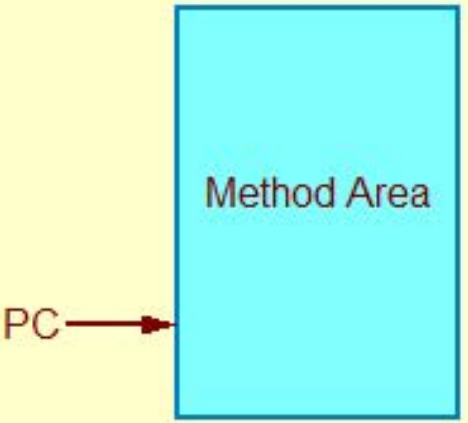
JVM metodialue (JVM Method Area)

Vastaa tavallista kääntäjän tuottamaa koodisegmenttiä

- metodialue on virtuaalikoneella suorittavan prosessin resurssi, ja siten yhteinen kaikille sen suorituksen yksiköille eli säikeille
- säikeen oma PC seuraavaan tavukoodiseen konekäskyyn

Allokoidaan (Java-ympäristön) omasta keosta

- JVM huolehtii tilanvarauksesta automaattisesti
- rajallinen tai dynaamisesti laajennettavissa
- tila loppu ⇨ OutOfMemoryError



Metodialueen tilanhallinta tapahtuu automaattisesti Java-ympäristön toimesta. Toteutuksesta riippuen, se voi pinon tapaan olla dynaamisesti laajennettavissa. Java-sovellukset voivat myös itse ottaa kantaa tilan loppumiseen valmiiksi määritellyn Java virtuaalikoneen keskeytyksen avulla.

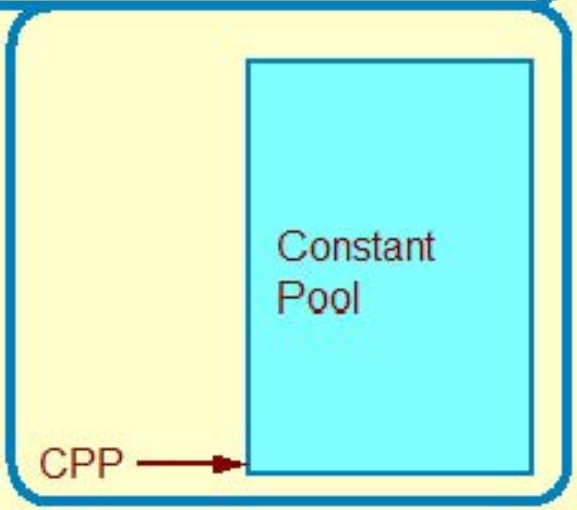
JVM vakioallas (JVM Runtime Constant Pool)

Vastaa vähän tavallista symbolitaulua

- tavalliset vakioarvot (käännösaikaiset literaalit)
- suoritusaikana dynaamisesti ratkaistavat attribuutit
  (symboliset viitteet luokkiin, liittymiin, metodeihin, kenttiin ja merkkijonoliteraaleihin)
- oma vakioallas jokaiselle luokalle (class) ja liittymälle (interface)
- rekisteri CPP osoittaa tällä hetkellä käytössä olevaan vakioaltaaseen

Talletetaan metodialueelle

- JVM huolehtii tilanvarauksesta automaattisesti
  luokan tai liittymän luomisen yhteydessä
- rajallinen tai dynaamisesti laajennettavissa
- tila loppu ⇨ OutOfMemoryError



Java virtuaalikoneen vakioallas vastaa vähän suoritusaikana mukana olevaa symbolitaulua sillä erotuksella, että Javassa vakioallasta käytetään aktiivisesti myös suoritusaikana symbolisten viitteiden ratkaisemisen. Normaalissa suoritusympäristössähän symbolitaulua ei oikeasti tarvita, koska kaikki symboliset viitteet on ratkottu joko käännös- tai linkitysaikana. Symbolitaulua pidetään mukana vain fiksujen virheilmoitusten tekemiseksi. Jokaiselle Javan luokalle tai liittymälle on oma vakioallas ja aktiivisena olevaan vakioaltaaseen osoittaa rekisteri CPP (Constant Pool Pointer). Jos vakioaltaan kautta viitattavalla symbolilla ei vielä ole arvoa, se voidaan dynaamisesti ratkaista viittaushetkellä!

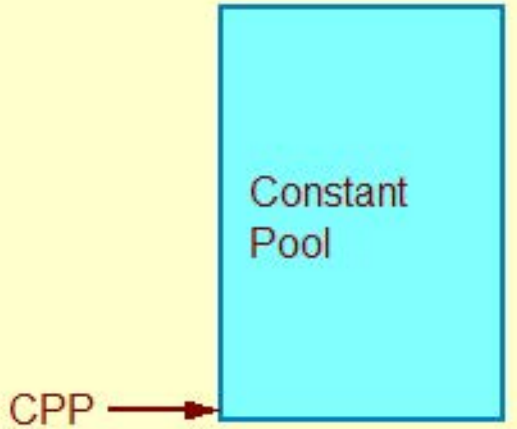
JVM vakioallas (JVM Runtime Constant Pool)

Vastaa vähän tavallista symbolitaulua

- tavalliset vakioarvot (käännösaikaiset literaalit)
- suoritusaikana dynaamisesti ratkaistavat attribuutit
  (symboliset viitteet luokkiin, liittymiin, metodeihin, kenttiin ja merkkijonoliteraaleihin)
- oma vakioallas jokaiselle luokalle (class) ja liittymälle (interface)
- rekisteri CPP osoittaa tällä hetkellä käytössä olevaan vakioaltaaseen

Talletetaan metodialueelle

- JVM huolehtii tilanvarauksesta automaattisesti
  luokan tai liittymän luomisen yhteydessä
- rajallinen tai dynaamisesti laajennettavissa
- tila loppu ⇨ OutOfMemoryError



Vakioallas luodaan automaattisesti luokan tai liittymän luomisen yhteydessä metodialueelta ja alkuaan kaikilla symboleilla on määrittelemätön arvo. Jos muistitilaa ei ole riittävästi, niin sitä voidaan jälleen joissakin JVM:n toteutuksissa dynaamisesti laajentaa.

JVM keko (JVM Heap)

Kaikki JVM tietorakenteet allokoidaan täältä

- pinot, metodialue, dynaamiset data-alueet
- voi olla dynaamisesti laajennettavissa
- ei tarvitse muodostaa yhtenäistä muistialuetta käyttöjärjestelmän kannalta
- tila loppu ⇨ OutOfMemoryError

Automaattinen roskien keruu (garbage collection)

- varattu mutta käyttämättömänä oleva muistialueet vapautuvat automaattisesti uusiokäyttöön
- Java-ohjelmissa ei tarvita eksplisiittistä *free*-operaatiota
  - vrt. olioiden luonti *new*-operaatiolla
- JVM:n roskien kerääjä (garbage collector) voi käynnistyä milloin vain ja suorittaa työnsä muun laskennan odottaessa
  - huono asia aika-kriittisissä sovelluksissa (esim. tosiaikasovellukset)
  - joissain JVM toteutuksissa roskien keruun käynnistymistä voi säätää tai sitten roskien keruu voi toimia omana prosessinaan JVM:n rinnalla samanaikaisesti

Java virtuaalikone ylläpitää kekoa, josta kaikki virtuaalikoneen käyttämät muistialueet varataan. Keon koko voi olla kiinteä tai dynaamisesti tarpeen mukaan vaihteleva, jos laitteisto ja käyttöjärjestelmä sen sallivat. Esimerkiksi, tulkitsemalla toteutetulle virtuaalikoneelle käyttöjärjestelmä voi aluksi antaa vain aika pienen määrän muistitilaa kekoa varten ja tarpeen mukaan keon kokoa voi sitten myöhemmin kasvattaa. Toisaalta, virtuaalimuistiteknologian avulla keko voi olla alkuaankin valtava, mutta osa siitä voi olla täysin olematonta tai sijaita ainoastaan tukimuistissa levyllä. Tilan loppumista ilmaisee etukäteen määritelty virhetilanne.

JVM keko (JVM Heap)

Kaikki JVM tietorakenteet allokoidaan täältä

- pinot, metodialue, dynaamiset data-alueet
- voi olla dynaamisesti laajennettavissa
- ei tarvitse muodostaa yhtenäistä muistialuetta käyttöjärjestelmän kannalta
- tila loppu ⇨ OutOfMemoryError

Automaattinen roskien keruu (garbage collection)

- varattu mutta käyttämättömänä oleva muistialueet vapautuvat automaattisesti uusiokäyttöön
- Java-ohjelmissa ei tarvita eksplisiittistä *free*-operaatiota
  - vrt. olioiden luonti *new*-operaatiolla
- JVM:n roskien kerääjä (garbage collector) voi käynnistyä milloin vain ja suorittaa työnsä muun laskennan odottaessa
  - huono asia aika-kriittisissä sovelluksissa (esim. tosiaikasovellukset)
  - joissain JVM toteutuksissa roskien keruun käynnistymistä voi säätää tai sitten roskien keruu voi toimia omana prosessinaan JVM:n rinnalla samanaikaisesti

Keosta varataan siis muistitilaa dynaamisesti Java-ohjelmien suoritusaikana ja vastaavasti käytössä olevaa muistitilaa vapautuu aina aika ajoin suoritusaikana. Vapautuvien tilojen hallintaan on periaatteessa kaksi eri menetelmää. Ensimmäisessä menetelmässä ohjelmassa pitää aina vapauttaa muistitila eksplisiittisesti free-operaatiolla. Toisessa vaihtoehdossa suoritusympäristö tekee aika ajoin ns. roskien keruu -operaation, jossa etsitään kaikki muistialueet, jotka eivät enää ole käytössä. Javassa on käytössä tämä roskien keruu, joten Java-ohjelmien (ja ohjelmoijien) ei tarvitse murehtia lainkaan käytöstä poistettavien muistialueiden vapauttamisesta.

JVM keko (JVM Heap)

Kaikki JVM tietorakenteet allokoidaan täältä

- pinot, metodialue, dynaamiset data-alueet
- voi olla dynaamisesti laajennettavissa
- ei tarvitse muodostaa yhtenäistä muistialuetta käyttöjärjestelmän kannalta
- tila loppu ⇨ OutOfMemoryError

Automaattinen roskien keruu (garbage collection)

- varattu mutta käyttämättömänä oleva muistialueet vapautuvat automaattisesti uusiokäyttöön
- Java-ohjelmissa ei tarvita eksplisiittistä *free*-operaatiota
  - vrt. olioiden luonti *new*-operaatiolla
- JVM:n roskien kerääjä (garbage collector) voi käynnistyä milloin vain ja suorittaa työnsä muun laskennan odottaessa
  - huono asia aika-kriittisissä sovelluksissa (esim. tosiaikasovellukset)
  - joissain JVM toteutuksissa roskien keruun käynnistymistä voi säätää tai sitten roskien keruu voi toimia omana prosessinaan JVM:n rinnalla samanaikaisesti

Huonona puolena roskien keruussa on, että se on työlästä. Kaikki muistialueet on käytävä läpi ja tarkistettava ovatko ne vielä käytössä vai eivät. Tähän voi kulua paljon aikaa ja roskien keruu voi vielä käynnistyä ihan milloin vain. Yleensä se käynnistyy silloin, kun vapaana oleva muistitilan määrä menee jonkun kynnysarvon alle tai kun keosta ei vaan löydy tarpeeksi suurta yhtenäistä muistialuetta sitä tarvittaessa. Tälle ongelmalle on jonkin sortin ratkaisuja, mutta yleisesti ottaen roskien keruu on suuri este Javan käytölle tosiaikasovelluksissa.

JVM natiivimenetelmien pinot (Native Method Stacks)

Virtuaalikoneen toteutus voi käyttää tavallisia pinoja ("*C stacks*") muilla kielillä (esim. *C*) kuin Javalla kirjoitettujen metodien tukena

- suurta laskentanopeutta vaativa metodi voidaan kirjoittaa C:llä Javan asemesta

Voidaan käyttää myös Java-tulkin toteutuksessa

- Java-tulkin voi kirjoittaa tehokkaasti esimerkiksi C:llä

Ei tarvita JVM toteutuksissa, joissa ei ole natiivimetodeja

Voivat olla kiinteän kokoisia tai dynaamisesti laajennettavissa

- tila loppu ⇨ StackOverflowError, OutOfMemoryError

Java on joustava ohjelmoijan ympäristö, mutta Java virtuaalikoneen suoritus nykyisin käytössä olevilla laitteistolla on usein aika hidasta - siis suorittimen näkökulmasta, ei ihmisen! Suoritusnopeutta ja sidoksia käyttöjärjestelmään auttavat natiivimenetelmien pinot, joiden avulla Java-ohjelmat voivat kutsua myös aivan tavallisesti (siis ei Java virtuaalikoneessa) suoritettavia metodeja. Näiden natiivimetodien aliohjelmien kutsut ja kutsun toteuttavat aktivointitietueet voidaan sitten toteuttaa natiivimenetelmien pinoissa tavanomaiseen tapaan. Myös Java-tulkin toteutuksessa voidaan hyödyntää näitä laitteiston natiivikonekielellä toteutettuja aliohjelmia.

JVM natiivimenetelmien pinot (Native Method Stacks)

Virtuaalikoneen toteutus voi käyttää tavallisia pinoja ("*C stacks*") muilla kielillä (esim. *C*) kuin Javalla kirjoitettujen metodien tukena

- suurta laskentanopeutta vaativa metodi voidaan kirjoittaa C:llä Javan asemesta

Voidaan käyttää myös Java-tulkin toteutuksessa

- Java-tulkin voi kirjoittaa tehokkaasti esimerkiksi C:llä

Ei tarvita JVM toteutuksissa, joissa ei ole natiivimetodeja

Voivat olla kiinteän kokoisia tai dynaamisesti laajennettavissa

- tila loppu ⇨ StackOverflowError, OutOfMemoryError

Natiivimenetelmien pinoja ei tietenkään tarvitse tukea ympäristöissä, joissa niitä ei tarvita. Natiivimenetelmien pinot voivat muiden Java virtuaalikoneen rakenteiden tavoin olla joko kiinteän kokoisia tai dynaamisesti laajennettavissa.

JVM rekisterit (JVM registers)

PC osoittaa seuraavaan tavukoodiseen käskyyn metodialueelle

CPP osoittaa vakioaltaan alkuun

LV osoittaa nykykehyksen alkuun

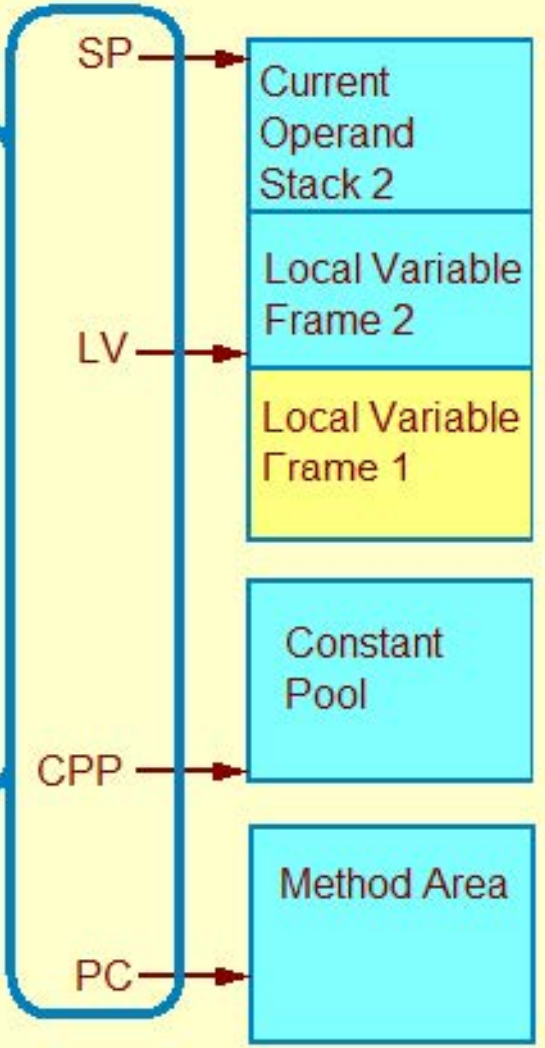
- vrt. FP ttk-91 -koneessa

SP osoittaa operandipinon huipulle

- vrt. SP ttk-91 -koneessa

Kaikki rekisterit implisiittisiä

- niitä ei erikseen nimetä missään konekäskyssä
- kussakin konekäskyssä on käytössä sen käskyn käyttämiin tiedonviittausmoodeihin liittyvät rekisterit



JVM:ssä on vain neljä rekisteriä, ja nekin ovat kaikki vain osoittimia työalueille. Kaikki välitulokset pidetään pinossa eikä siis tavanomaiseen tapaan suorittimen rekistereissä, kuten esimerkiksi ttk-91 tai Pentium arkkitehtuureissa. Useimmiten JVM toteutetaan ohjelmiston avulla tavanomaisissa pöytäkoneissa tai läppäreissä, jolloin työalueet toteutetaan muistialueina. Virtuaalikoneen rekisterit sentään voidaan sijoittaa laitteistorekistereihin, mutta jo operandipinon toteutus laiterekistereitä hyödyntäen on monimutkaista. JVM ei vain kuvaudu kovin hyvin tavanomaisen rekisteriarkkitehtuurin päälle.

JVM rekisterit (JVM registers)

PC osoittaa seuraavaan tavukoodiseen käskyyn metodialueelle

CPP osoittaa vakioaltaan alkuun

LV osoittaa nykykehyksen alkuun

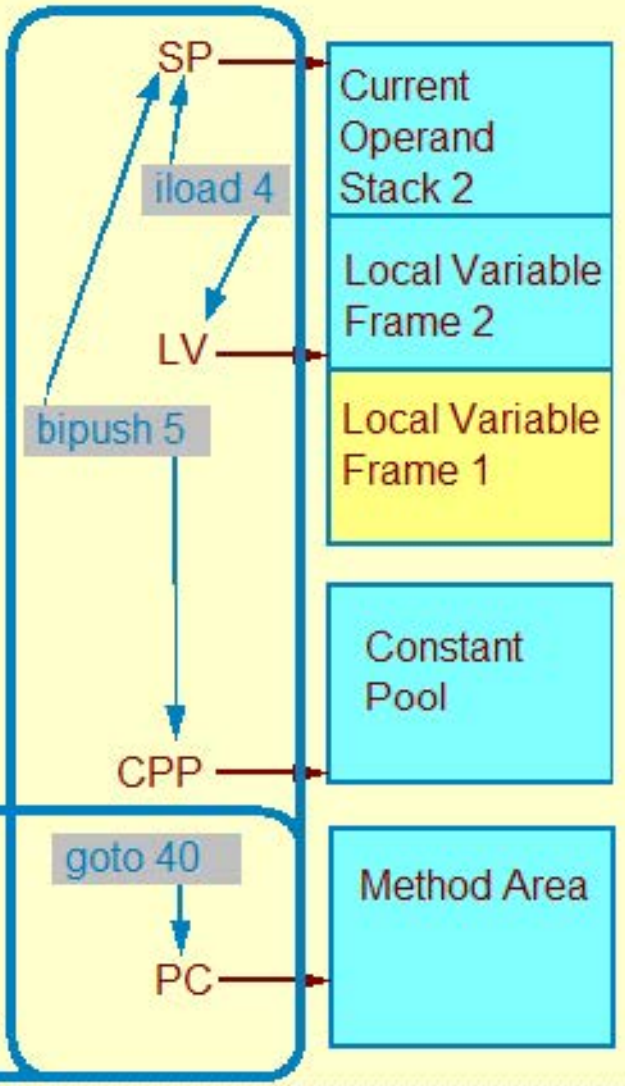
- vrt. FP ttk-91 -koneessa

SP osoittaa operandipinon huipulle

- vrt. SP ttk-91 -koneessa

Kaikki rekisterit implisiittisiä

- niitä ei erikseen nimetä missään konekäskyssä
- kussakin konekäskyssä on käytössä sen käskyn käyttämiin tiedonviittausmoodeihin liittyvät rekisterit



Toinen mielenkiintoinen piirre Javan virtuaalikoneen tavukoodissa on kaikkien rekistereiden implisiittisyys. Konekäskyissä ei siis koskaan nimetä yhtään rekisteriä. Iload-käskyn argumentti 4 viittaa 4. paikalliseen muuttujaan rekisterin LV viittaamassa nykykehyksessä. Iload-käsky talletaa tuon 4. paikallisen kokonaislukuarvoisen muuttujan arvon rekisterin SP osoittaman operandipinon huipulle ja samalla muuttaa rekisterin SP arvoa vastaavasti. Bipush-käsky laittaa 5. vakion rekisterin CPP osoittamasta vakioaltaasta operandipinoon ja goto-käsky lisää arvon 40 paikanlaskurin arvoon.

JVM kehys (JVM Frame)

Luodaan metodin kutsun yhteydessä

Ennen kutsua

- olion viite ja parametrit pinoon

Kutsussa luodaan uusi kehys

- paikalliset muuttujat
- linkitys, PC:n ja LV:n vanhat arvot talteen

Metodin aikana

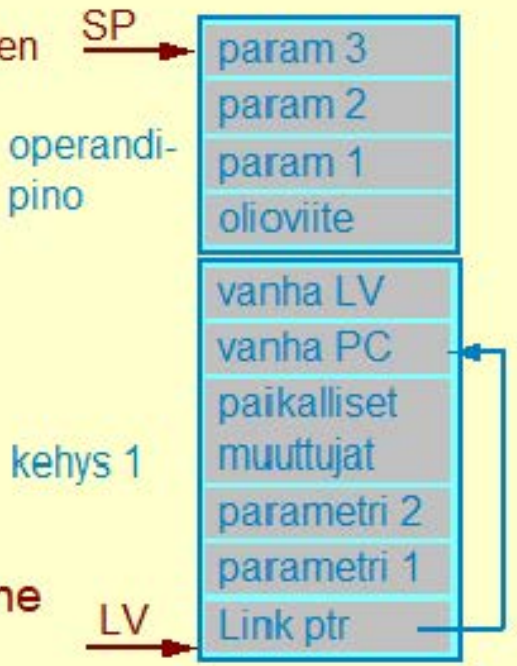
- paluuarvo pinon pinnalle

Puretaan metodista poistuttaessa

- linkitys puretaan, PC ja LV ladataan
- paluuarvo

Dynaamisen linkityksen toteutusväline

Keskeytysten toteutusväline



Metodin kutsu toteutetaan JVM:ssa hyvin samanlaisella tavalla kuin muissakin konearkkitehtuureissa. Kaikki tieto välittyy pinon kautta, joten viitattavan olion osoite ja kaikki parametrit talletetaan ensin operandipinoon. Erona ttk-91 arkkitehtuuriin on se, että pinoon laitetaan myös viitattavan olion viite - ttk-91:ssähän viitattava rutiini annettiin vasta aliohjelman kutsukäskyssä.

JVM kehys (JVM Frame)

Luodaan metodin kutsun yhteydessä

Ennen kutsua

- olion viite ja parametrit pinoon

Kutsussa luodaan uusi kehys

- paikalliset muuttujat
- linkitys, PC:n ja LV:n vanhat arvot talteen

Metodin aikana

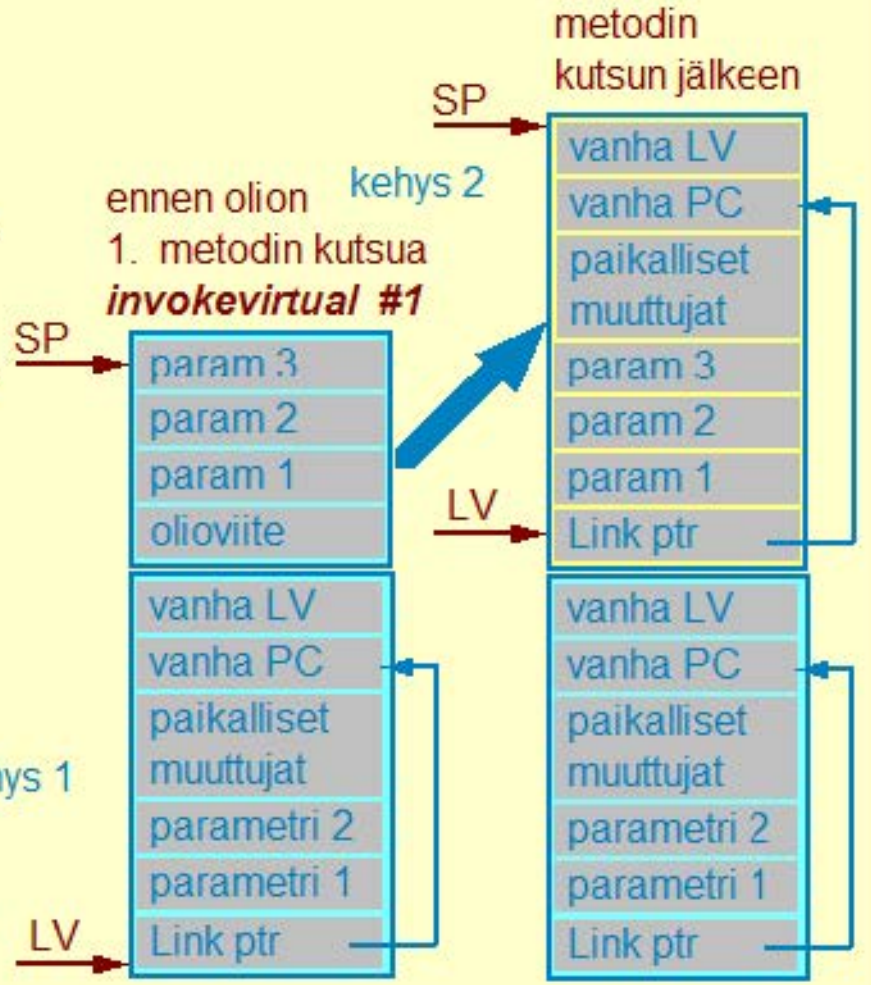
- paluuarvo pinon pinnalle

Puretaan metodista poistuttaessa

- linkitys puretaan, PC ja LV ladataan
- paluuarvo

Dynaamisen linkityksen toteutusväline

Keskeytysten toteutusväline



Varsinaisen metodin kutsukäskyn yhteydessä operandipinosta luodaan uuden metodin kehys, josta varataan tilaa paikallisille muuttujille. Sinne talletetaan myöskin kutsuhetken PC ja LV eli paluuosoite ja suoritusympäristö. Aikaisemman olioviitteen paikalle laitetaan kehyksessä linkkiosoitin, joka osoittaa kehyksen lopussa olevaan kutsuhetken PC:n talletuspaikkaan.

JVM kehys (JVM Frame)

Luodaan metodin kutsun yhteydessä

Ennen kutsua

- olion viite ja parametrit pinoon

Kutsussa luodaan uusi kehys

- paikalliset muuttujat
- linkitys, PC:n ja LV:n vanhat arvot talteen

Metodin aikana

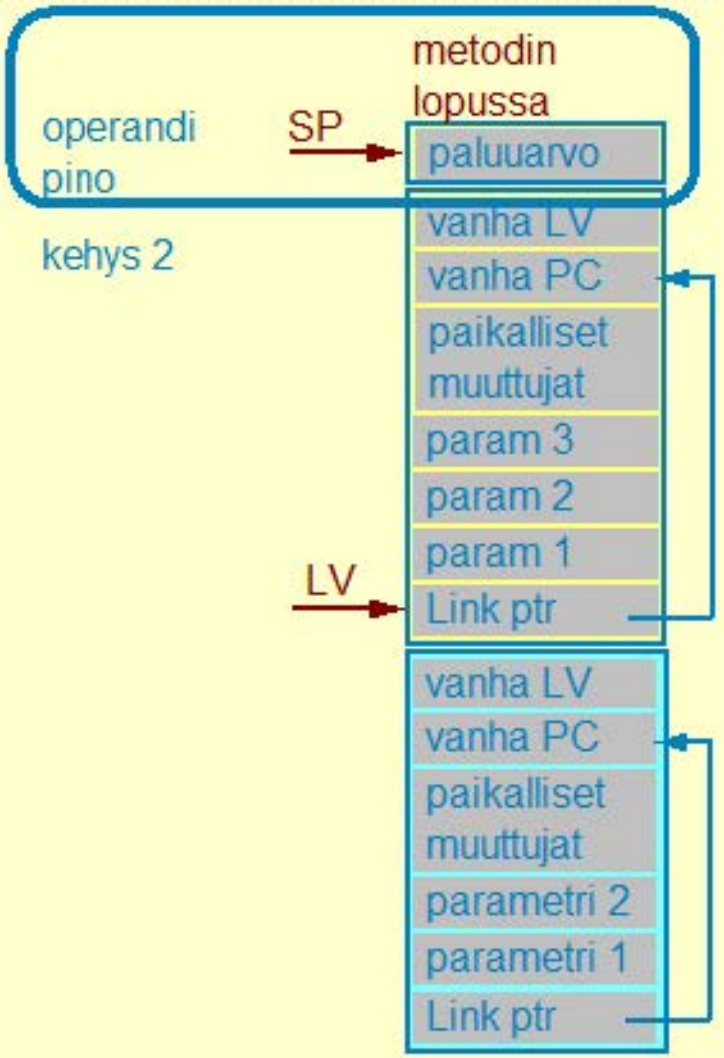
- paluuarvo pinon pinnalle

Puretaan metodista poistuttaessa

- linkitys puretaan, PC ja LV ladataan
- paluuarvo

Dynaamisen linkityksen toteutusväline

Keskeytysten toteutusväline



Metodin suoritus päättyy sitten aikanaan. Jos metodi palauttaa jonkun arvon, niin suorituksen lopussa se on ainoana alkiona operandipinossa.

JVM kehys (JVM Frame)

Luodaan metodin kutsun yhteydessä

Ennen kutsua

- olion viite ja parametrit pinoon

Kutsussa luodaan uusi kehys

- paikalliset muuttujat
- linkitys, PC:n ja LV:n vanhat arvot talteen

Metodin aikana

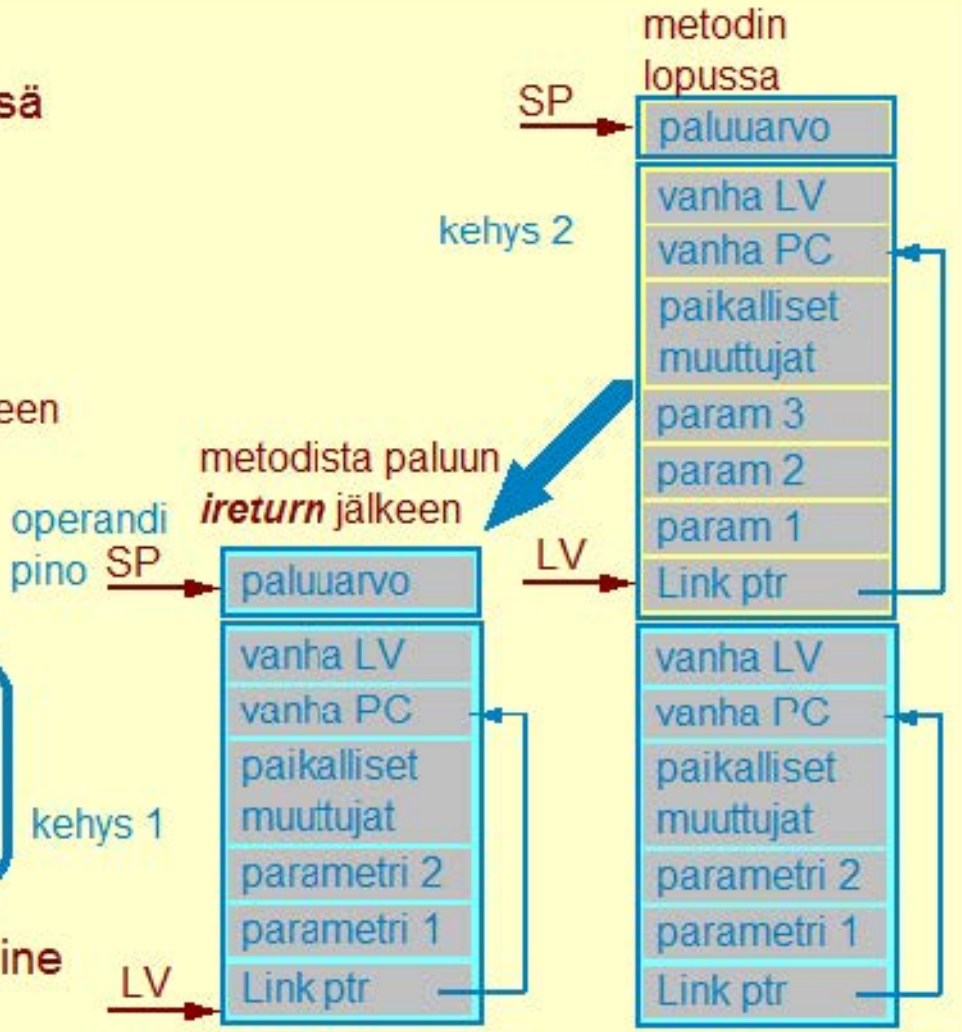
- paluuarvo pinon pinnalle

Puretaan metodista poistuttaessa

- linkitys puretaan, PC ja LV ladataan
- paluuarvo

Dynaamisen linkityksen toteutusväline

Keskeytysten toteutusväline



Metodista paluukäskyn yhteydessä koko sen kehys puretaan ja tilalle jää ainoastaan metodin paluuarvo operandipinon pinnalle, josta se on helppo kutsuvan metodin ottaa käyttöön.

JVM kehys (JVM Frame)

Luodaan metodin kutsun yhteydessä

Ennen kutsua

- olion viite ja parametrit pinoon

Kutsussa luodaan uusi kehys

- paikalliset muuttujat
- linkitys, PC:n ja LV:n vanhat arvot talteen

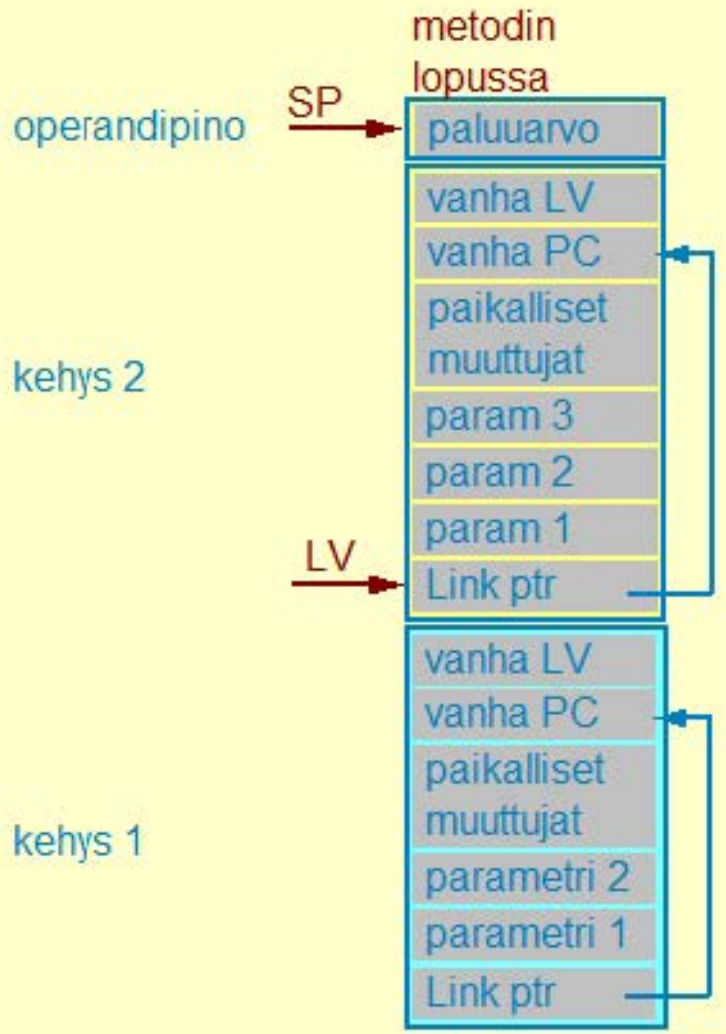
Metodin aikana

- paluuarvo pinon pinnalle

Puretaan metodista poistuttaessa

- linkitys puretaan, PC ja LV ladataan
- paluuarvo

Dynaamisen linkityksen toteutusväline
Keskeytysten toteutusväline



Tätä samaa menettelyä on nyt helppo muokata dynaamisen linkityksen toteuttamiseen samoin kuin keskeytysten hallintaan. Emme kuitenkaan tarkastele näitä asioita tämär tarkempaa tällä kertaa.

Paikalliset muuttujat

Talletettu kehykseen

- sana (32 bittiä) kerrallaan
- kaksoistarkkuuden luvut käyttävät 2 peräkkäistä sanaa
- big-endian tavujärjestys

Viittaukset indekseinä rekisterin LV suhteen

iload 2	lataa 2. parametrin arvo pinoon
dload 6	lataa 3. muuttujan arvot (2 sanaa) pinoon

Operandipino sisältää välitulokset ja paluuarvon
Pinokone eli ALU-käskyissä 0 kpl nimettyjä operandeja

iadd	poista 2 kokonaislukua pinosta, laske ne yhteen ja laita tulos pinoon

Paikalliset muuttujat talletetaan 32-bitin sanoissa kehykseen. JVM tukee myös pitkiä kokonaislukuja ja kaksoistarkkuuden liukulukuja, jotka molemmat talletetaan aina kahteen peräkkäiseen 32-bitin sanaan. Vastaavasti taulukot ja tietueet talletetaan useampisanaisena tietona peräkkäisiin sanoihin. Tavujärjestys on big-endian.

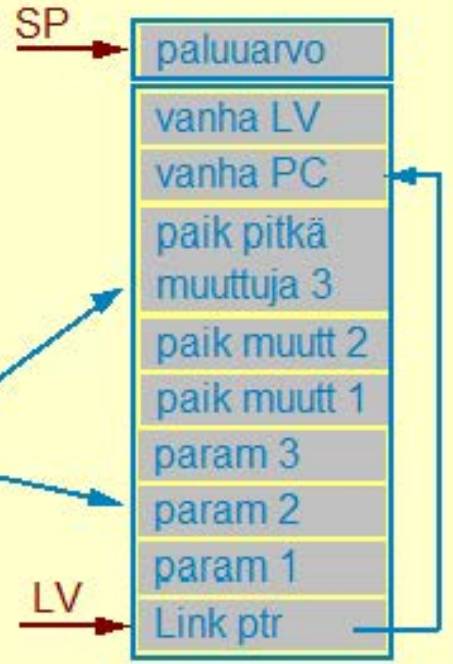
Paikalliset muuttujat

Talletettu kehykseen

- sana (32 bittiä) kerrallaan
- kaksoistarkkuuden luvut käyttävät 2 peräkkäistä sanaa
- big-endian tavujärjestys

Viittaukset indekseinä rekisterin LV suhteen

iload 2	lataa 2. parametrin arvo pinoon
dload 6	lataa 3. muuttujan arvot (2 sanaa) pinoon



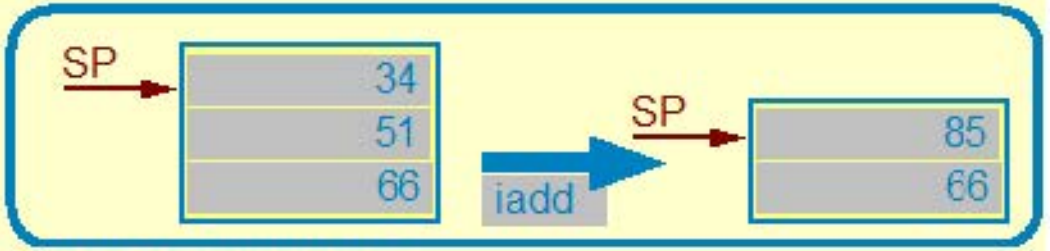
Operandipino sisältää välitulokset ja paluuarvon

Pinokone eli ALU-käskyissä 0 kpl nimettyjä operandeja

iadd	poista 2 kokonaislukua pinosta, laske ne yhteen ja laita tulos pinoon

Kaikki viittaukset kehyksessä olevaan tietoon tapahtuvat LV-rekisterin kautta. Tiedon osoite on siten aina positiivinen kokonaisluku, joka on suhteessa LV:n arvoon.

Paikalliset muuttujat



Talletettu kehykseen

- sana (32 bittiä) kerrallaan
- kaksoistarkkuuden luvut käyttävät 2 peräkkäistä sanaa
- big-endian tavujärjestys

Viittaukset indekseinä rekisterin LV suhteen

iload 2	lataa 2. parametrin arvo pinoon
dload 6	lataa 3. muuttujan arvot (2 sanaa) pinoon

Operandipino sisältää välitulokset ja paluuarvon
Pinokone eli ALU-käskyissä 0 kpl nimettyjä operandeja

iadd	poista 2 kokonaislukua pinosta, laske ne yhteen ja laita tulos pinoon

Kaikki välitulokset pidetään operandipinossa ja niihin pystytään viittaamaan ainoastaan SP:n kautta. JVM on pinokone, mikä tarkoittaa että aritmetikkakäskyissä kaikki operandit löytyvät oletusarvoisista paikoista pinosta ja tulos talletetaan aina myos pinoon. Lisaksi operandit poistetaan pinosta aritmetiikkaoperaation aikana, mika on usein ärsyttävää. Paljon käytettävää operandia (esim. muuttujan X arvoa) pitää olla yhtämittaa kopioimassa tai monistamassa pinon pinnalle uusia operaatioita varten.

JVM tiedonosoitusmoodit

Tietotyypit

- kokonaisluku tai boolean (i), pitkä kokonaisluku (l)
- liukuluku (f), kaksoistarkkuden liukuluku (d), olio (a)
- tiedon pituus: byte (b), char (c), short (s)

```
iload
lload
fadd
sipush
aload
```

konekäskyn ensimmäiset kirjaimet kertovat pituuden ja/tai tyypin

Välitön operandi

```
iinc  2  34
```

Indeksoitu viittaus

- parametri tai paikallinen muuttuja

```
iinc  2  34
```

Pino-osoitus

```
iadd
```

Taulukko-osoitus

```
aload 5
iload 4
iaload
istore 3
```


JVM:lla on viisi perustietotyyppiä: lyhyet ja pitkät kokonaisluvut ja liukuluvut sekä pointterit. Lyhyempiä tietolajeja ovat vielä tavut, 2-tavuiset merkit ja lyhyet kokonaisluvut. Useasta konekäskystä on omat muotonsa usealle siihen sopivalle tietotyypille - kyseinen tietotyyppi on sitten koodattu operaatiokoodiin.

JVM tiedonosoitusmoodit

iload
lload
fadd
sipush
aload

Tietotyypit

- kokonaisluku tai boolean (i), pitkä kokonaisluku (l)
- liukuluku (f), kaksoistarkkuden liukuluku (d), olio (a)
- tiedon pituus: byte (b), char (c), short (s)

Välitön operandi

```
x += 34;
iinc  2 34
```

jälkimmäinen operandi on konekäskyssä oleva vakio

Indeksoitu viittaus

- parametri tai paikallinen muuttuja

```
iinc  2 34
```

Pino-osoitus

```
iadd
```

Taulukko-osoitus

```
aload 5
iload 4
iaload
istore 3
```


Varsinaisista tiedonosoitusmoodeista ensimmäinen on välitön operandi, joka on tavalliseen tapaan konekäskyyn talletettu lyhyt vakio.

JVM tiedonosoitusmoodit

```
iload
lload
fadd
sipush
aload
```

Tietotyypit

- kokonaisluku tai boolean (i), pitkä kokonaisluku (l)
- liukuluku (f), kaksoistarkkuden liukuluku (d), olio (a)
- tiedon pituus: byte (b), char (c), short (s)

Välitön operandi

```
iinc  2  34
```

Indeksoitu viittaus

- parametri tai paikallinen muuttuja

```
x += 34;
iinc  2  34
```

ensimmäisen operandin (x) osoite on (LV)+2

Pino-osoitus

```
iadd
```

Taulukko-osoitus

```
aload 5
iload 4
iaload
istore 3
```


Indeksoitu viittaus tarkoittaa nykykehyksessä olevaa parametria tai paikallista muuttujaa. Niihin viittaus tapahtuu 8-bittisen indeksin avulla, mikä on viitatun tiedon suhteellinen osoite LV:n osoittamassa kehyksessä.

JVM tiedonosoitusmoodit

Tietotyypit

- kokonaisluku tai boolean (i), pitkä kokonaisluku (l)
- liukuluku (f), kaksoistarkkuden liukuluku (d), olio (a)
- tiedon pituus: byte (b), char (c), short (s)

```
iload
lload
fadd
sipush
aload
```

Välitön operandi

```
iinc  2  34
```

Indeksoitu viittaus

- parametri tai paikallinen muuttuja

```
iinc  2  34
```

Pino-osoitus

```
a = b+c;
iadd
```

operandit ja tulos SP:n osoittaman pinon pinnalla

Taulukko-osoitus

```
aload 5
iload 4
iaload
istore 3
```


Useissa käskyissä yksi tai useampi viite kohdistuu operandipinoon, jolloin niiden sijainti löytyy pino-osoittimen SP avulla. Pino-viittausten yhteydessä on tyypillistä, että myös SP:n arvo vaihtuu.

JVM tiedonosoitusmoodit

```
iload
lload
fadd
sipush
aload
```

Tietotyypit

- kokonaisluku tai boolean (i), pitkä kokonaisluku (l)
- liukuluku (f), kaksoistarkkuden liukuluku (d), olio (a)
- tiedon pituus: byte (b), char (c), short (s)

Välitön operandi

```
iinc  2  34
```

Indeksoitu viittaus

- parametri tai paikallinen muuttuja

```
iinc  2  34
```

Pino-osoitus

```
iadd
```

Taulukko-osoitus

a = T[i];

```
aload 5
iload 4
iaload
istore 3
```

taulukon osoite pinoon
taulukon indeksi pinoon
korvaa pinon pinnalla olevat taulukon alkuosoite ja indeksi kyseisellä taulukon alkiolla

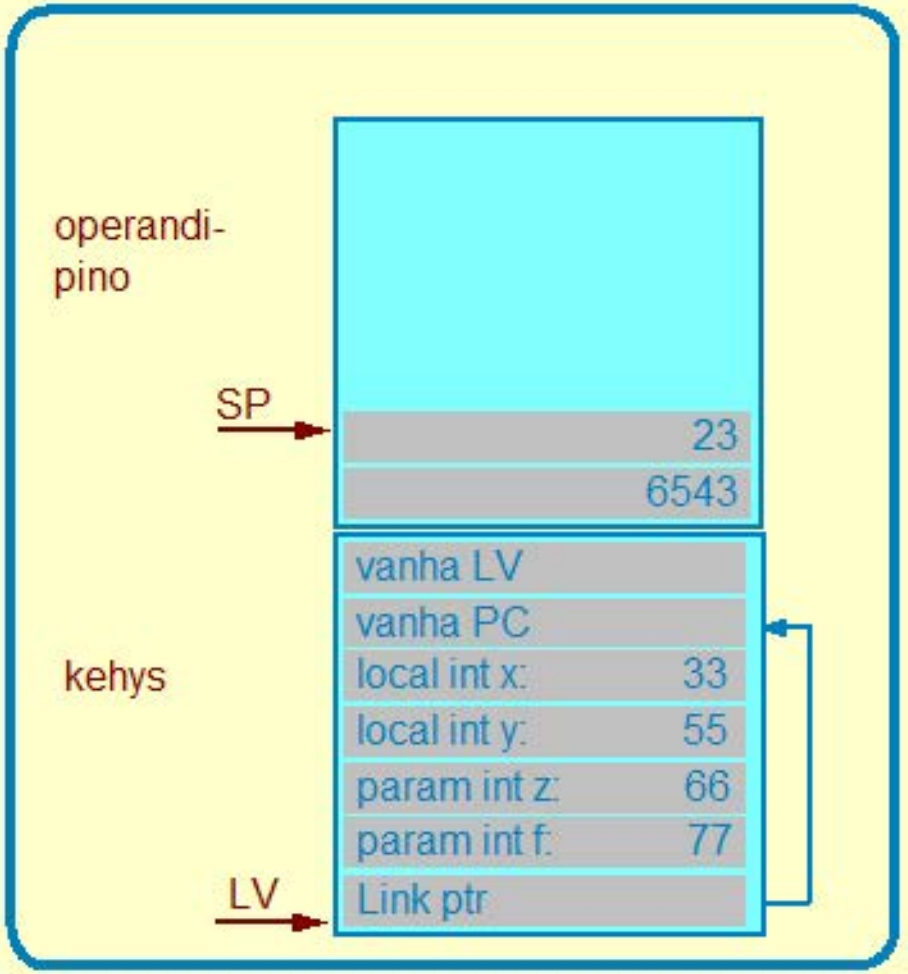
Taulukkoihin viittaaminen on hämmästyttävän monimutkaista. Joka taulukkoviitteellä meidän tulee ensin ladata pinoon sekä taulukon osoite että vitatun alkion sijainti-indeksi taulukossa. Varsinainen viite taulukkoalkioon voidaan tehdä vasta tämän jälkeen. Ei ihme, että Java ei ole ensisijainen kieli suurta taulukkolaskentaa vaativissa sovelluksissa. Useimmissa rekisteripohjaisissa tietokonearkkitehtuureissa ainakin yksiulotteiseen taulukkoon viittaminen tapahtuu hyvin nopeasti yhden konekäskyn aikana indeksoidun tiedonosoitusmoodin avulla. Näin ei siis ole Javassa.

JVM yksinkertainen esimerkki

Java lauseke x = z + y;

Tavukoodi

```
iload 2
iload 3
iadd
istore 4
```



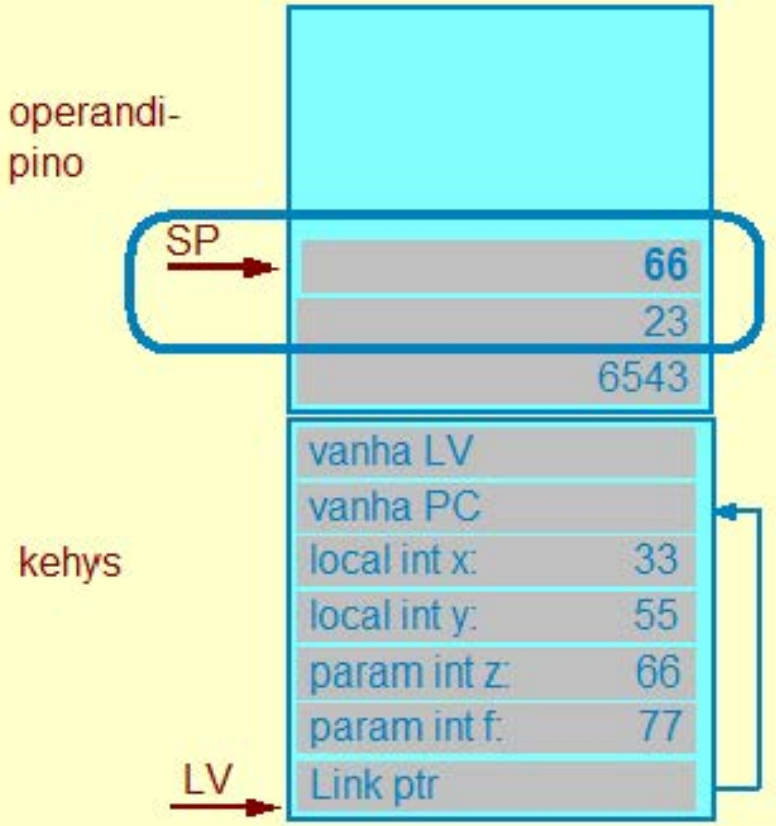
Tässä yksinkertaisessa JVM esimerkissä tarkastellaan kuinka Java-lauseke 'x = z + y' on käännetty tavukoodiksi ja kuinka tämä tavukoodin pätkä suoritetaan. Tässä oletetaan, että olemme jossakin metodissa, jolla on ollut kaksi kokonaislukuarvoista parametria: z ja f, ja jolla on kaksi paikallista kokonaislukuarvoista muuttujaa: x ja y. Metodin sisällä laskentaa on tehty jonkin verran ja välitulokset 23 ja 6543 ovat operandipinossa. Operandipinolle on itse asiassa tässä JVM toteutuksessa varattu tilaa kiinteä määrä, mutta se ei kaikki vielä ole käytössä. Pino-osoitin SP osoittaa käytössä olevan alueen huipulle.

JVM yksinkertainen esimerkki

Java lauseke x = z + y;

Tavukoodi

```
iload 2
iload 3
iadd
istore 4
```



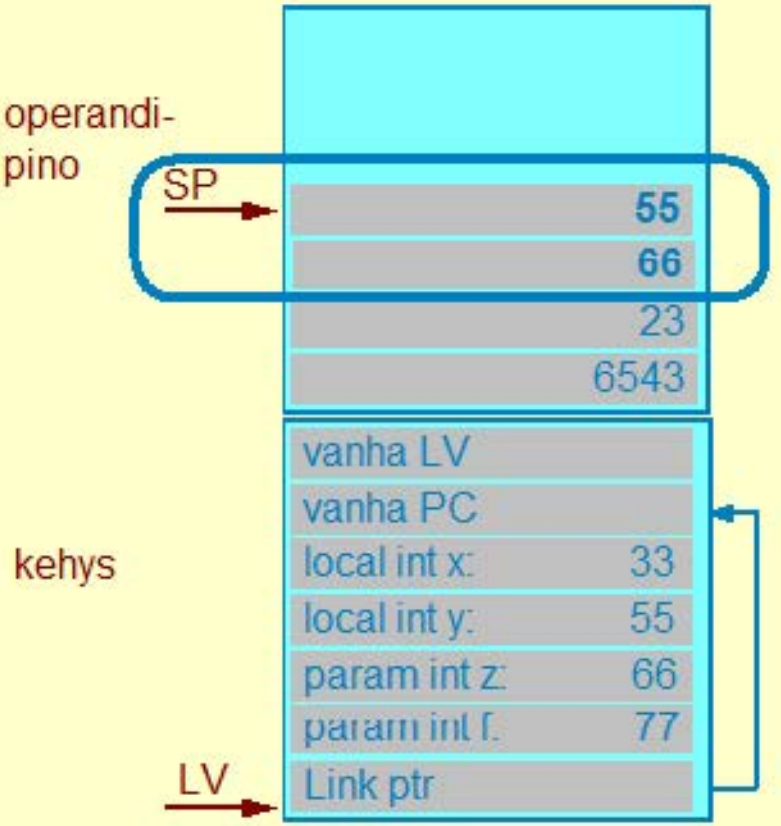
Yhteenlaskulauseke kääntyy neljäksi tavukoodin käskyksi. Ensin parametrin z arvo pitää ladata pinon huipulle.

JVM yksinkertainen esimerkki

Java lauseke x = z + y;

Tavukoodi

```
iload 2
iload 3
iadd
istore 4
```

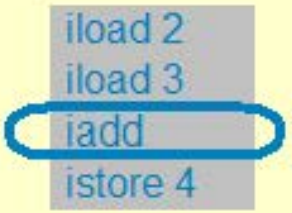


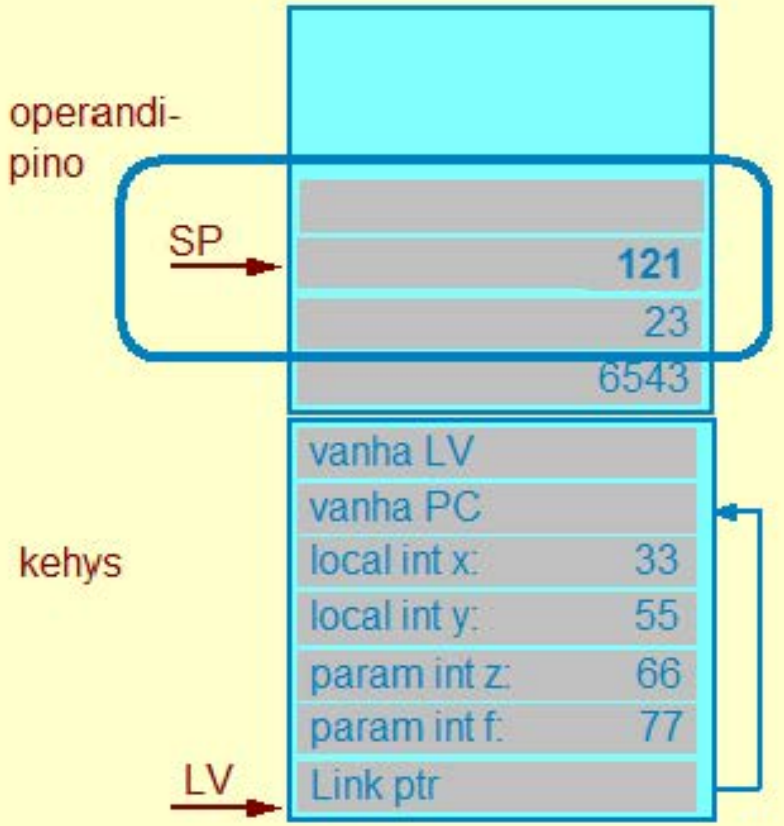
Paikallisen muuttujan y arvo kopioidaan vastaavasti pinon huipulle.

JVM yksinkertainen esimerkki

Java lauseke x = z + y;

Tavukoodi





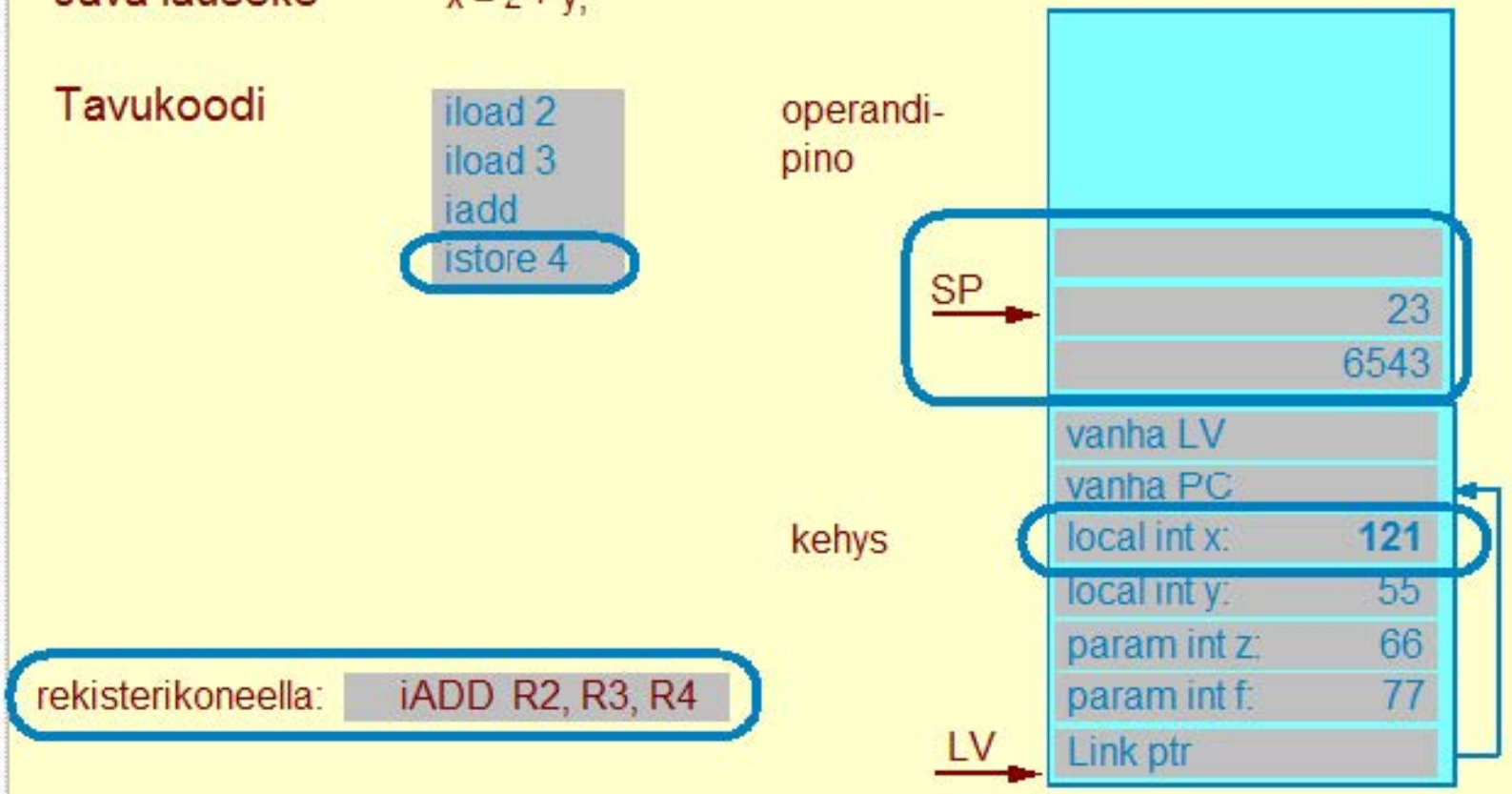
Yhteenlaskukäsky ottaa molemmat operandit pinon huipulta ALU:un ja sijoittaa yhteenlaskun tuloksen niiden paikalle pinon huipulle.

JVM yksinkertainen esimerkki

Java lauseke x = z + y;

Tavukoodi

```
iload 2
iload 3
iadd
istore 4
```



rekisterikoneella: iADD R2, R3, R4

Lopulta yhteenlaskun tulos voidaan kopioida pinon huipulta paikallisen muuttujan x arvoksi. Huomaatte varmaan, että tämänkin yksinkertaisen lausekkeen toteuttamisen tavukoodilla meni huimat 4 konekäskyä. Vielä huonompaa suorituskyvyn kannalta on, että kaikki data pidetään pinossa, mikä useimmissa JVM toteutuksissa tarkoittaa keskusmuistia. Useimmissa rekisteriarkkitehtuureissa parametrien ja paikallisten muuttujien arvot pidettäisiin luultavasti rekistereissä, joten koko lauseke voitaisiin ehkä toteuttaa yhdellä kolmea rekisteriä käyttävällä konekäskyllä.

JVM käskykanta

Peruslaskutoimitukset

Boolean

Pinon hallinta

Load/Store

Vertailut

Kontrollinsiirto

Muut

- muunnokset, taulukot, sekalaiset

typeADD
typeSUB
typeMUL
typeDIV
typeREM
typeNEG

type = i, l, f, d

iadd

fmul

JVM käskykantaan sisältyy tietenkin kaikki peruslaskutoimitukset. Jokaiselle neljälle perustietotyypille on kullekin omat käskynsä. Huomaa, että jakojäännös on määritelty myös liukuluvuille. Operadien täytyy olla oikeata tyyppiä, joten esimerkiksi dADD-käskylle ei voi antaa argumenttina pinossa tavallista liukulukua.

JVM käskykanta

Peruslaskutoimitukset

Boolean

Pinon hallinta

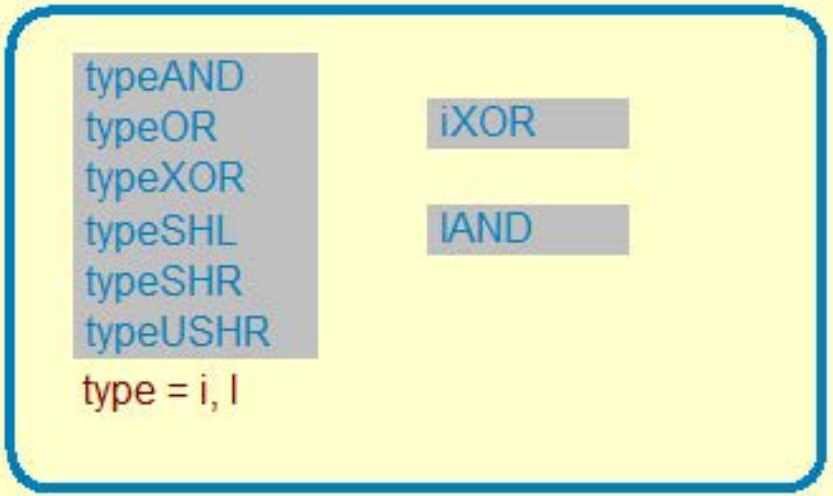
Load/Store

Vertailut

Kontrollinsiirto

Muut

- muunnokset, taulukot, sekalaiset



Boolean-operaatiot tehdään bittikohtaisesti 32 tai 64-bittisille alkioille. Oikealle siirrossa tavallinen SHR täyttää etumerkillä, kun taas USHR täyttää nollilla.

JVM käskykanta

Peruslaskutoimitukset

Boolean

Pinon hallinta

Load/Store

Vertailut

Kontrollinsiirto

Muut

- muunnokset, taulukot, sekalaiset

typeCONST const
BIPUSH con8
SIPUSH con16
LDC ind8
ACONST_NULL
type = i, l, f, d

iCONST 123456789

SIPUSH 2054

LDC 20

DUPxx
POP
POP2
SWAP

POP

DUP_X2

Pinon hallintaan on paljon käskyjä. Ensinnäkin pinoon voidaan tuoda arvoja usealla eri tavalla, joko suoraan tavukoodin koodialueelta tai paikallisista muuttujista. Pinon pinnalla olevia arvoja voidaan monistaa ja vaihdella eri tavoin. Esimerkiksi, jos samaa muuttujaa tarvitaan laskutoimituksessa monta kertaa, sen arvo on hyvä monistaa pinon pinnalle. Monistuksessa on 6 eri varianttia. Pinosta voidaan myös poistaa alkioita sijoittamatta niitä minnekään ja pinon päällimmäisten alkioiden arvoja voidaan vaihtaa keskenään. Tällaisten käskyjen olemassaolo viittaa siihen, että puhdas pinokone ei aina ole niin käytännöllinen.

JVM käskykanta

Peruslaskutoimitukset

Boolean

Pinon hallinta

Load/Store

Vertailut

Kontrollinsiirto

Muut

- muunnokset, taulukot, sekalaiset

typeLOAD ind8
typeALOAD
BALOAD
SALOAD
CALOAD
AALOAD

iLOAD 4

type = i, l, f, d

typeSTORE ind8
typeASTORE
BASTORE
SASTORE
CASTORE
AASTORE

fSTORE 6

Pinoon voidaan ladata paikallisten muuttujien arvoja tai taulukoiden osoitteita erilaisilla LOAD-käskyillä. Useimmat load-käskyt on kuitenkin varattu erilaisten taulukoiden alkioiden lataamiseen, yksi käsky kullekin taulukon alkion pituudelle. Vastaavaat talletuskäskyt ovat tietenkin myös olemassa.

JVM käskykanta

Peruslaskutoimitukset

Boolean

Pinon hallinta

Load/Store

Vertailut

Kontrollinsiirto

Muut

- muunnokset, taulukot, sekalaiset

```
IF_ICMPrel    OFFSET16
IF_ACMPEQ     OFFSET16
IF_ACMPNE     OFFSET16
IFrel         OFFSET16
IFNULL        OFFSET16
IFNONNULL     OFFSET16
LCMP
FCMPL
FCMPG
DCMPL
DCMPG
```

rel = eq, lt, le, ge, gt, ne, null, nonnull

```
ipush 4  ; x
ipush 5  ; y
ifeq  260
```


Erilaisia vertailuja varten pieni joukko käskyjä. Vertailun perusteella tehdään joko ehdollinen hyppy tai sitten tulos talletetaan pinoon jatkokäsittelyä varten. Liukuluvuille on omat vertailukäskynsä, mutta ei yhtä- tai erisuuruutta varten. Liukulukujen yhtäsuuruutta ei yleensä ole järkevää testata pyöristysvirheiden vuoksi.

JVM käskykanta

Peruslaskutoimitukset

Boolean

Pinon hallinta

Load/Store

Vertailut

Kontrollinsiirto

Muut

- muunnokset, taulukot, sekalaiset

```
INVCKEVIRTUAL  IND16
INVCKESTATIC   IND16
INVCKEINTERFACE ...
INVCKESPECIAL  IND16
JSR  OFFSET16
typeRETURN
RETURN
RET IND8
GOTO  OFFSET16
```

type = i, l, f, d, a

```
INVOKEVIRTUAL  2
```

```
iRETURN
```


Kontrollinsiirtoja varten on erilaisten metodien kutsut ja niistä paluukäskyt. Metodit voivat palauttaa arvonaan perustyypin, olion osoitteen tai ei mitään. Javan 'finally' lauseen toteutusta varten on JSR- ja RET-käskyt. Onhan mukana myös vanha kunnon GOTO-käskykin.

JVM käskykanta

Peruslaskutoimitukset

Boolean

Pinon hallinta

Load/Store

Vertailut

Kontrollinsiirto

Muut

- muunnokset, taulukot, sekalaiset

type2type
i2c
i2b

i2f

type = i, l, f, d

ANEWARRAY IND16
NEWARRAY atype
MULTIANEWARRAY IND IND
ARRAYLENGTH

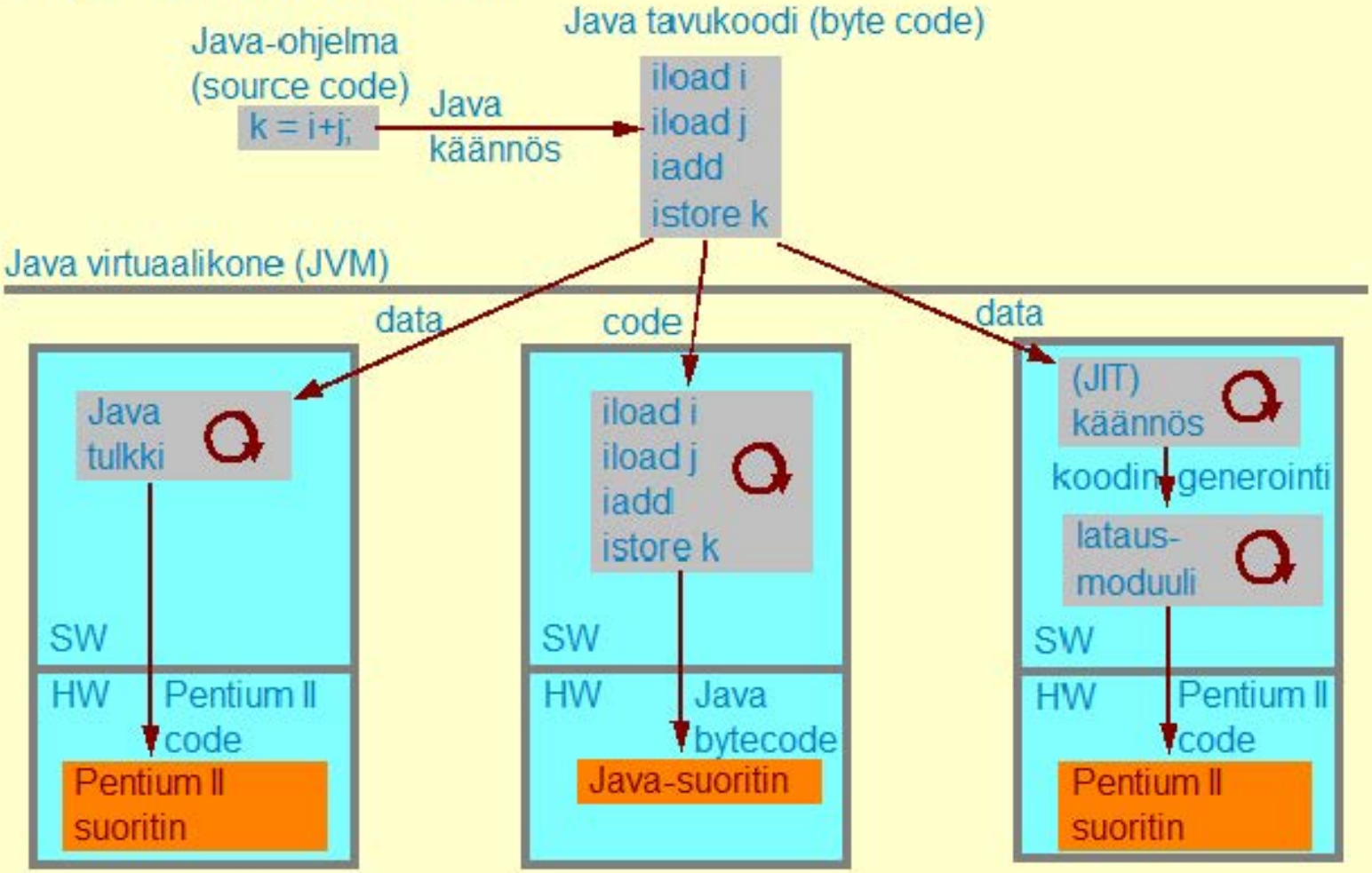
ANEWARRAY 24

atype = i, l, f, d, a

IINC IND8, CON8
NOP
NEW IND16
ATHROW
GETFIELD IND16
PUTFIELD IND16
TABLESWITCH
jne jne

Lisäksi mukana on vielä erinäinen joukko sälää. Käskyjen argumenttien tyyppien pitää aina olla oikein, joten tarvitaan jonkin verran tyypinmuunnoskäskyjä. Taulukoiden luomista ja käsittelyä varten tarvitaan omat käskynsä. IINC-käsky on kätevä taulukkoindeksien kasvattamiseen yhdellä. NOP-käsky on myös mukana. NEW-käskyllä luodaan olioita ja ATHROW-käskyllä saadaan aikaan keskeytys. Olion kenttiä manipuloidaan GETFIELD- ja PUTFIELD-käskyillä. TABLESWITCH- ja LOOKUPSWICTH-käskyillä toteutetaan nopea monivalinta. Loput käskyt ja niiden tarkemmat speksit löytyvät verkosta, jos haluatte niihin tarkemmin tutustua.

Java-ohjelmien suoritustavat



Kuten jo aikaisemmin mainitsimme, tavukoodisia Java-ohjelmia voidaan suorittaa usealla eri tavalla: tulkitsemalla, kääntämällä tai erityisesti Javaa varten kehitetyllä Java-suorittimella. Näitä kaikkia suoritustapoja yhdistää se, että niissä kaikissa on joko laitteisto- tai ohjelmistotasolla toteutettu tavukoodia suorittava JVM. Esittelemme seuraavaksi kaikki nämä JVM eri toteutustavat eli Java-ohjelmien eri suoritustavat vähän yksityiskohtaisemmin.

Java-tulkki

Luokkamäärittely Javan luokkatiedostossa (*class file*)

- otsake, vakiot, oikeudet, liittymät, kentät, metodit, attribuutit

Emuloi JVM konekielen (tavukoodi, byte code) käskyjä

- yksi tavukoodin käsky kerrallaan

JVM rekisterit ja muistialueet emuloitu tulkin tietorakenteina muistissa

- vrt. Titokone ja ttk-91 -arkkitehtuurin rekisterit ja muisti

Hidasta, mutta joustavaa

- kaikki JVM rakenteet (rekisterit, pino, metodialue, jne) sijaitsevat muistissa
- emulointi vain tavukoodin käsky kerrallaan, ei mahdollisuutta usean tavukoodisen käskyn samanaikaiseen suoritukseen
- tarvittaessa viitattu luokka voidaan ladata verkosta suoritusaikana

Olemme hieman yksinkertaistaneet: kääntäjän tuottaman luokkatiedoston esitysmuotoa ottamalla mukaan vain suoritusaikana käytetyn tavukoodin. Todellisuudessa luokkatiedostossa on mukana kaikki luokan määrittelyssä mukana olevat tiedot sopivasti paketoidussa muodossa. Luokkatiedoston avulla JVM voidaan sitten alustaa luokan metodin suoritusta varten.

Java-tulkki

Luokkamäärittely Javan luokkatiedostossa (*class file*)

- otsake, vakiot, oikeudet, liittymät, kentät, metodit, attribuutit

Emuloi JVM konekielen (tavukoodi, byte code) käskyjä

- yksi tavukoodin käsky kerrallaan

JVM rekisterit ja muistialueet emuloitu tulkin tietorakenteina muistissa

- vrt. Titokone ja ttk-91 -arkkitehtuurin rekisterit ja muisti

Hidasta, mutta joustavaa

- kaikki JVM rakenteet (rekisterit, pino, metodialue, jne) sijaitsevat muistissa
- emulointi vain tavukoodin käsky kerrallaan, ei mahdollisuutta usean tavukoodisen käskyn samanaikaiseen suoritukseen
- tarvittaessa viitattu luokka voidaan ladata verkosta suoritusaikana

Java-tulkki suorittaa metodien tavukoodisia käskyjä yksi käsky kerrallaan, hyvin samalla tavalla kuin esimerkkikoneemme ttk-91 Titokone-simulaattori suorittaa ttk-91 -konekielellä esitettyjä ohjelmia. Java-tulkki eli siis JVM on nyt tavallinen käyttöjärjestelmän tunnistama sovellus, joka lukee tavukoodisia konekäskyjä datara yksi käsky kerrallaan ja suorittaa niitä tavanomaisen tapaan JVM käskyjen suoritussyklin mukaisesti.

Java-tulkki

Luokkamäärittely Javan luokkatiedostossa (*class file*)

- otsake, vakiot, oikeudet, liittymät, kentät, metodit, attribuutit

Emuloi JVM konekielen (tavukoodi, byte code) käskyjä

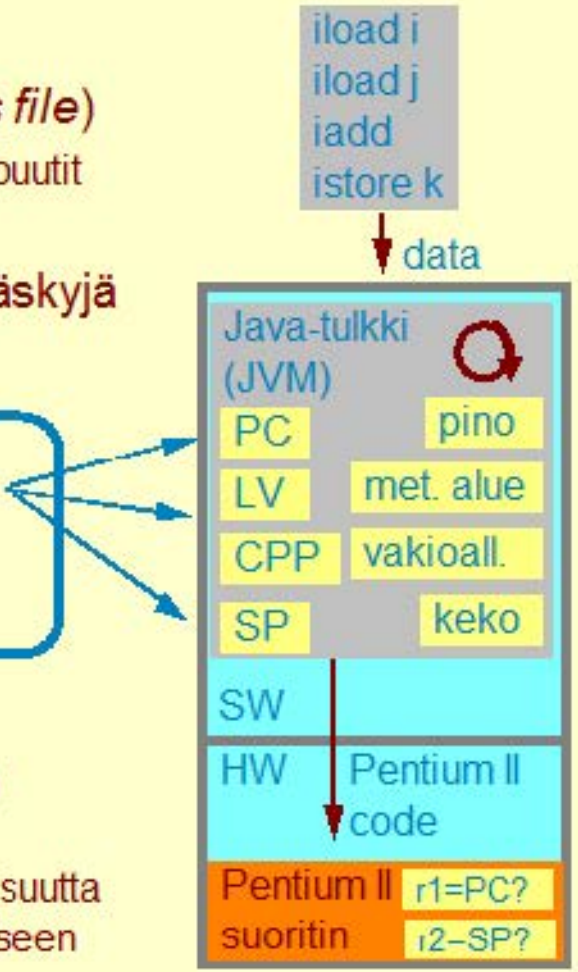
- yksi tavukoodin käsky kerrallaan

JVM rekisterit ja muistialueet emuloitu tulkin tietorakenteina muistissa

- vrt. Titokone ja ttk-91 -arkkitehtuurin rekisterit ja muisti

Hidasta, mutta joustavaa

- kaikki JVM rakenteet (rekisterit, pino, metodialue, jne) sijaitsevat muistissa
- emulointi vain tavukoodin käsky kerrallaan, ei mahdollisuutta usean tavukoodisen käskyn samanaikaiseen suoritukseen
- tarvittaessa viitattu luokka voidaan ladata verkosta suoritusaikana



JVM toteuttava tulkki sisältää omina tietorakenteinaan kaikki JVM:n rekisterit ja muistialueet. Kunkin tavukoodisen käskyn toteutus sitten manipuloi näitä muistissa olevia tietorakenteita ja sillä tavoin emuloi hypoteettisen JVM:n toimintaa tällä tavoin simuloidussa laitteistossa. Emulointiin sisältyy tietenkin huomattava määrä muistiviitteitä, joita voi kuitenkin hieman vähentää varaamalla esimerkiksi muutama laiterekisteri JVM rekistereiksi.

Java-tulkki

Luokkamäärittely Javan luokkatiedostossa (*class file*)

- otsake, vakiot, oikeudet, liittymät, kentät, metodit, attribuutit

Emuloi JVM konekielen (tavukoodi, byte code) käskyjä

- yksi tavukoodin käsky kerrallaan

JVM rekisterit ja muistialueet emuloitu tulkin tietorakenteina muistissa

- vrt. Titokone ja ttk-91 -arkkitehtuurin rekisterit ja muisti

Hidasta, mutta joustavaa

- kaikki JVM rakenteet (rekisterit, pino, metodialue, jne) sijaitsevat muistissa
- emulointi vain tavukoodin käsky kerrallaan, ei mahdollisuutta usean tavukoodisen käskyn samanaikaiseen suoritukseen
- tarvittaessa viitattu luokka voidaan ladata verkosta suoritusaikana

JVM ohjelmistototeutus tulkin avulla on itse asiassa hyvin yksinkertaista. Huonona puolena sillä on kuitenkin suorituksen hitaus, koska (a) lähes kaikki tieto pidetään koko ajan muistissa eikä laiterekistereissä ja (b) mitään normaalin suorittimen suoritusnopeutta lisääviä optimointeja ei voi tehdä, koska tavukoodin käskyt täytyy emuloida alusta loppuun yksi kerrallaan. Järjestelmä on kuitenkin hyvin joustava, koska minkä tahansa toiminnallisuuden voi helposti toteuttaa ohjelmistolla. Esimerkiksi, suoritusaikana voidaan viitata muualla sijaitseviin luokkamäärittelyihin ja nuo luokat voidaan sitten ladata verkon kautta tarvittaessa.

Käännös natiiviympäristö konekielelle

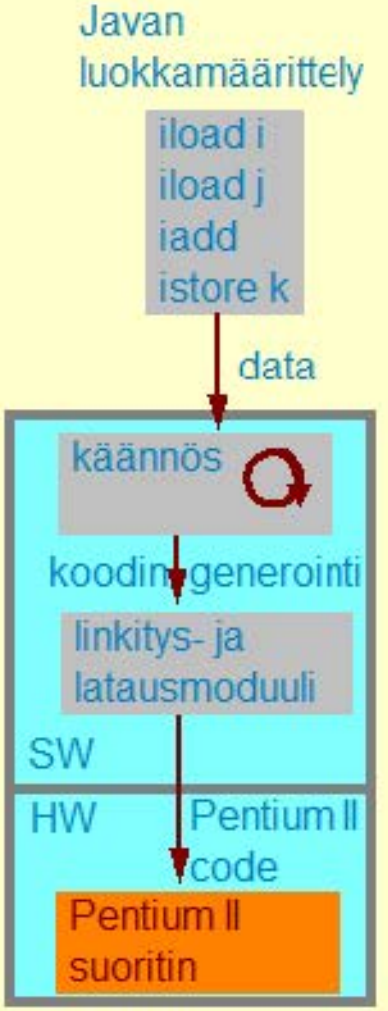
Tavukoodi käännetään ennen suoritusta natiiviympäristön suorittimen omalle konekielelle, linkitetään ja ladataan normaalia suoritusta varten

Käännös voidaan toteuttaa korkean tason kielen (esim. *C*) kautta

- käännä tavukoodinen esitystapa C:llä esitetyksi
  - helpompi toteuttaa kuin koko kääntäjä yllä
- käytä tavallista, jo olemassaolevaa C-kääntäjää kääntämään C-kielellä esitetty ohjelma konekieliseen muotoon

Ongelma: dynaamisen linkityksen tarve

- Java on suunniteltu tilanteisiin, joissa dynaaminen linkitys on normaali tapahtuma



Suoritusnopeuden puolesta paras tapa suorittaa tavukoodia esimerkiksi Pentium-II ympäristössä on kääntää tavukoodi etukäteen suoraan Pentium-II konekielelle. Käännös tehdään samalla tavalla kuin minkä tahansa muunkin ohjelmointikielen kääntäminen. Käännöksen jälkeen kaikki ohjelman tarvitsemien luokkien linkitysmoduulit linkitetään yhteen, joten lopulta saadaan ihan tavallinen ladattava objektimoduuli. Tässä muodossa ohjelman suoritus on hyvin nopeata, koska Pentium-II suoritin osaa suorittaa montaa peräkkäistä konekäskyä limittäin samanaikaisesti.

Käännös natiiviympäristö konekielelle

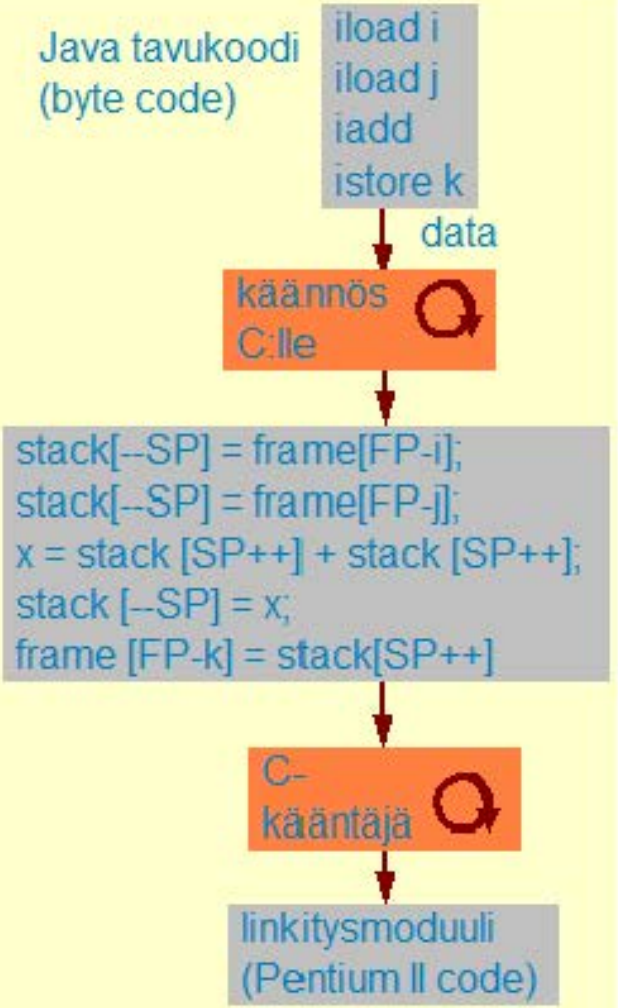
Tavukoodi käännetään ennen suoritusta natiiviympäristön suorittimen omalle konekielelle, linkitetään ja ladataan normaalia suoritusta varten

Käännös voidaan toteuttaa korkean tason kielen (esim. C) kautta

- käännä tavukoodinen esitystapa C:llä esitetyksi
  - helpompi toteuttaa kuin koko kääntäjä yllä
- käytä tavallista, jo olemassaolevaa C-kääntäjää kääntämään C-kielellä esitetty ohjelma konekieliseen muotoon

Ongelma: dynaamisen linkityksen tarve

- Java on suunniteltu tilanteisiin, joissa dynaaminen linkitys on normaali tapahtuma



Tavukoodin kääntäminen natiiviympäristön konekielelle voidaan myös tehdä vähän helpommin korkean tason kielen kautta. Tavukoodi voidaan kääntää ensin C:lle ja sitten ihan tavallisen C-kääntäjän avulla natiiviympäristön konekielelle. Etuna tässä lähestymistavassa on se, että C-kääntäjä on jo valmiiksi toteutettu ja se sisältää jo valmiiksi suoritusnopeuden kannalta optimoidun koodin generointipalikat.

Käännös natiiviympäristö konekielelle

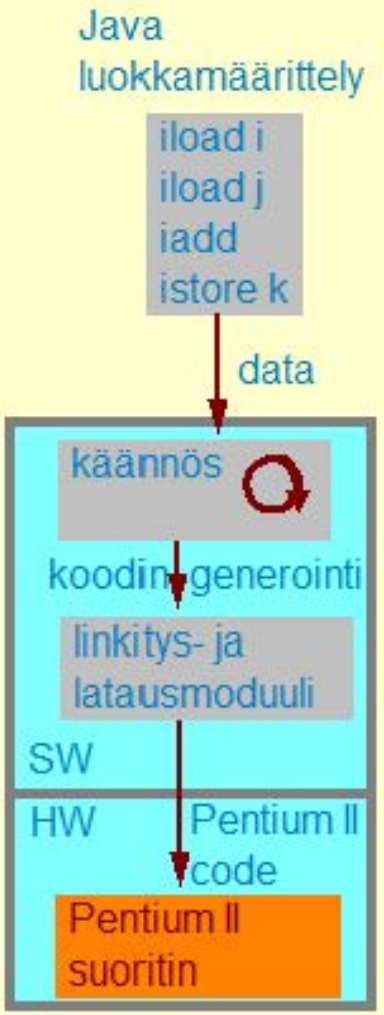
Tavukoodi käännetään ennen suoritusta natiiviympäristön suorittimen omalle konekielelle, linkitetään ja ladataan normaalia suoritusta varten

Käännös voidaan toteuttaa korkean tason kielen (esim. *C*) kautta

- käännä tavukoodinen esitystapa C:llä esitetyksi
  - helpompi toteuttaa kuin koko kääntäjä yllä
- käytä tavallista, jo olemassaolevaa C-kääntäjää kääntämään C-kielellä esitetty ohjelma konekieliseen muotoon

Ongelma: dynaamisen linkityksen tarve

- Java on suunniteltu tilanteisiin, joissa dynaaminen linkitys on normaali tapahtuma



Ongelmana etukäteen kääntämisessä ja linkittämisessä on juuri se, että käännös ja linkitys pitää tehdä etukäteen täydellisesti kaikille ohjelman mahdollisesti käyttämille luokille. Usein on niin, että yhdellä ohjelman suorituskerralla vain osaa siinä viitatuista luokista todellisuudessa tarvitaan. Etukäteen käännetty ja linkitetty moduuli on siten usein tarpeettoman suuri. Voipa jopa olla, että on täysin epäkäytännöllistä linkittää moduuliin kaikki mahdolliset sen viittaamat luokat. Esimerkiksi, on turha linkittää luokkaan ranskalaiset lokalitettipiirteet, koska niitä tarvitaan vain ranskankielisellä kielialueella.

Just In Time -käännös (JIT-käännös)

Käännös ja linkitys tapahtuu vasta kun luokkaan viitataan ensimmäisen kerran

Luokka käännetään natiivikonekielelle dynaamisesti linkitettäväksi moduuliksi

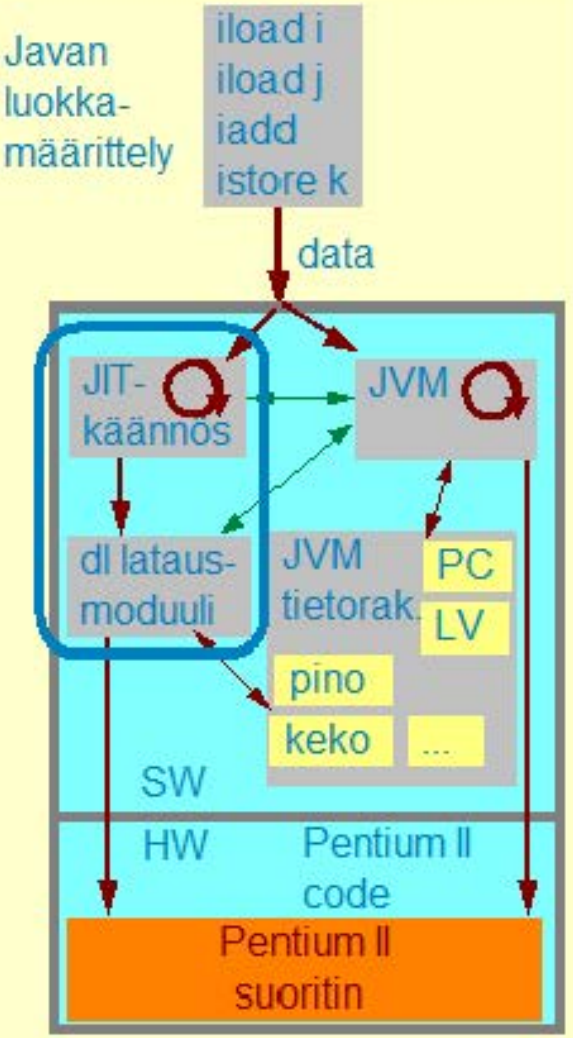
Tarvitsee paljon muistia

- optimoiva kääntäjä on muistisyöppö

Voi hidastaa suoritusta, jos käännökseen menee enemmän aikaa kuin emulointiin

- optimoiva kääntäjä on suoritusaikasyöppö
- emuloi ensin ja käännös vasta 2. metodin kutsukerralla?

JVM rekisterit ja muistialueet emuloitu tulkin ja käännetyn natiivikoodin yhteisinä tietorakenteina



JIT-käännös välttää liian suurten objektimoduulien ongelman kääntämällä ja linkittämällä uudet luokat vasta viittaushetkellä paikalleen. Tähän tietenkin voi kulua jonkin verran aikaa - varsinkin, jos viitattu luokka pitää hakea ensin jostain verkosta. Luokka käännetään dynaamisesti linkitettäväksi moduuliksi natiivikonekielelle, joten sen suoritus on sitten nopeata.

Just In Time -käännös (JIT-käännös)

Käännös ja linkitys tapahtuu vasta kun luokkaan viitataan ensimmäisen kerran

Luokka käännetään natiivikonekielelle dynaamisesti linkitettäväksi moduuliksi

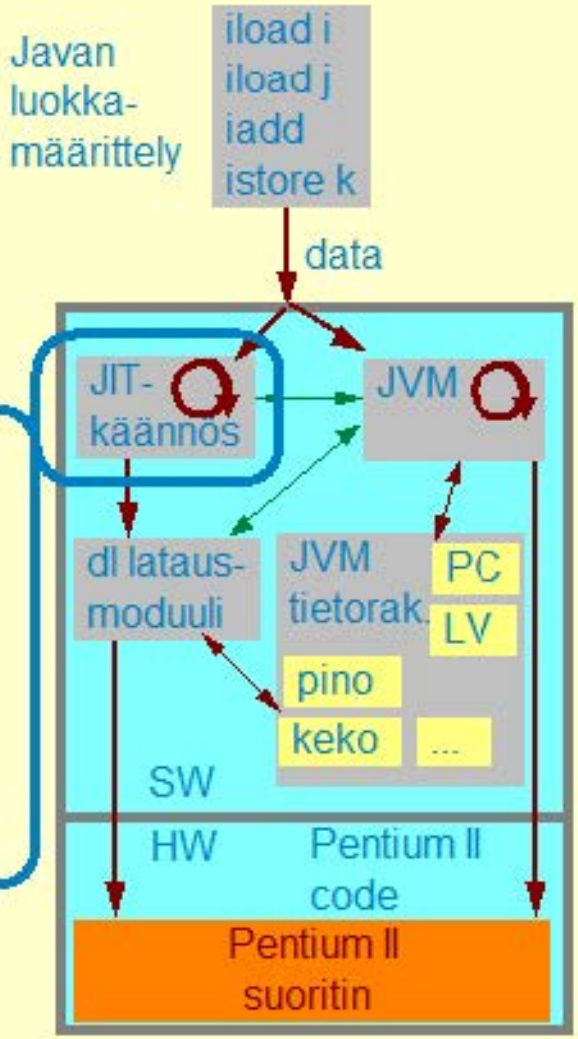
Tarvitsee paljon muistia

- optimoiva kääntäjä on muistisyöppö

Voi hidastaa suoritusta, jos käännökseen menee enemmän aikaa kuin emulointiin

- optimoiva kääntäjä on suoritusaikasyöppö
- emuloi ensin ja käännös vasta 2. metodin kutsukerralla?

JVM rekisterit ja muistialueet emuloitu tulkin ja käännetyn natiivikoodin yhteisinä tietorakenteina



Huonona puolena JIT-käännöksessä on kääntäjän läsnäolo suoritusaikana. Kääntäjä tarvitsee paljon muistitilaa ja varsinkin optimoidun koodin generointi voi viedä paljon aikaa. Jos juuri suurella vaivalla käännetystä ja linkitetystä luokasta suoritetaan vain jokin lyhyt metodi, niin tulkitseminen olisi voinut olla viisaampi ratkaisu. JIT-kääntäjissä onkin usein erilaisia heuristisia ratkaisuja kääntämisajankohdan ja generoitavan koodin optimointitason päättämiseen.

Just In Time -käännös (JIT-käännös)

Käännös ja linkitys tapahtuu vasta kun luokkaan viitataan ensimmäisen kerran

Luokka käännetään natiivikonekielelle dynaamisesti linkitettäväksi moduuliksi

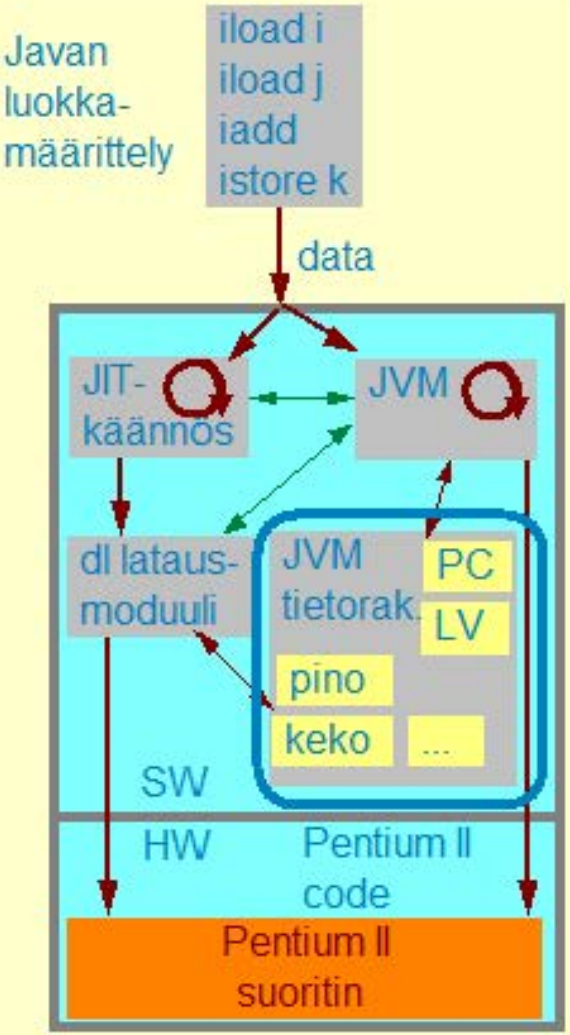
Tarvitsee paljon muistia

- optimoiva kääntäjä on muistisyöppö

Voi hidastaa suoritusta, jos käännökseen menee enemmän aikaa kuin emulointiin

- optimoiva kääntäjä on suoritusaikasyöppö
- emuloi ensin ja käännös vasta 2. metodin kutsukerralla?

JVM rekisterit ja muistialueet emuloitu tulkin ja käännetyn natiivikoodin yhteisinä tietorakenteina



Tavukoodin suoritusta ja JIT-kääntämistä valvoo JVM-sovellus, jonka määrittelemät JVM rekisterit ja muistialueet ovat myös JIT-käännettyjen moduulien viitattavissa. Tällä tavoin samoja emuloitavia tietorakenteita voidaan tehokkaasti käyttää sekä käännetyistä luokista että JVM-tulkista.

Java-suoritin

Laitteistototeutus JVM:sta

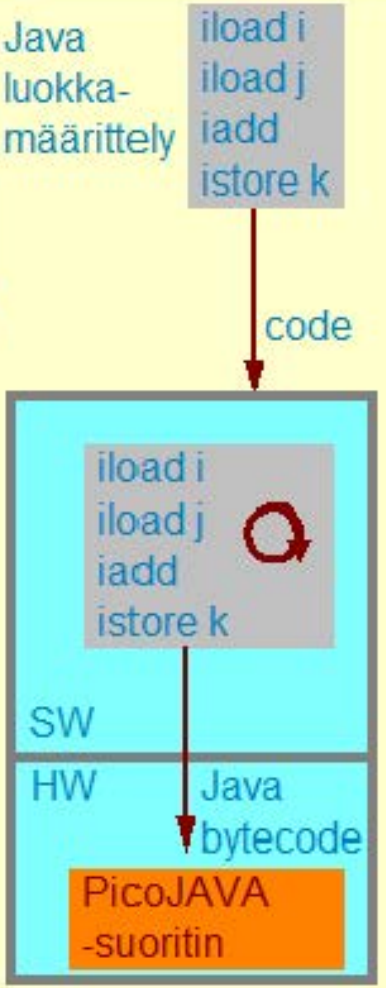
- koneessa voidaan suoraan suorittaa Java-luokkia ja tavukoodia
- laitteistotason toteutus tavukoodin käskyille

Sun PicoJAVA II

- suorittimen määrittely - ei laitteistototeutus
- määrittelyn mukaan rakennettu laitteisto sopii hyvin Javan luokkamäärittelyjen (Java Class File) suorittamiseen
- tarvitaan myös käyttöjärjestelmä!

Sun PicoJAVA II rakenne

- kaikki 226 JVM konekäskyä omina konekäskyinä
- 115 muuta konekäskyä käyttöjärjestelmän ja muiden ohjelmointikielien (lue *C*) toteuttamiseksi
- valinnainen välimuisti ja liukulukusuoritin



Java-suoritin on laitteisto, jossa on suoraan tavukoodia konekielenään ymmärtävä suoritin. Luokkamäärittelyssä olevat metodien koodit voidaan siis suoraan suorittaa laitteistolla muiden konekielien tapaan. Suoritin voidaan nyt suunnitella kokonaan tavukoodin suorittamista varten ja on myös mahdollista optimoida koodin suoritusta siten, että usea perkkäinen tavukoodin käsky on samanaikaisesti suorituksessa. Vähän huonona puolena tässä on, että tavukoodin suorittamista varten optimoitu järjestelmä ei välttämättä ole oikein hyvä muilla korkean tason kielillä kuvattujen ohjelmien suorittamiseen.

Java-suoritin

Laitteistototeutus JVM:sta

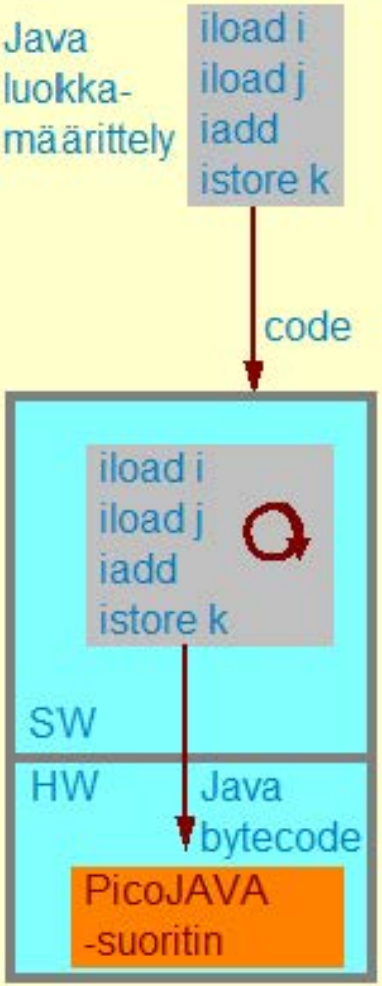
- koneessa voidaan suoraan suorittaa Java-luokkia ja tavukoodia
- laitteistotason toteutus tavukoodin käskyille

Sun PicoJAVA II

- suorittimen määrittely - ei laitteistototeutus
- määrittelyn mukaan rakennettu laitteisto sopii hyvin Javan luokkamäärittelyjen (Java Class File) suorittamiseen
- tarvitaan myös käyttöjärjestelmä!

Sun PicoJAVA II rakenne

- kaikki 226 JVM konekäskyä omina konekäskyinä
- 115 muuta konekäskyä käyttöjärjestelmän ja muiden ohjelmointikielien (lue *C*) toteuttamiseksi
- valinnainen välimuisti ja liukulukusuoritin



Java on Sun Microsystems'in kehittämä ja omistama tuote. Ei siis ole hämmästyttävää, että juuri Sunilta tulee JVM laitteistotason toteutusehdotus. PicoJAVA II on Sun'in määrittely JVM:n toteuttamiseksi. Sen tarkoituksena on yrittää standardoida JVM laitteistototeutuksia siten, että eri valmistajien laitteet olisivat mahdollisimman yhteensopivia. PicoJAVA II on siis vasta paperilla oleva määrittely. Ideana on, että kuka tahansa valmistaja voisi sitten tämän määrittelyn perusteella toteuttaa varsinaisen lastulla olevan suorittimen. Lisäksi suoritinta hyödyntäviä järjestelmiä voidaan suunnitella, vaikka itse Java-suoritinta ei vielä olisikaan.

Java-suoritin

Laitteistototeutus JVM:sta

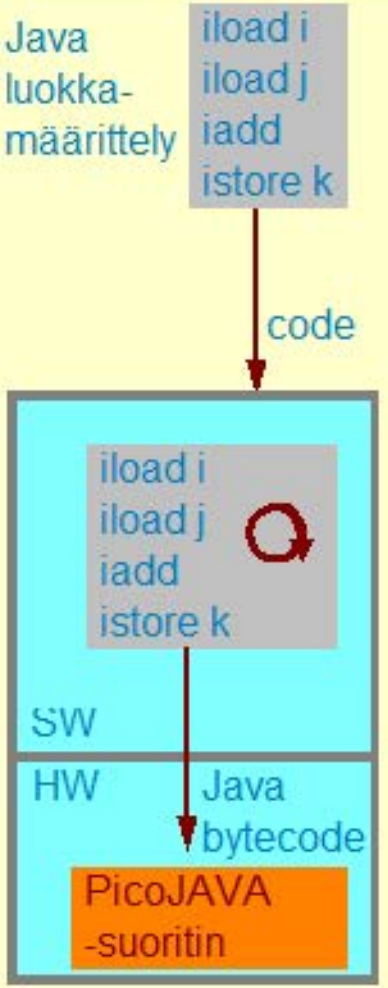
- koneessa voidaan suoraan suorittaa Java-luokkia ja tavukoodia
- laitteistotason toteutus tavukoodin käskyille

Sun PicoJAVA II

- suorittimen määrittely - ei laitteistototeutus
- määrittelyn mukaan rakennettu laitteisto sopii hyvin Javan luokkamäärittelyjen (Java Class File) suorittamiseen
- tarvitaan myös käyttöjärjestelmä!

Sun PicoJAVA II rakenne

- kaikki 226 JVM konekäskyä omina konekäskyinä
- 115 muuta konekäskyä käyttöjärjestelmän ja muiden ohjelmointikielien (lue *C*) toteuttamiseksi
- valinnainen välimuisti ja liukulukusuoritin



PicoJAVA II toteuttaa kaikki Javan tavukoodin 226 konekäskyä. Siihen sisältyy kuitenkin myös 115 muuta konekielen käskyä, joiden avulla järjestelmälle voidaan kääntää tehokkasti toimiva C:llä kirjoitettu käyttöjärjestelmä. On mukavaa käyttää jo olemassa olevia käyttöjärjestelmän osia uuteen järjestelmään, mutta C:llä kirjoitettujen moduulien kääntäminen tehokkaaksi tavukoodiksi oli selvästikin liian vaikeata. JVM on hyvä ideaalikone, mutta tehoton käytänrön tarpeisiin. Määrittelyssä on mukana myös välimuisti ja liukulukusuoritin valinnaisina komponentteina. Välimuistia ei siis tarvitse ottaa mukaan toteutukseen, jos suoritusnopeus ei ole ongelma.

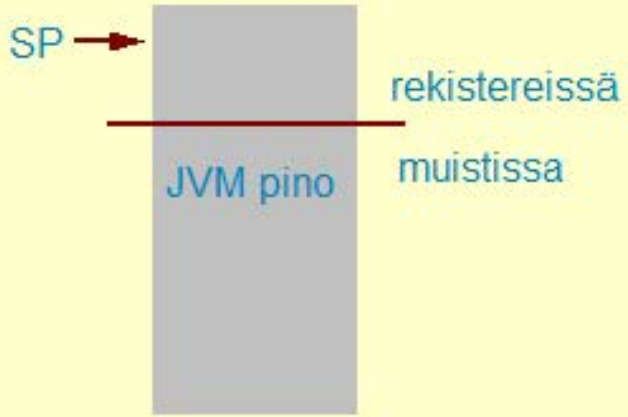
PicoJAVA II ja JVM pino

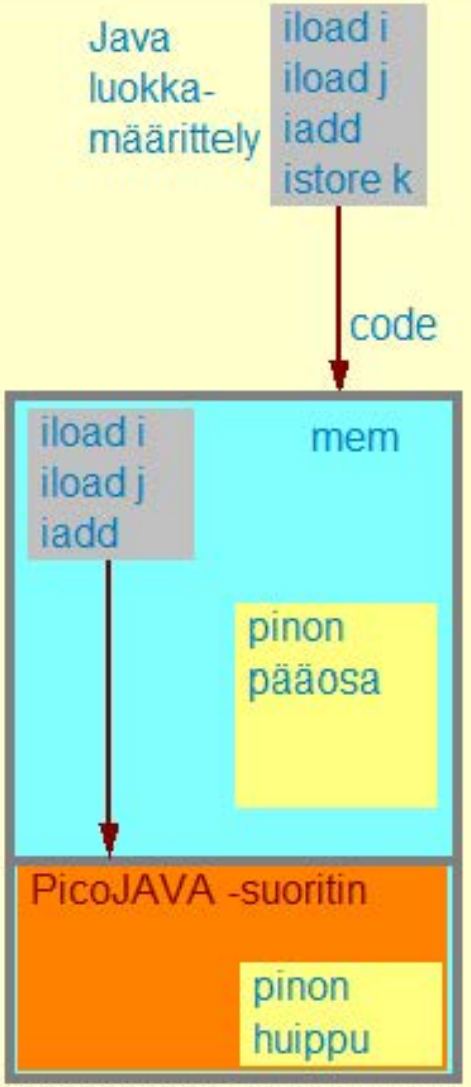
Muistissa oleva pino hidastaa laskentaa

- kaikki välitulokset pinossa
- jokainen aritmetiikkaoperaatio poistaa operandit pinosta

PicoJAVA II:ssa pinon huippu pidetään erikoisrekistereissä (pinovälimuisti, stack cache)

- huipulla olevat 64 sanaa ovat suorittimella



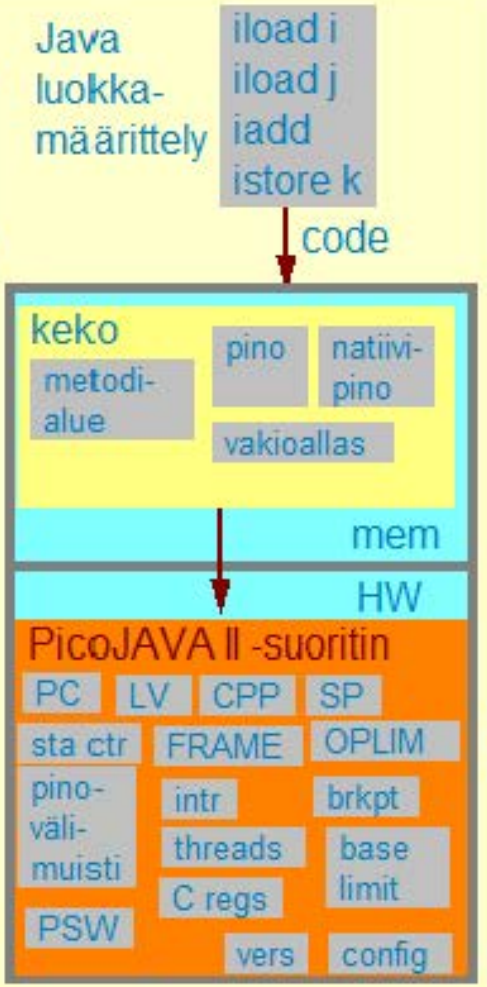


JVM:n tulkituissa ja käännetyissä toteutuksissa suurena suorituskyvyn hidasteena on kaikkien välitulosten pitäminen muistissa pinossa sen sijaan, että niitä pidettäisiin tehokkaasti optimoiduissa rekistereissä suorittimella. PicoJAVA II:ssa yritetään päästä tähän samaan tulokseen toteuttamalla JVM:n pino osittain laitteistolla. Suorittimella on 64 sanan erikoisrekisterialue, jossa pyritään pitämään JVM:n pinon päällimäiset alkiot. Useimmat pinoviitteet kohdistuvat nyt toivon mukaan näihin erikoisrekistereihin. Aika monimutkaisen tuntuinen toteutus tavallisiin viitattaviin rekistereihin verrattuna.

PicoJAVA II rekisterit

25 rekisteriä á 32 bittiä

PC, LV, CPP, SP (pino kasvaa pienempiin osoitteisiin)
OPLIM - alaraja SP:lle, pinon ylivuotokeskeytys
FRAME - nykymetodin paluuosoite
PSW - tilarekisteri
pinovälimuistin hallintarekisteri - mitkä pinon alkiot ovat pinovälimuistissa?
keskeytysten ja breakpoint'ien hallintarekisterit (4 kpl)
säikeiden hallintarekisterit (4 kpl)
roskien keruun hallintarekisteri
C ja C++ ohjelmien apurekisterit (4 kpl)
rajarekisterit etuoikeutetulle muistialueelle
suorittimen version numeron ja konfiguraation ilmaisevat rekisterit

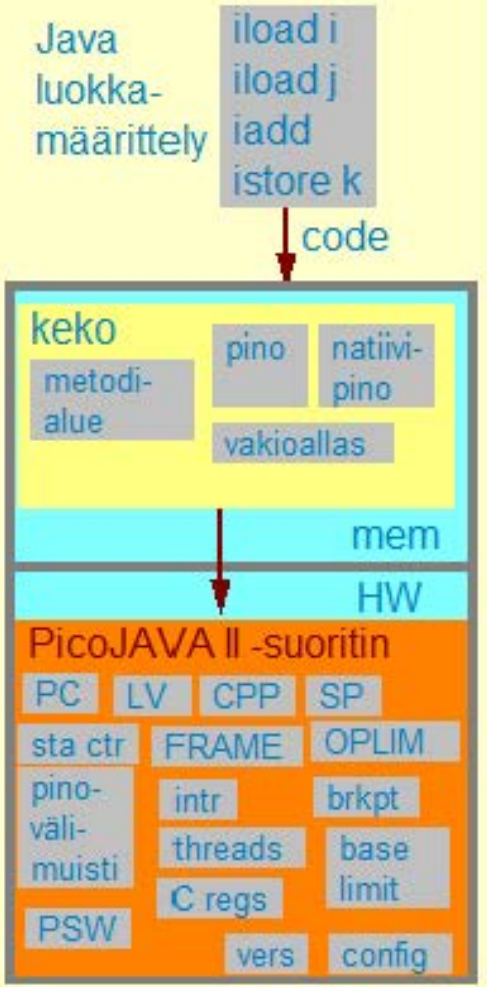


PicoJAVA II:ssa on yhteensä 25 32-bittistä rekisteriä. Tottakai kaikki JVM:n neljä rekisteriä on toteutettu laitteistolla. Mielenkiintoista käytännön toteutuksen kannalta on tietenkin, että tarvitaan 21 muuta rekisteriä hyvin toimivan laitteiston rakentamiseen.

PicoJAVA II rekisterit

25 rekisteriä á 32 bittiä

PC, LV, CPP, SP (pino kasvaa pienempiin osoitteisiin)
OPLIM - alaraja SP:lle, pinon ylivuotokeskeytys
FRAME - nykymetodin paluuosoite
PSW - tilarekisteri
pinovälimuistin hallintarekisteri - mitkä pinon alkiot ovat pinovälimuistissa?
keskeytysten ja breakpoint'ien hallintarekisterit (4 kpl)
säikeiden hallintarekisterit (4 kpl)
roskien keruun hallintarekisteri
C ja C++ ohjelmien apurekisterit (4 kpl)
rajarekisterit etuoikeutetulle muistialueelle
suorittimen version numeron ja konfiguraation ilmaisevat rekisterit



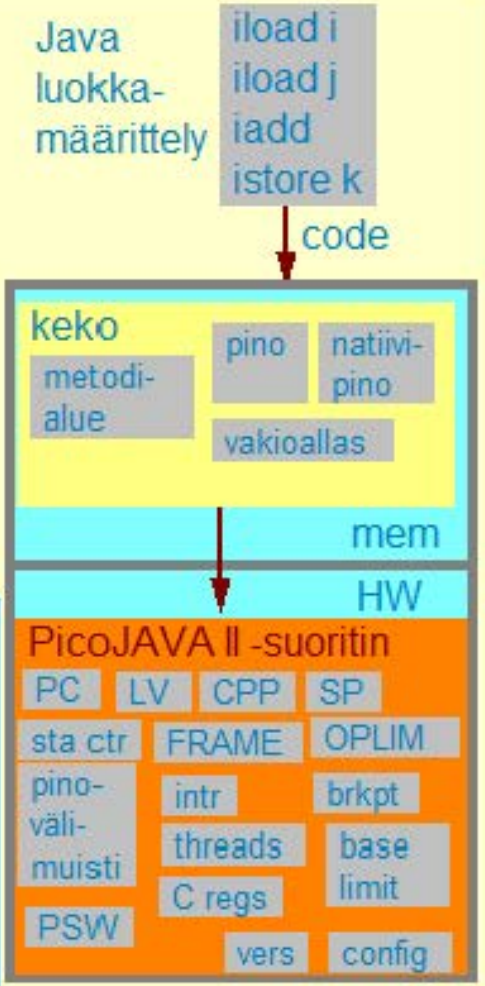
PicoJAVA II:ssa on oma rekisteri pinon ylivuototarkistusta varten. Paluuosoite löytyy nopeasti oman rekisterinkautta ilman, että sitä täytyy hakea pinosta. Suorittimen hallintaan tarvitaan tietenkin tilarekisteri PSW. Pinovälimuistin kontrollointiin tarvitaan oma apurekisteri, joka kertoo nopeasti, mitkä pinon huipulla olevat alkiot ovat tällä hetkellä nopeasti saatavilla pinovälimuistissa suorittimella.

PicoJAVA II rekisterit

25 rekisteriä á 32 bittiä

PC, LV, CPP, SP (pino kasvaa pienempiin osoitteisiin)
OPLIM - alaraja SP:lle, pinon ylivuotokeskeytys
FRAME - nykymetodin paluuosoite
PSW - tilarekisteri
pinovälimuistin hallintarekisteri - mitkä pinon alkiot ovat pinovälimuistissa?

keskeytysten ja breakpoint'ien hallintarekisterit (4 kpl)
säikeiden hallintarekisterit (4 kpl)
roskien keruun hallintarekisteri
C ja C++ ohjelmien apurekisterit (4 kpl)
rajarekisterit etuoikeutetulle muistialueelle
suorittimen version numeron ja konfiguraation ilmaisevat rekisterit



Tavallisten keskeytysten ja ohjelmallisesti määriteltävissä olevien breakpoint'ien toteutukseen tarvitaan neljä rekisteriä, samoin kuin JVM säikeiden toteutukseen. Roskien keruun hallinnointiin on oma rekisterinsä ja C ja C++ -ohjelmia varten on neljä yleiskäyttöistä rekisteriä. Käyttöjärjestelmän etuoikeutettu muistialue on suojattu omilla rekistereillään. Lisäksi rekistereistä löytyy suorittimen versionumero ja konfiguraatiotietoja, joiden avulla esimerkiksi nähdään koko ajan, onko suorittimella välimuistia vai ei.

PicoJAVA II ylimääräisiä käskyjä

Read/write ylimääräisille rekistereille

Osoittimien manipulointikäskyt

- tarvitaan C/C++ ohjelmissa

C/C++ aliohjelmien kutsu/paluu

Natiivi HW manipulointi

- clear cache

Muut käskyt

- power off

JVM:n omien 226 tavukoodisen käskyn lisäksi PicoJAVA II:ssa on siis 115 muuta konekäskyä ylimääräisten rekistereiden ja muun järjestelmän hallintaan. C ja C++ kieliset ohjelmat tarvitsevat tukea pointereiden hallintaan sekä natiivimetodien eli C/C++ aliohjelmien kutsun ja paluun toteuttamiseen natiivimetodien pinon kautta. Välimuistin hallintaan on useita konekäskyjä samoir kuin pienlaitteissa virrankäytön kontrolloimiseen.

PicoJAVA II ylimääräisiä käskyjä

Read/write ylimääräisille rekistereille

Osoittimien manipulointikäskyt

- tarvitaan C/C++ ohjelmissa

C/C++ aliohjelmien kutsu/paluu

Natiivi HW manipulointi

- clear cache

Muut käskyt

- power off

Sun. picoJava-II Programmer's Reference Manual. Sun Microsystems, March 1999.

Jos haluat lisätietoja PicoJava II suoritinmäärittelystä, niin tutkaile verkosta. Sunin reference manual löytyy esimerkiksi tämän linkin takaa.

PicoJAVA II toteutuksia

Sun microJAVA 701 (SUN Microsystems)

- valinnainen välimuisti, oma muistiväylä, PCI väylä muille laitteille
- 16 ohjelmoitavaa I/O-johdinta (esim. näppäimet, LEDit)
- 3 ohjelmoitavaa ajastinta (kellolaitekeskeytyksiin)
- suunnattu halpoihin kannettaviin laitteisiin (esim. kämmenmikroihin)

Sun ultraJAVA (SUN Microsystems)

- nopeampi, parempi, kalliimpi
- suunnattu grafiikka- ja multimediasovelluksiin (esim. kämmenpelikone)

JEM (Rockwell Collins)
PCS1000 (Patriot Scientific) - lääketieteellisiä pienlaitteita
MJ501 (LG Semicon) - TV, älykortit

JSR 1 = Java Specification Request for Real-Time Specification for Java

- aJ-80, aJ-100

PicoJava II arkkitehtuurista on useita toteutuksia. Sun'illa niitä on ainakin kaksi kappaletta. Microjava 701 on suunniteltu pieniin kannettaviin laitteisiin, jotka eivät tarvitse suurta laskentatehoa. Niissä voi olla kuitenkin välimuisti tarvittaessa. Ulkoisia laitteita varten on PCI väylä. Käyttöliittymän toteuttamiseksi on 16 erityistä ohjelmoitavaa I/O-johdinta, joita voidaan käyttää vaikkapa näppäimien tai LED-valojen kontrollointiin. Käyttöjärjestelmää ja sovellusten osia voi herätellä 3 erilaisen ajastimen avulla.

PicoJAVA II toteutuksia

Sun microJAVA 701 (SUN Microsystems)

- valinnainen välimuisti, oma muistiväylä, PCI väylä muille laitteille
- 16 ohjelmoitavaa I/O-johdinta (esim. näppäimet, LEDit)
- 3 ohjelmoitavaa ajastinta (kellolaitekeskeytyksiin)
- suunnattu halpoihin kannettaviin laitteisiin (esim. kämmenmikroihin)

Sun ultraJAVA (SUN Microsystems)

- nopeampi, parempi, kalliimpi
- suunnattu grafiikka- ja multimediasovelluksiin (esim. kämmenpelikone)

JEM (Rockwell Collins)
PCS1000 (Patriot Scientific) - lääketieteellisiä pienlaitteita
MJ501 (LG Semicon) - TV, älykortit

JSR 1 = Java Specification Request for Real-Time Specification for Java

- aJ-80, aJ-100

UltraJAVA on myös Sun'in suunnitelema. Se on suunniteltu vaativampaan laskentaan kuin edellinen laitteisto, kuten esimerkiksi kämmenpelikoneisiin. Kumpikaan näistä suorittimista ei ilmeisesti enää ole tuotannossa. Niiden pohjana oleva PicoJAVA II määrittely sen sijaan elää edelleen muiden valmistajien suorittimissa.

PicoJAVA II toteutuksia

Sun microJAVA 701 (SUN Microsystems)

- valinnainen välimuisti, oma muistiväylä, PCI väylä muille laitteille
- 16 ohjelmoitavaa I/O-johdinta (esim. näppäimet, LEDit)
- 3 ohjelmoitavaa ajastinta (kellolaitekeskeytyksiin)
- suunnattu halpoihin kannettaviin laitteisiin (esim. kämmenmikroihin)

Sun ultraJAVA (SUN Microsystems)

- nopeampi, parempi, kalliimpi
- suunnattu grafiikka- ja multimediasovelluksiin (esim. kämmenpelikone)

JEM (Rockwell Collins)
PCS1000 (Patriot Scientific) - lääketieteellisiä pienlaitteita
MJ501 (LG Semicon) - TV, älykortit

JSR 1 = Java Specification Request for Real-Time Specification for Java

- aJ-80, aJ-100

Myös monella muulla valmistajalla on PicoJAVA II'n määrittelyn perusteella tehtyjä toteutuksia. Esimerkiksi Patriot Scientific'in PCS1000 toteutus on suunniteltu nimenomaan lääketieteellisiä pienlaitteita varten.

PicoJAVA II toteutuksia

Sun microJAVA 701 (SUN Microsystems)

- valinnainen välimuisti, oma muistiväylä, PCI väylä muille laitteille
- 16 ohjelmoitavaa I/O-johdinta (esim. näppäimet, LEDit)
- 3 ohjelmoitavaa ajastinta (kellolaitekeskeytyksiin)
- suunnattu halpoihin kannettaviin laitteisiin (esim. kämmenmikroihin)

Sun ultraJAVA (SUN Microsystems)

- nopeampi, parempi, kalliimpi
- suunnattu grafiikka- ja multimediasovelluksiin (esim. kämmenpelikone)

JEM (Rockwell Collins)

PCS1000 (Patriot Scientific) - lääketieteellisiä pienlaitteita

MJ501 (LG Semicon) - TV, älykortit

JSR 1 = Java Specification Request for Real-Time Specification for Java

- aJ-80, aJ-100

www.jcp.com www.ajile.com

PicoJAVA II'n suurimpia ongelmia on sen soveltumattomuus tosiaikasovelluksiin, lähinnä roskien keruu -ongelman vuoksi. Sun'in 'Java Community Process' yhteisö on yrittänyt ratkoa ongelmaa määrittelemällä speksejä tosiaika-Javalle. Työ on alkanut vuonna 1998 ja on käynnissä ecelleen.

MAJC - Microprocessor Architecture for Java Computing

Sun Microsystems'in suoritinarkkitehtuurin määrittely

- tavoitteena suuri nopeus Java, C ja C++ sovelluksille
- suunnattu multimediasovelluksiin (langattomassa) verkossa
  - interaktiivinen TV, virtuaalitodellisuussovellukset, pelit
- tulee hyvin JIT-käännöstä (nopeus on tärkeätä!)

MAJC 5200 - Sun'in MAJC-arkkitehtuurin toteutus

- 1-4 suoritinta (1999 esiteltiin 2 suorittimen lastu)
- useiden (peräkkäin kutsuttavien) metodien samanaikainen suoritus eri suorittimilla
  - ennakoivalle (spekulatiiviselle) suoritukselle oma kasa
  - peruutus (rollback), jos spekuloitu suoritus tehtiin liian aikaisin tai turhaan
- 4 säiettä suorituksessa per suoritin
  - säikeen vaihto nopeampaa kuin muistista luku
  - välimuistin hudin aikana suoritetaan muita säikeitä
  - laiterekisterit kaikille 4:lle säikeelle: hyper-threaded processor
- VLIW-arkkitehtuuri: 4 konekäskyä suorituksessa samanaikaisesti per prosessori

Sun on tehnyt myös toisen Java-suoritinarkkitehtuurin määrittelyn, MAJC'n. Se on suunniteltu nimenomaan suurta nopeutta vaativiin sovelluksiin ja tästä syystä siinä on keskitytty JIT-käännökseen perustuvaan suoritusmoodiin. Painopiste on siis muualla kuin Javan tavukoodin suorituksessa. Ajateltuina sovellusalueina olisivat interaktiiviset TV-palvelut ta erilaiset virtuaalitodellisuussovellukset, joista tietenkin videopelit ovat hyvä esimerkki.

MAJC - Microprocessor Architecture for Java Computing

Sun Microsystems'in suoritinarkkitehtuurin määrittely

- tavoitteena suuri nopeus Java, C ja C++ sovelluksille
- suunnattu multimediasovelluksiin (langattomassa) verkossa
  - interaktiivinen TV, virtuaalitodellisuussovellukset, pelit
- tulee hyvin JIT-käännöstä (nopeus on tärkeätä!)

MAJC 5200

Sun XVR-1000 Video Card (sisältää MAJC 5200)

MAJC 5200 - Sun'in MAJC-arkkitehtuurin toteutus

- 1-4 suoritinta (1999 esiteltiin 2 suorittimen lastu)
- useiden (peräkkäin kutsuttavien) metodien samanaikainen suoritus eri suorittimilla
  - ennakoivalle (spekulatiiviselle) suoritukselle oma kasa
  - peruutus (rollback), jos spekuloitu suoritus tehtiin liian aikaisin tai turhaan
- 4 säiettä suorituksessa per suoritin
  - säikeen vaihto nopeampaa kuin muistista luku
  - välimuistin hudin aikana suoritetaan muita säikeitä
  - laiterekisterit kaikille 4:lle säikeelle: hyper-threaded processor
- VLIW-arkkitehtuuri: 4 konekäskyä suorituksessa samanaikaisesti per prosessori

Sun on tehnyt ensimmäisen toteutuksen myös tästä arkkitehtuurista. Toteutus on nimeltään MAJC-5200 ja siinä on 4 suoritinta. Arkkitehtuuri oli jo ilmestyessään 1999 hyvin uraauurtava. Suorittimia oli siis neljä kappaletta, jolloin 4 eri säiettä voitiin suorittaa samanaikaisesti. Tai sitten yhtä säiettä voitiin suorittaa hyvin nopeasti ovelalla jipolla, jossa peräkkäisiä metodin kutsuja oli useita samanaikaisesti suorituksessa, kukin omalla suorittimellaan. Kullakin suorittimella oli neljät rekisterit, joiden avulla muistinoutoa odottaessa sille löytyi joku muu säie suoritukseen. Lisäksi vielä kunkin säikeen suoritus oli nopeata, koska yhdellä kertaa voitiin suorittaa jopa neljää peräkkäistä konekäskyä samanaikaisesti.

C# eli "C sharp"

```
// Hello1.cs
public class Hello1
{
   public static void Main()
   {
      System.Console.WriteLine("Hello, world!");
   }
}
```

http://msdn.microsoft.com/library

Javan kaltainen kieli

- Anders Hejlsberg
  - kehitti myös Turbo Pascal, Delphi ja J++ kielet

Osa Microsoftin .Net-ympäristöä

- Mono - open source .Net for Linux www.go-mono.com
- nivoutuu hyvin Windows XP'n kanssa

ECMA (European Computer Manufacturers Association) standardi

- mm. Microsoft, HP ja Intel

MSIL - MicroSoft Intermediate Language (virtuaalikoneen välikieli)

- sopiva välikieli 'kaikille' korkean tason kielille
  - C#, C, C++, Pascal, Java, Visual Basic, etc
- suoritus ainoastaan (JIT) käännösten avulla
- CLR (Common Language Runtime) virtuaalikoneen määrittely

Java on Sunin kehittämä kieli, ja Sun Microsystems omistaa siihen kaikki oikeudet. Microsoft teki siihen pieniä lisäyksiä, jotka Microsoftin mukaan tekivät Javasta paremmin yhteensopivan Mcrosoftin muiden systeemien kanssa. Sun ei pitänyt tästä, nosti oikeusjutun Microsoftia vastaan ja voitti. No, Microsoft vastasi omalla tavallaan ja kehitti oman Javan, jonka nimeksi tuli C#. C# ei ole ilmeisesti lainkaan huono, vaikka sen käyttö ei olekaan levinnyt maailmalle samalla tavalla kuin Javan käyttö. Lisäksi Microsoft sai muita yhtiöitä mukaan ja määritteli C#'lle Eurooppalaisen standardin, joten C# on nyt jopa vakaampi kieli kuin Java.

C# eli "C sharp"

Javan kaltainen kieli

- Anders Hejlsberg
  - kehitti myös Turbo Pascal, Delphi ja J++ kielet

Osa Microsoftin .Net-ympäristöä

- Mono - open source .Net for Linux www.go-mono.com
- nivoutuu hyvin Windows XP'n kanssa

ECMA (European Computer Manufacturers Association) standardi

- mm. Microsoft, HP ja Intel

MSIL - MicroSoft Intermediate Language (virtuaalikoneen välikieli)

- sopiva välikieli 'kaikille' korkean tason kielille
  - C#, C, C++, Pascal, Java, Visual Basic, etc
- suoritus ainoastaan (JIT) käännösten avulla
- CLR (Common Language Runtime) virtuaalikoneen määrittely

Javan välikielen tapaan C#'lla on väikieli, MSIL. Microsoftille MSIL on kuitenkin paljon enemmän kuin pelkkä C#'n välikieli. MSIL on kaikkien Microsoftin käyttämien ohjelmointikielten yhteinen välikieli, jonka avulla voidaan toteuttaa esimerkiksi dynaaminen linkitys verkon kautta. Normaalistihan linkitys tapahtuu vasta objektimoduulitasolla. Kaikki ohjelmointikielet käännettäisiin siis MSIL'ällä, jolla esitettyjä ohjelmia voi sitten suorittaa millä tahansa C#'in virtuaalikoneella, CLR'llä. Toisin kuin Javan tavukoodin, MSIL'n suoritusta ei ole suunniteltu tulkittavaksi vaan suoran tai JIT-käännöksen avulla tehtäväksi. C#-ohjelmien suoritusnopeuden pitäisi tällä tavoin olla vastaava kuin tavallisilla ohjelmontikielillä (esim. C tai C++) tehtyjen ohjelmien.

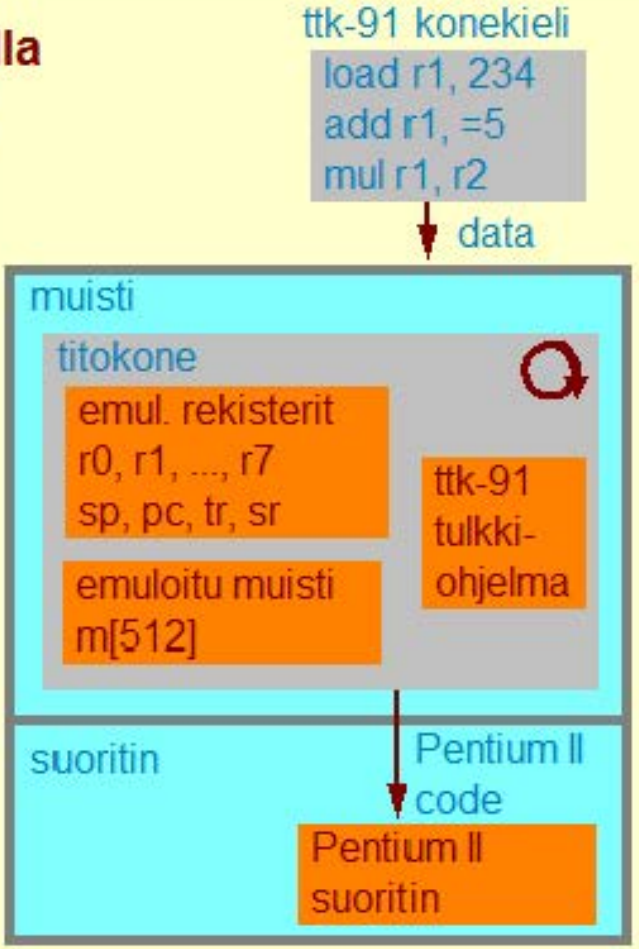
Ttk-91 arkkitehtuurin emulointi tulkin avulla

Tulkki on osana Titokone-simulaattoria

Yksi ttk-91 konekielen käsky kerrallaan

Ttk-91 koneen rekisterit ja muisti emuloitu tulkin tietorakenteina muistissa

Oma aliohjelma tai koodinpätkä jokaiselle ttk-91 konekäskylle



Muistelkaa, kuinka Java virtuaalikoneen tulkki lukee Javan tavukoodin käskyjä yksi käsky kerrallaan ja emuloi niiden suoritusta muokkaamalla JVM:n emuloituja rakenteita tulkin muistissa. Ttk-91 koneen tulkki toimii täsmälleen samalla tavalla. Se lukee datana ttk-91 konekielisiä käskyjä syötetiedostosta ja emuloi niiden toimintaa muokkaamalla tietorakenteina toteutettuja ttk-91 koneen emuloituja osia, kuten rekistereitä ja muistia. Erona JVM suoritukseen on lähinnä se, että ttk-91 ja sen suoritusympäristö eivät ole niin täydellisesti määriteltyjä kuin JVM, joten ttk-91'lle tehdyt ohjelmat ovat aika rajoitettuja I/O:n ja KJ-palvelujen suhteen.

Transmetan Crusoe suoritin

Intelin x86-konekielen emulointi, JIT-käännös

Natiivi käskykanta ei tärkeä sinällään

- kaikki suoritus emulointia, myös KJ!

Tavoite: nopeampi, sama lastun teknologia?

Tulos: yhtä nopea, vähemmän virtaa

Laitteistotuki emuloinnille

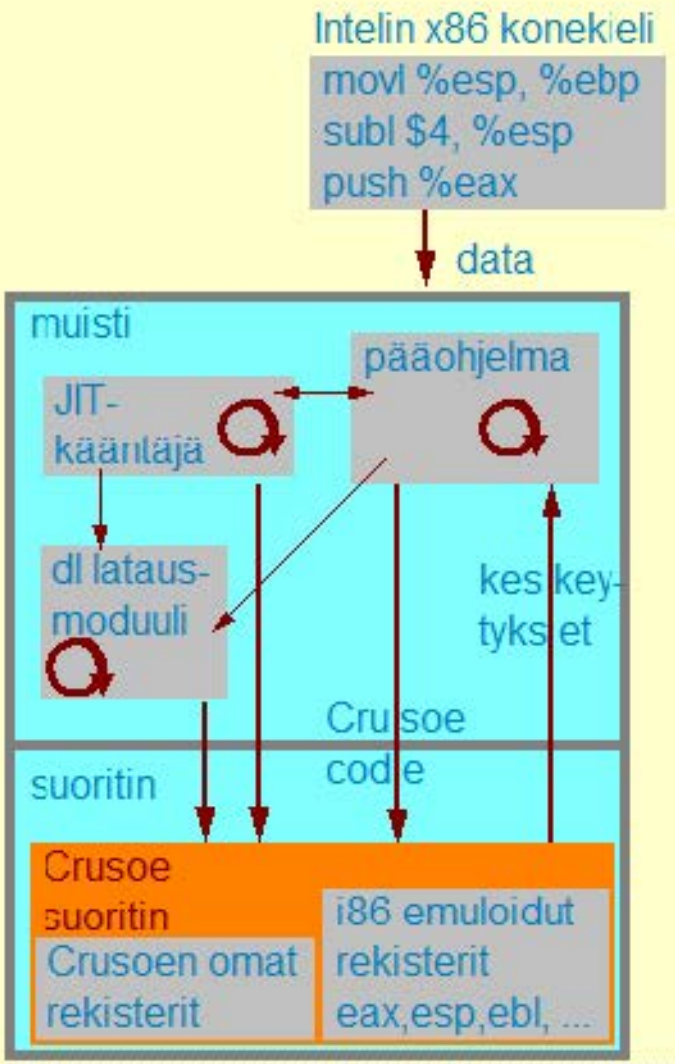
- emuloitavat x86-rekisterit laiterekistereinä
- emuloidulle x86-muistille laitetukea

Nopea emulointi

- monta x86-käskyä samaan aikaan emuloinnissa, sekajärjestyksessä, laitteistotuen avulla

Tarkat keskeytykset

- suorituksen peruutus
- uusi käännös hitaalle koodille
- uusi hidas tarkka emulointi keskeytyskohtaan



Transmetan Crusoe-suoritin on mielenkiintoinen toteutus. Transmetan perustajat olivat aikaisemmin tutkineet ja toteuttaneet useitakin Intelin arkkitehtuurin täsmällisiä simulaattoreita. Funktionaalisuus oli vielä suhteellisen helppoa toteuttaa, mutta simulaattorit jäivät aina jälkeen todellisten laitteistojen suorituskyvyssä. Kantavana ideana Crusoessa on, että suunnitellaan suoritin nimenomaan täsmällistä emulointia varten, eli siis suoritin, joka tekee täsmälleen samat asiat kuin Intelin suoritin. Ei liene kaukaa haettu kysymys, että miksi?

Transmetan Crusoe suoritin

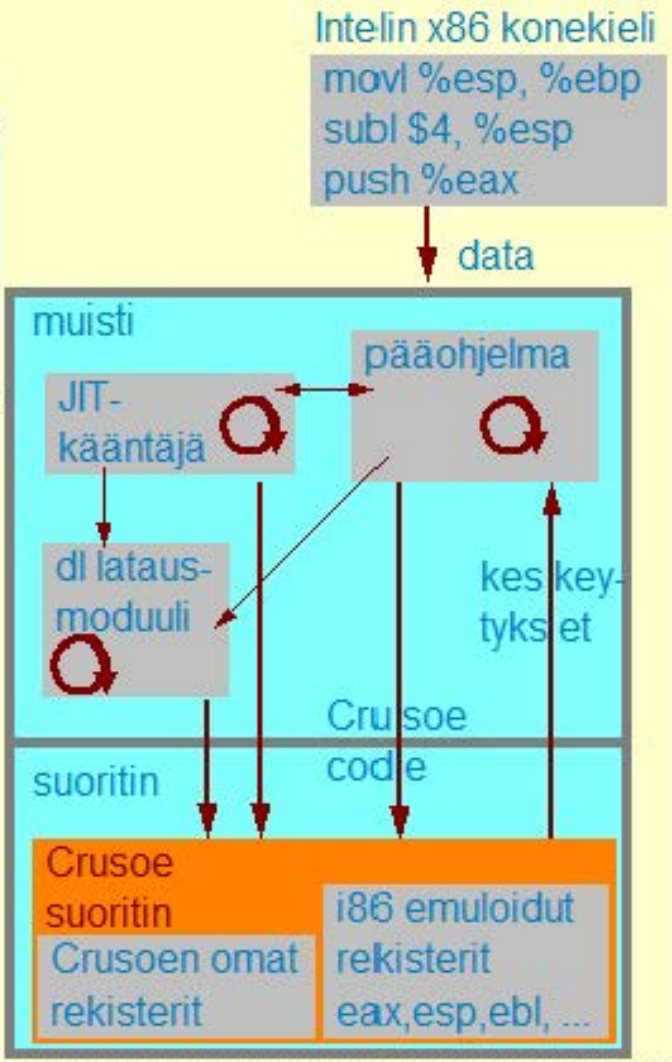
Intelin x86-konekielen emulointi, JIT-käännös
Natiivi käskykanta ei tärkeä sinällään
- kaikki suoritus emulointia, myös KJ!

Tavoite: nopeampi, sama lastun teknologia?
Tulos: yhtä nopea, vähemmän virtaa

Laitteistotuki emuloinnille
- emuloitavat x86-rekisterit laiterekistereinä
- emuloidulle x86-muistille laitetukea

Nopea emulointi
- monta x86-käskyä samaan aikaan emuloinnissa, sekajärjestyksessä, laitteistotuen avulla

Tarkat keskeytykset
- suorituksen peruutus
- uusi käännös hitaalle koodille
- uusi hidas tarkka emulointi keskeytyskohtaan



Ideana on siis tehdä samat asiat kuin alkuperäinen Intelin arkkitehtuurikin, mutta miksi? Tutkijoiden oivallus oli, että emulointi voisi sopivan laitteistotuen avulla olla nopeampaa kuin tavanomainen suorittimen toteutus. Esimerkiksi, emulaattorin käskykanta voisi olla yksinkertaisempi ja siten nopeampi kuin Intelin toteutus. Idea oli hyvä, mutta tavoite karkasi vähän käsistä, kun Intelillä otettiin vähän samantapaisia ajatuksia käyttöön myös Intelin omissa suorittimissa. Yksinkertaisemman suorittimen toisena etuna on pienempi virrankulutus, joka sitten jäikin Crusoen markkinavaltiksi. Pieni virrankulutus kun on mukava asia kannettavissa laitteissa.

Transmetan Crusoe suoritin

Intelin x86-konekielen emulointi, JIT-käännös

Natiivi käskykanta ei tärkeä sinällään

- kaikki suoritus emulointia, myös KJ!

Tavoite: nopeampi, sama lastun teknologia?

Tulos: yhtä nopea, vähemmän virtaa

Laitteistotuki emuloinnille

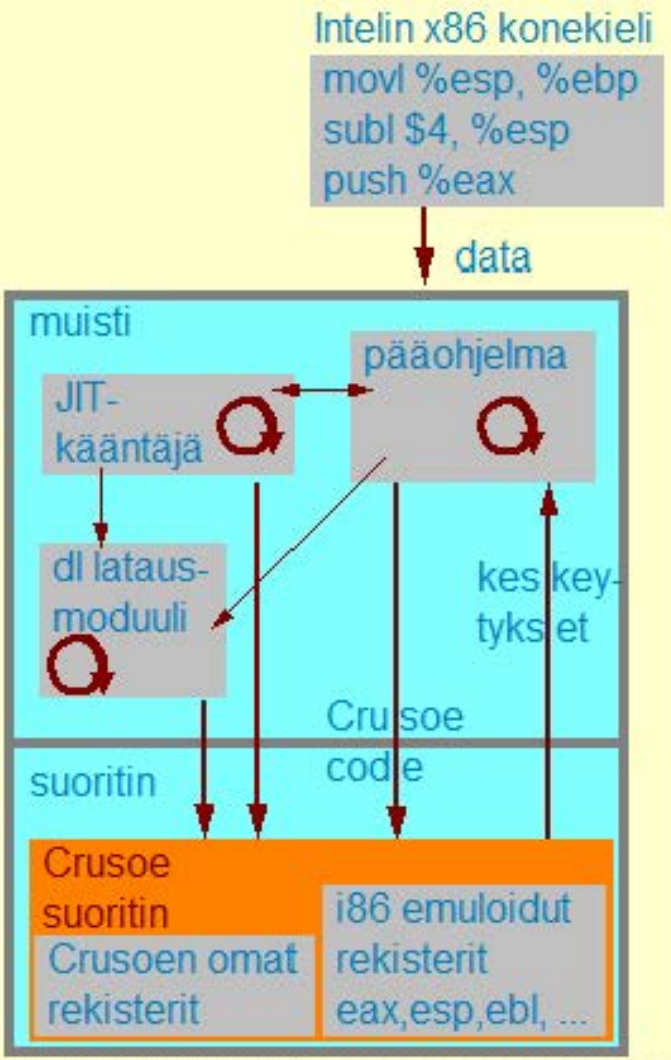
- emuloitavat x86-rekisterit laiterekistereinä
- emuloidulle x86-muistille laitetukea

Nopea emulointi

- monta x86-käskyä samaan aikaan emuloinnissa, sekajärjestyksessä, laitteistotuen avulla

Tarkat keskeytykset

- suorituksen peruutus
- uusi käännös hitaalle koodille
- uusi hidas tarkka emulointi keskeytyskohtaan



Emuloinnin nopeuttamiseksi Crusoen suorittimella on useita nimenomaan emulointia varten lisättyjä laitteita. Hyvänä esimerkkinä on emuloitavan suorittimen laiterekistereiden toteutus Crusoen omina ylimääräisinä laiterekistereinä. Emuloitavien rekistereiden käyttö on selvästi nopeampaa tällä tavoin. Normaalistihan ne on emulaattoreissa ja simulaattoreissa toteutettu muistissa olevina tietorakenteina. Toinen nopeutta antava piirre on se, että emuloitavia Intelin arkkitehtuurin konekäskyjä ei emuloida yksi kerrallaan, vaan useita yhtä aikaa. Ryhmä Intelin käskyjä käännetään Crusoen konekäskyjen joukoksi, joka suoritetaan Crusoella vielä rinnakkain sillä tavoin, että jopa kymmeniä Crusoen konekäskyjä on samaan aikaan suorituksessa.

Transmetan Crusoe suoritin

Intelin x86-konekielen emulointi, JIT-käännös

Natiivi käskykanta ei tärkeä sinällään

- kaikki suoritus emulointia, myös KJ!

Tavoite: nopeampi, sama lastun teknologia?

Tulos: yhtä nopea, vähemmän virtaa

Laitteistotuki emuloinnille

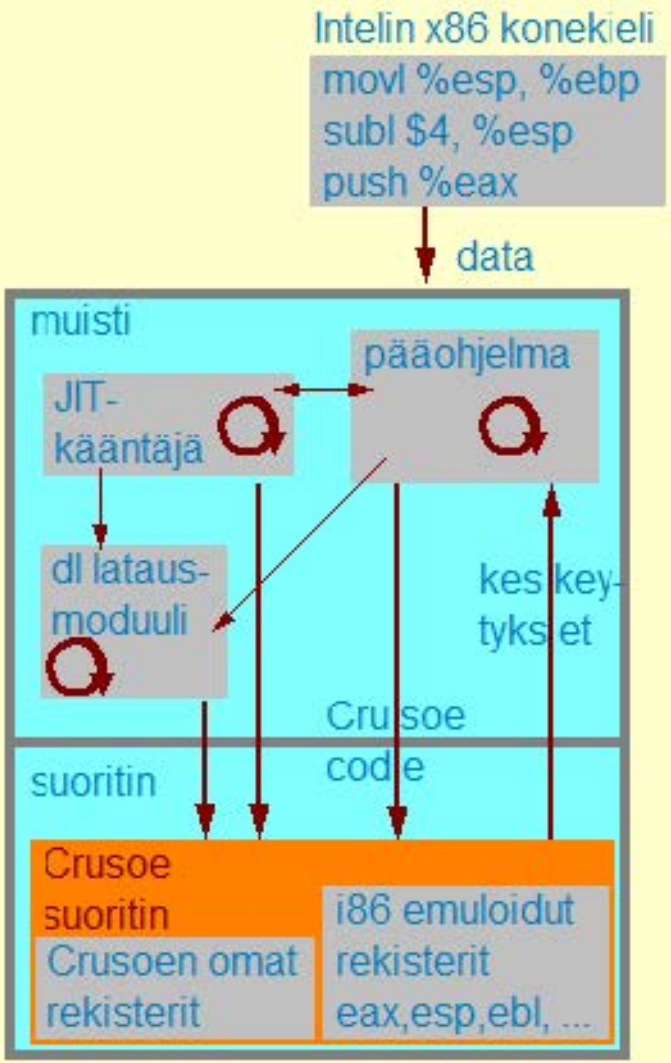
- emuloitavat x86-rekisterit laiterekistereinä
- emuloidulle x86-muistille laitetukea

Nopea emulointi

- monta x86-käskyä samaan aikaan emuloinnissa, sekajärjestyksessä, laitteistotuen avulla

Tarkat keskeytykset

- suorituksen peruutus
- uusi käännös hitaalle koodille
- uusi hidas tarkka emulointi keskeytyskohtaan



Edellämainittu suoritus aiheuttaa ongelmia keskeytysten kanssa. Useat keskeytykset pitäisi pystyä käsittelemään välittömästi kyseisen Intelin konekäskyn jälkeen, mutta Crusoen suorittimella yhden Intelin konekäskyn suoritusta ei enää voida eristää. Ratkaisuna tälle on tietokantamaailmasta tuttu tapahtumien peruutus. Yhden Intelin konekäskyryhmän suorituksen jälkeen tilanne on vakaa ja se pistetään talteen. Tarkan keskeytyksen tapahtuessa suoritus peruutetaan talletettuun tilaan. Emulointi jatkuu sitten tästä pisteestä hitaasti, yksi Intelin konekäsky kerrallaan, kunnes päästään uudestaan keskeytystilanteeseen. Tällä kertaa keskeytys voidaan sitten käsitellä täsmälleen samalla tavalla kuin Intelin suoritinkin sen käsittelisi.

Transmetan Crusoe suoritin

Intelin x86-konekielen emulointi, JIT-käännös

Natiivi käskykanta ei tärkeä sinällään

- kaikki suoritus emulointia, myös KJ!

Tavoite: nopeampi, sama lastun teknologia?

Tulos: yhtä nopea, vähemmän virtaa

Laitteistotuki emuloinnille

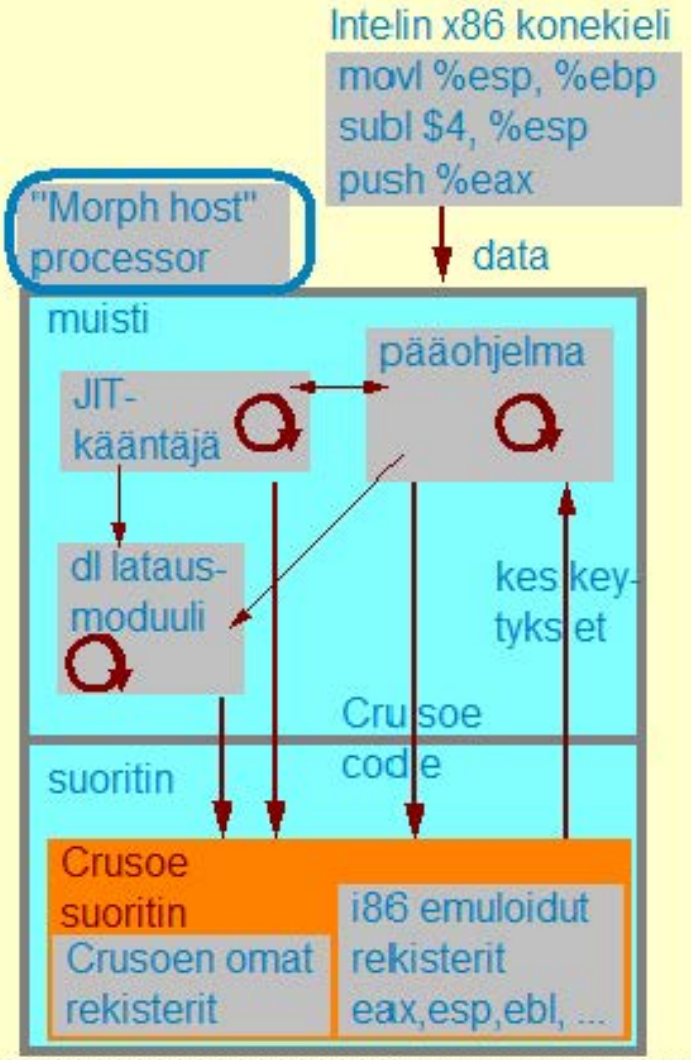
- emuloitavat x86-rekisterit laiterekistereinä
- emuloidulle x86-muistille laitetukea

Nopea emulointi

- monta x86-käskyä samaan aikaan emuloinnissa, sekajärjestyksessä, laitteistotuen avulla

Tarkat keskeytykset

- suorituksen peruutus
- uusi käännös hitaalle koodille
- uusi hidas tarkka emulointi keskeytyskohtaan



Transmetaa perustettaessa tarvittiin tietenkin huomattava alkupääoma. Kun Crusoe suorittimen toteutusmallia yritettiin selittää mahdollisille sijoittajille, asia ei oikein mennyt jakeluun, eikä rahaa tullut. Sitten hoksatttin antaa koko kokonaisuudelle uusi nimi, 'morph host processor', joka kuulosti siltä, että olisi keksitty jokin aivan uudenlainen suoritin. Näinhän ei tietenkään ollut, mutta nyt sijoittajia löytyi.

Transmetan Crusoe suoritin

Intelin x86-konekielen emulointi, JIT-käännös

Natiivi käskykanta ei tärkeä sinällään

- kaikki suoritus emulointia, myös KJ!

Tavoite: nopeampi, sama lastun teknologia?

Tulos: yhtä nopea, vähemmän virtaa

Laitteistotuki emuloinnille

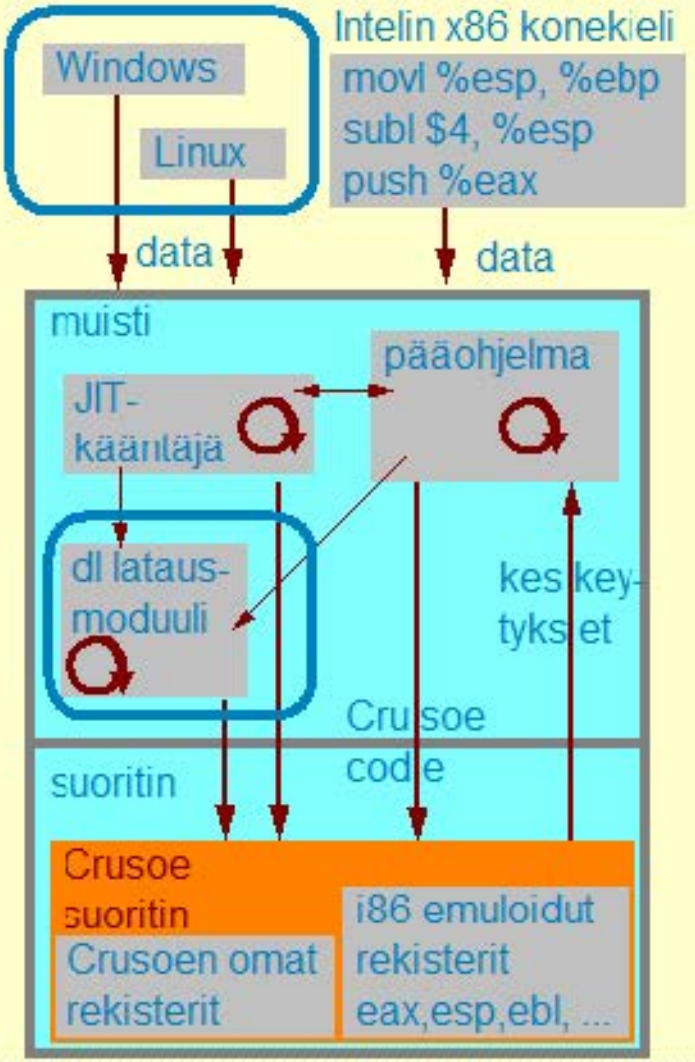
- emuloitavat x86-rekisterit laiterekistereinä
- emuloidulle x86-muistille laitetukea

Nopea emulointi

- monta x86-käskyä samaan aikaan emuloinnissa, sekajärjestyksessä, laitteistotuen avulla

Tarkat keskeytykset

- suorituksen peruutus
- uusi käännös hitaalle koodille
- uusi hidas tarkka emulointi keskeytyskohtaan



Crusoe-suorittimeen liittyy olennaisena osana aina kaikki emulaattorin ohjelmistot, jotka vievät merkittävästi muistitilaa. Varsinaista käyttöjärjestelmää ei ole, vaan mikä tahansa normaali Intelin arkkitehtuurille käännetty käyttöjärjestelmä tulkitaan sellaisenaan. Eli vaikka suoritin on Crusoe, kaikki käytettävät ohjelmistot, käyttöjärjestelmä mukaanlukien, voidaan antaa x86-muotoisina. Suorituskyvyn vuoksi on järkevää pitää usein käytettävistä moduuleista tallessa valmiiksi Crusoelle käännetyt latausmoduulit. Linuxin kehittäjän Linus Torvalds'in ensimmäinen työpaikka USA:ssa oli juuri Transmeta, jossa hän porttasi Linuxin Crusoe-suorittimelle.

Tulkinta ja emulointi

Java-ohjelman suoritus
Java tavukoodi, Java-virtuaalikone

Java-ohjelmien emulointi ja käännös
Java-suorittimet

C#
ttk-91, Crusoe

Olemme nyt käyneet läpi emuloinnin käyttämistä ohjelmien suoritustapana. Emuloinnissa on olennaista, että ohjelmiston avulla toteutetaan kaikki ne samat toiminnot, jotka todellisessa suorittimessa tehdään laitteistolla. Emuloinnin etuna on joustavuus, mutta suoritusnopeus yleensä kärsii. Esittelimme Java-ohjelmien kaikki suoritustavat ja JVM'n rakenteen. Esittelimme myös lyhyesti Microsoftin kilpailijan Javalle, C#, ja C#'n suoritusympäristön. Kävimme lopuksi läpi Crusoe-suorittimen, joka on alkuaankin suunniteltu nimenomaan emulointia varten.