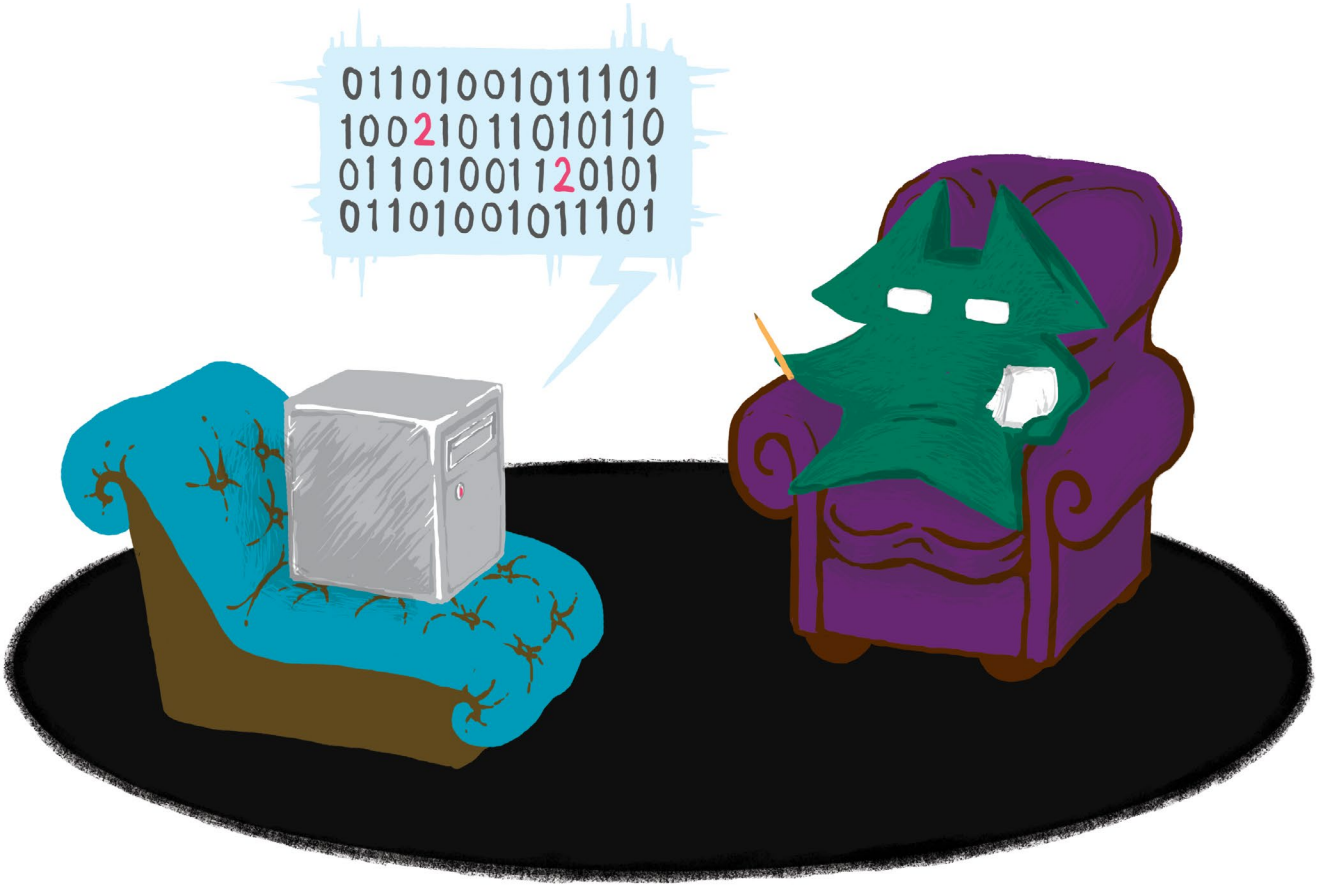


Konekielen kiemurat: Portti tietokoneen sielunelämään

Konekieli on aina kuulostanut hohdokkaalta tietokoneharrastajan korvaan – onhan se laitteenläheisin taso, jolle ohjelmoija pääsee. Vaikka konekieli ei olekaan enää portti taianomaisiin ohjelmointitemppuihin, auttaa sen ymmärtäminen edelleen käsittämään tekniikkaa.

Teksti: Ville-Matias Heikkilä

Kuvat: Mitol Meerna, Ville Matias Heikkilä, Visual6502.org, AMD

Aiemmin konekieliohjelmointi oli liki pakollinen taito esimerkiksi peli- tai demo-ohjelmoijalle, mutta nykyisin konekieltä tuottavat lähinnä korkean tason kielten kääntäjät. Konekielen lukutaidosta on silti edelleen hyötyä: kääntäjän työtä pystyy arvioimaan ja lähdekoodittomia ohjelmia tutkimaan ja muokkaamaan. Niin kone kuin sen ohjelmatkin muuttuvat taidon myötä paljon käsinkosketeltavammiksi. Vanhojen koneiden, mikrokontrollerien ja matalan tason tietoturva-aukkojen parissa askarteleville konekieli on edelleen tärkeä väline.

Konekieliä on monia

Kaikki koneet eivät suinkaan ymmärrä samaa konekieltä. Esimerkiksi PC-suorittimet käyttävät X86-konekieltä ja mobiililaitteet useimmiten ARM-konekieltä. Yksittäistä konekieltä kutsutaan myös käskykannaksi tai arkkitehtuuriksi.

Tässä artikkelissa keskitytään selkeyden vuoksi neljään käskykantaan mikrotietotekniikan historian varrelta: 6502, X86, 68K ja ARM. Koska nämä käskykannat eroavat melkoisesti suunnittelufilosofioiltaan, saa niihin tutustumalla myös yleiskäsityksen siitä, millaisia konekieliä on olemassa.

MOS Technologyn 6502 on ollut suosituimpia 8-bittisiä suorittimia. Sitä lähisukulaisineen ja klooneineen ovat käyttäneet esimerkiksi Applen, Atarin ja Commodoren 8-bittiset koneet sekä Nintendon NES. 6502:n perinteinen kilpailija on ollut Zilogin Z80, joka perustuu Intelin 8080:aan.

Eri konekielten käskyjä biteiksi purettuna.

64-bittisen nyky-X86:n rekisteristön vanhimmat osat ovat 70-luvulta.

Uudempia, lähinnä sulautetuissa järjestelmissä käytettyjä 8-bittisiä käskykantoja ovat AVR ja PIC.

Intelin X86 on tuttu IBM PC -yhteensopivista koneista. Alun perin 16-bittistä käskykantaa on laajennettu ja uudistettu sittemmin radikaalisti – 386-suorittimessa 32-bittiseksi ja 2000-luvulla AMD:n aloitteesta 64-bittiseksi. Uudistuksista huolimatta historian eri kerrokset näkyvät edelleen hyvin läpi X86-konekielessä.

Motorolan 68K:ta käytti suurin osa PC:n kilpailijoista 90-luvun alkupuolelle asti: Amiga, Atari ST, Macintosh, Unix-työasemat. Se perustuu 70-luvun isompien tietokoneiden käskykantoihin ja on suunnittelultaan tyylipuhdas CISC (Complex Instruction Set Computer).

ARM on tämän hetken suosituin käskykanta. Se dominoi etenkin mobiililaitteita mutta saattaa ennen pitkää syrjäyttää myös X86:n. Alkujaan Archimedes-kotimikrossa käytetty käskykanta nousi suosioon, koska se tarjosi paljon tehoa vähäisellä piimäärällä. ARM on RISC-käskykanta (Reduced Instruction Set Computer). Muita RISCejä ovat esimerkiksi MIPS, SPARC, PowerPC ja AVR.

Käskyn olemus

Konekielikäskyt ovat melko tiivisrakenteisia ykkösten ja nollien jonoja. Ohessa on esitetty saman tehtävän suorittava käsky usealla eri konekielellä. Kukin käsky summaa luvun 3 johonkin suorittimen sisäiseen rekisteriin, mutta esimerkiksi bittileveys vaihtelee: 6502:n `adc` laskee 8-bittisillä luvuilla, joiden summat voivat olla enintään muutamia satoja, kun taas ARMin 32 bittiä leveä toimitus yltää miljardeihin asti. Suorittimen tai käskykannan bittisyys tarkoittaa yleensä nimenomaan peruslaskutoimitusten suurinta bittileveyttä.

Ykkösten ja nollien jonot ovat ihmisille vaikeaselkoisia. Ihmiset käsittelevät konekieltä tämän vuoksi yleensä symbolisessa eli assembly-muodossa. Assembly-esityksestä voi myös arvata, mitä käsky tekee, vaikka käskykanta olisi ennestään tuntematon: esimerkiksi alkukirjaimet `ad(d)` viittaavat yhteenlaskuun. Samalla konekielellä voi olla useita erilaisia assembly-kielimuotoja, joita eri assembly-kääntäjät eli assemblerit käyttävät – esimerkiksi X86:lla Intel- ja AT&T-syntaksit.

Konekielikäsky jakautuu yleensä operaatiokoodiin, osoitusmuotoon ja operandeihin. Operaatio on ”verbi”, jota vastaa assemblyssä ensimmäinen sana, jota kutsutaan muistikaaksi (mnemonic), esimerkiksi `add`. Operandit ovat sen jäljessä tulevia ”substantiiveja”: rekistereitä, lukuja ja muistiosoituksia. Osoitusmuotoja taasen voi verrata ihmiskielten taivutusmuotoihin. Ne ilmaisevat, miten operandiosa tulkitaan – onko kyseessä esimerkiksi muistiosoite vai suora luku – ja antavat sille mahdollisia lisämääreitä: esimerkiksi 68K-käskyn pääte `.b`, `.w` tai `.l` ilmaisee, suoritetaanko toimitus 8-, 16- vai 32-bittisenä.

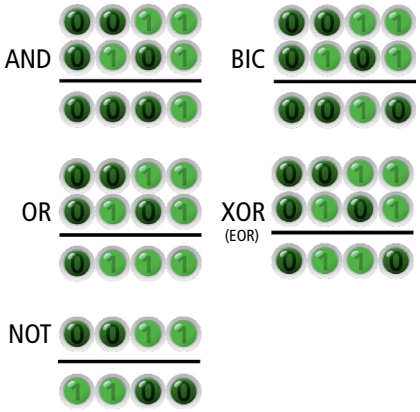
Data pyörii rekistereissä

Useimmissa konekielissä suurin osa datankäsittelystä tapahtuu rekistereissä, jotka voi ajatella suorittimensisäisiksi kiinteiksi muuttujiksi. Rekisterien määrät, leveydet ja käyttötavat vaihtelevat huomattavasti käskykannasta toiseen.

6502:n rekisteristö on hyvin pieni ja kukin rekisteri on sidottu tiettyihin tehtäviin. Suurin osa laskutoimituksissa on pakko tehdä akkurekisterissä eli A:ssa. Indeksirekisterit X ja Y soveltuvat lähinnä muistiosoitukseen ja silmukkalaskureiksi, mihin taas A ei sovellu. Näiden lisäksi 6502:ssa on vain pino-osoitin S, statusrekisteri P ja seuraavan käskyn muistiosoitteen ilmoittava käskyosoitin PC. PC on ainoa 16-bittinen rekisteri, muut ovat 8-bittisiä. Suppeaa rekisteriavaruutta täydentää niinsanottu nollasivu eli muistin 256 ensimmäistä tavua, ja monet muistiosoitustavat onnistuvatkin vain nollasivun kautta.

ARMin ja muiden riscien rekisteristö on puolestaan hyvin symmetrinen ja yleiskäyttöinen. Periaatteessa mitä tahansa rekisteriä voi käyttää mihin tahansa. Ainoat poikkeukset ovat rekisteri R15 eli käskyosoitin sekä erillinen statusrekisteri. Perus-ARMissa on 32-bittisiä rekistereitä 16, mutta useimmissa muissa risceissä perusrekistereitä on 32 tai enemmän.

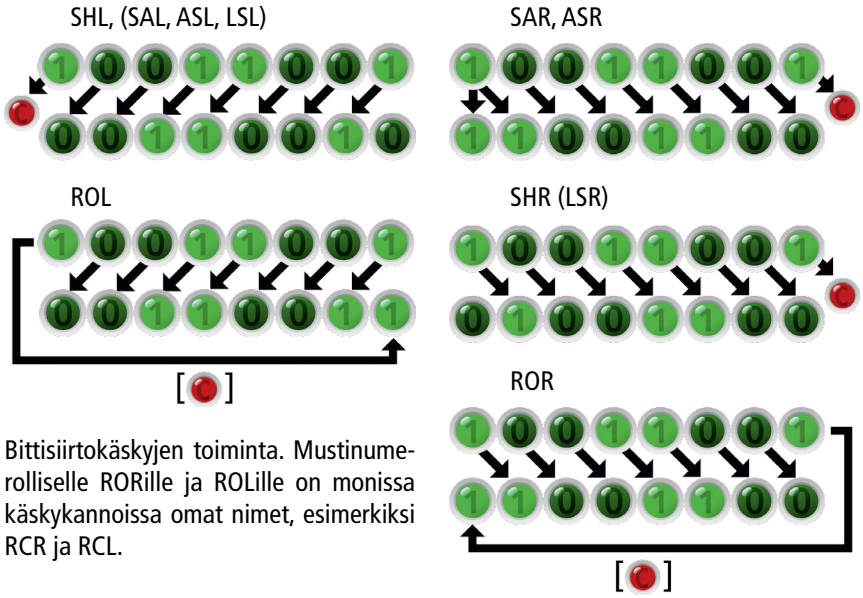
X86:n rekisterit ovat alkujaan erikoistuneita: esimerkiksi muistiosoitukseen on voinut käyttää vain rekistereitä BX, SI, DI ja BP. 32-bittinen päivitys kuitenkin löyhensi rajoituksia. Nykyisessä 64-bittisessä toimintamoodissakin on silti edelleen jäljellä käskyjä, jotka on sidottu tiettyihin rekistereihin: esimerkiksi yksitavuinen käsky `stosb` tallentaa 8-bittisen AL-rekisterin (*accumulator low*) sisällön muistipaikkaan, johon alkuperäisen DI-rekisterin (*destination*



Tämän artikkelin käskykantojen bitittäiset operaatiot. BIC on käytössä ARMissa.

	Intel-X86	68K	AT&T-X86
Operandijärjestys	add kohde,lähde	add.w lähde,kohde	addw lähde,kohde
Muistiosoitus	add ax,[1234]	add.w 1234,kohde	addw 1234,%ax
Välitön luku	add ax,1234	add.w #1234,kohde	addw $1234,%ax
Indeksoitu osoitus	[ebx+esi+8]	8(a0,d1.L)	8(%ebx,%esi)
Heksadesimaali	1234h	$1234	0x1234
Käskyn oma sijainti	jmp $	jmp pc	jmp .
Datatavu	db 123	ds.b 123	.byte 123

Assembly-syntaksit ovat yleensä melko samankaltaisia, mutta niiden välillä on myös hämääviä eroja. Tässä muutamia esimerkkejä.



Bittisiirtokäskyjen toiminta. Mustinumerolliselle RORille ja ROLille on monissa käskykannoissa omat nimet, esimerkiksi RCR ja RCL.

8×	4×	2×	1×	Etumerkitön	Etumerkillinen
0	0	0	0	0	+0
0	0	0	1	1	+1
0	0	1	0	2	+2
0	0	1	1	3	+3
0	1	0	0	4	+4
0	1	0	1	5	+5
0	1	1	0	6	+6
0	1	1	1	7	+7
1	0	0	0	8	-8
1	0	0	1	9	-7
1	0	1	0	10	-6
1	0	1	1	11	-5
1	1	0	0	12	-4
1	1	0	1	13	-3
1	1	1	0	14	-2
1	1	1	1	15	-1

Nelibittiset kokonaisluvut tulkittuna etumerkittöminä ja etumerkillisinä kahden komplementtia käyttäen.

index) 64-bittiseksi laajennettu versio RDI osoittaa.

68K:n perusrekisteristö on jaettu kahdeksaan data- ja osoiterekisteriin D0-D7 ja A0-A7, joista A7:ää käytetään pino-osoittimena. Erikseen on vielä statusrekisteri CCR ja käskyosoitin PC. Osoiterekisterit ovat alun perin 24-bittisiä mutta laajennettiin 68020-suorittimessa 32 bittiin. Kaikilla rekistereillä voi suorittaa laskutoimituksia melko yleiskäyttöisesti, mutta muistiosoitukset on hoidettava osoiterekistereillä.

Käskyt taipuvat osoitusmuodoissa

Rakenteeltaan yksinkertaisimmat konekäskyt eivät saa operandeja lainkaan, joten niiden toiminta on sidottu tiettyihin rekistereihin. Edellä mainittu X86:n `stosb` on esimerkki tästä niinsanotusta implisiittisestä osoitusmuodosta. Muita esimerkkejä ovat aliohjelmasta palaavat käskyt (`ret`, `rts`) ja lippujen nosto- ja laskukäskyt (`sec`, `clc`).

Käskyn tyypillinen operandimäärä vaihtelee konekielestä toiseen. 6502:ssa useimmat käskyt ovat yksioperandisia. Tämä operandi on yleensä muistiosoitus, jolloin laskutoimitus tapahtuu akkurekisterin ja muistipaikan välillä. X86 ja 68K ovat kaksioperandisia, eli käskyyn kuuluu lähde- ja kohdeoperandi. Tyypilliseen ARM-käskyyn puolestaan kuuluu kolme operandia – kaksi lähdettä ja yksi kohde. Nollaoperandisina konekielinä voidaan pitää Forth-tyylisiä pinopohjaisia konekieliä.

Useimmilla suorittimilla valtaosa konekielikoodista on rekisterienvälistä pyörittelyä. Operandeina voi kuitenkin käyttää rekisterien lisäksi myös välittömiä lukuja (*immediate*) tai erilaisia muistiviittauksia.

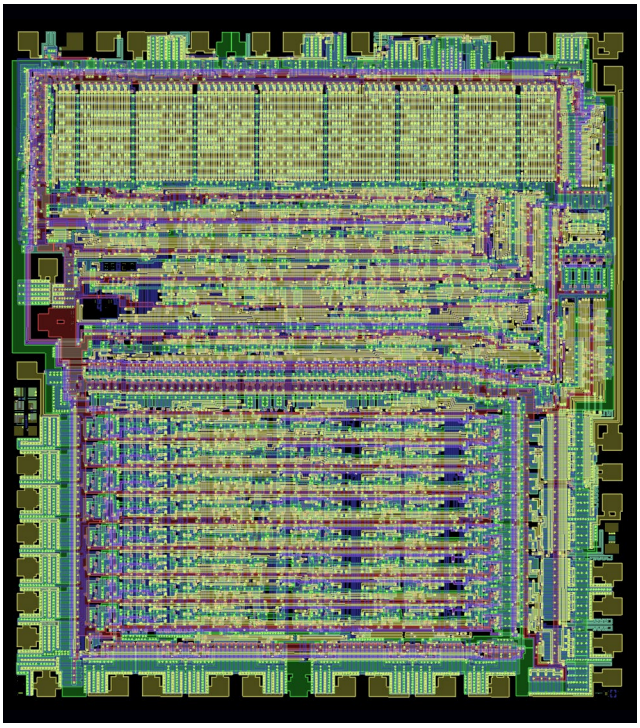
Operandien yhdistämisessä on usein

```
clc         asl $FE
lda $FE     rol $FF
adc #$34    asl $FE
sta $FE     rol $FF
lda $FF     asl $FE
adc #$12    rol $FF
sta $FF
```

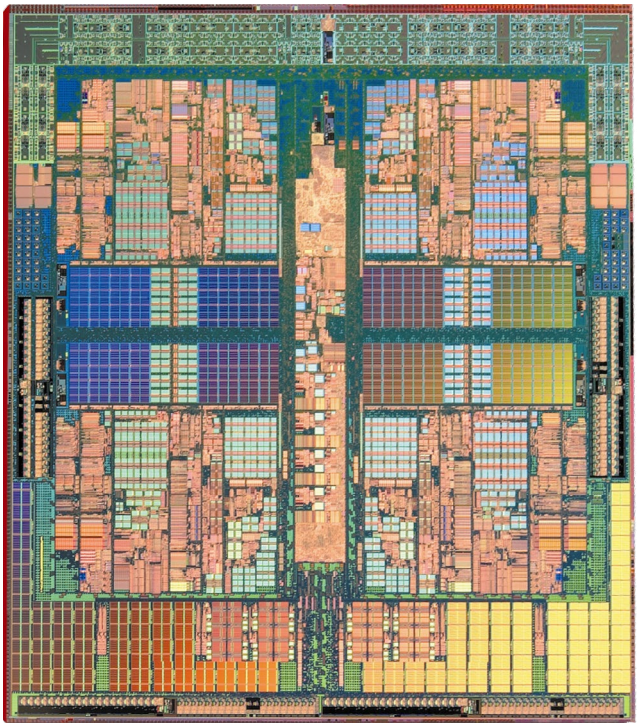
16-bittisten lukujen käsittelyä 8-bittisellä 6502:lla: vasemmanpuoleinen esimerkki lisää muistipaikkoihin $FE ja $FF tallennetun luvun arvoon heksadesimaaliluvun $1234, oikeanpuoleinen kertoo sen kahdeksalla bittisiirtoja käyttäen.

```
lp: cmp r0,r1
    subgt r0,r0,r1
    suble r1,r1,r0
    bne lp
```

Suurimman yhteisen tekijän laskeva silmukka ARMille ehdollista suoritusta käyttäen. Euklideen algoritmi vähentää suuremmasta luvusta pienemmän, kunnes luvut ovat yhtäsuuret.



6502-suorittimen sisukset. Alapuoliskoa hallitsee 8-kaistainen laskenta- ja rekisteriyksikkö, ylimpänä puolestaan näkyy käskyt suoritusvaiheiksi kuvaava mikrokooditaulukko. Näiden välissä on muu toimintalogiikka, esimerkiksi hyppyjen ja lippujen käsittely.



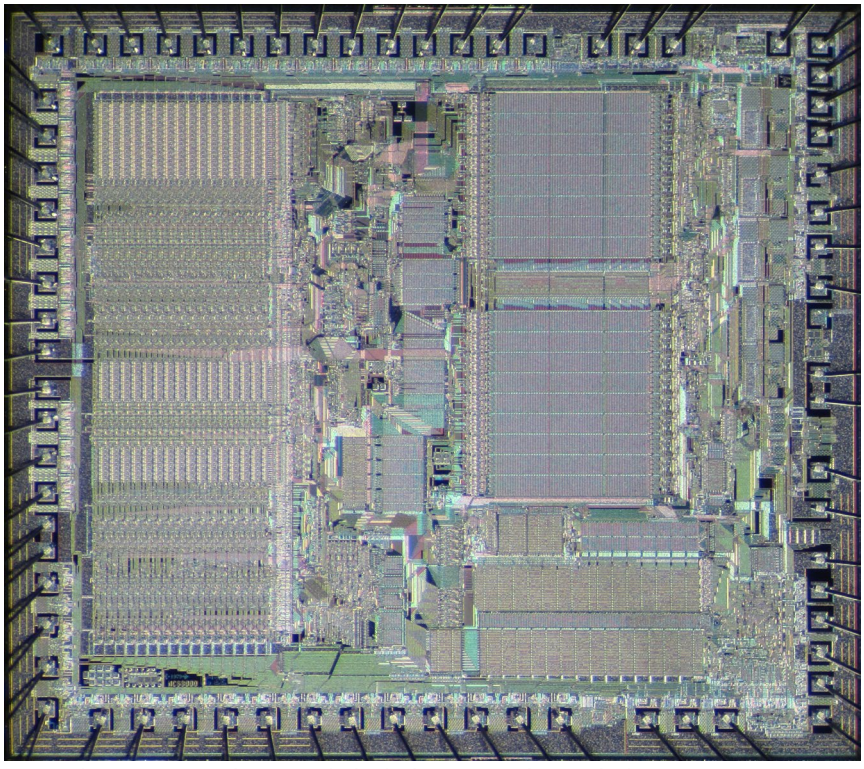
AMD Phenom X4 -suorittimen sisukset. Neljän symmetrisesti asetetun 64-bittisen suoritinytimen alasta valtaosa on välimuistia ja käskyjen purku- ja järjestelylogiikkaa.

rajoitteita: X86:ssa jommankumman operandin on aina oltava rekisteri tai välitön luku, eli esimerkiksi suoraa käskyä ”lisää muistipaikan 1 arvoon muistipaikan 2 arvo” ei ole. Muistiviittaukset voivat kuitenkin cisc-filosofian mukaisesti olla hyvinkin monimutkaisia. Esimerkiksi 32-bittinen X86-käsky `mov eax,[ebx+ecx*4+1256]` muodostaa muistiosoitteen summaamalla vakion ja kaksi rekisteriä, joista ECX:n bittejä vielä siirretään kaksi pykälää vasemmalle.

ARMin kaltaisissa risceissä useimmat käskyt voivat saada operandeikseen vain rekistereitä tai välittömiä lukuja. Muistinkäsittely on hoidettava omilla load-store-käskyillään (`ld`, `st`, `mov`), jotka eivät laske mitään.

ARMin ja 68K:n muistinkäsittelyä tehostavat osoitusmuodot, joissa rekisterin sisältöä kasvatetaan tai vähennetään samalla, kun sitä käytetään muistiosoituksessa. Tämä on kätevää muistialueita läpikäytäessä.

Muistipaikkoihin on monesti kätevää viitata suoran osoitteen sijaan pitämällä käskyn omaa sijaintia kiinnekohtana. 6502:n ja X86:n ehdollisilla hyppykäskyillä voi hypätä enintään 128 tavua eteen- tai taaksepäin, minkä ansiosta käskyn pituudeksi riittää kak-



Motorola 68000:n sisukset. Löydätkö laskenta- ja rekisteriyksikön?

```
lp:  movem (a0)+,(d1-d7)
     movem (d1-d7),-(a1)
     dbne d0,lp
```

```
lp:  subcc r2,r2,#1
     ldmia r0,(r3-r13)
     stmdb r1,(r3-r13)
     bne lp
```

Silmukka, joka kopioi muistialueen sisällön käänteisessä järjestyksessä toiselle muistialueelle rekisterijoukkokäskyjä käyttäen. Vasemmalla 68K, oikealla ARM.

si tavua. Ohjelmakoodia, joka ei käytä suoria osoitteita lainkaan, kutsutaan sijaintiriippumattomaksi (*position-independent*), koska sitä voi ajaa sellaisenaan mistä kohdasta muistia tahansa.

Laskenta on koneen leipätyö

Valtaosa suorittimista laskee oletusarvoisesti binäärimuotoisilla kokonaisluvuilla. 6502:ssa, 68K:ssa ja X86:ssa on tarjolla myös binäärikoodattu desimaali (BCD), jossa kutakin desimaalinumeroa 0-9 vastaa neljä bittiä. Liukulukuja taasen käsitellään omilla liukulukuyksiköillään, joilla on omat rekisterinsä ja käsittelykäskynsä.

Negatiiviset kokonaisluvut ilmaistaan lähes aina niinsanottuna kahden komplementtina, jossa etumerkki vaihdetaan kääntämällä bitit ympäri ja lisäämällä tulokseen yksi. Pelkkiä ykkösiä sisältävä luku on siis arvoltaan -1, aivan kuin kasettinauhurin laskuri, joka 000:sta taaksepäin kelatessa pyörähtää 999:ään. Saman bittijonon voi tulkita joko etumerkillisenä tai etumerkittömänä, ja erot tulevat esiin etenkin kertoessa, jakaessa ja vertaillessa.

Kokonaislukujen yhteen- ja vähennyslaskut (`add`, `sub`) ovat tarjolla joka konekielessä. Kerto- ja jakolaskut (`mul`, `div`) puuttuvat yleensä kasibittisisistä koneista, joissa ne on toteutettava aliohjelmilla tai taulukoilla. Risceissäkin on yleensä suoraan vain kertolasku.

Bittioperaatioita ovat paitsi bitittäiset loogiset operaatiot (`and`, `or`, `eor`/`xor`) ja bittisiirrot (*shifts*), joita on monta lajia. Bittioperaatioiden toiminta on esitetty oheisissa kaavioissa. ”Aritmeettisen” ja ”loogisen” bittisiirron merkitysero on se, että ”aritmeettisessa” luku oletetaan etumerkilliseksi, joten sen ylin bitti pidetään ennallaan.

ARMin erikoisuutena on se, ettei siinä ole lainkaan itsenäisiä käskyjä bittisiirroille, mutta bittisiirron voi yhdistää minkä tahansa laskentaoperaation toiseen lähdeoperandiin. Esimerkiksi `add r0,r1,r2 asr r3` vastaa C-kielen lauseketta `r0=r1+(r2>>r3)`.

Joskus laskutoimituksen tulos ei mahdu kohderekisteriin. Esimerkiksi kahden 8-bittisen luvun summa on 9-bittinen. Summan ylin bitti tallennetaan yleensä niinsanottuun muistinumerolippuun (C, *carry flag*). Muistinumeroa käytetään laskutoimitusten ketjuttamiseen: käskyt `adc`/`addx` ja `sbc`/`sbb`/`subx` ovat yhteen- ja vähennyslaskuja, jotka ottavat mukaan aiemmasta käskystä jääneen muistinumeron.

EX, EXG, XCHG	exchange	Vaihda rekisterien sisällöt keskenään.
LD	load	Lataa muistista.
MOV, MOVE	move	Kopioi dataa rekisteristä tai muistista rekisteriin tai muistiin.
POP, PL	pop, pull	Poimi pinon päällimmäinen arvo.
PUSH, PH	push	Lisää pinon päällimmäiseksi.
ST	store	Tallenna muistiin.

Datansiirto.

ADC, ADDX	add with carry/extend	Summaa muistinumeron kera.
ADD	add	Summaa.
DEC	decrement	Vähennä yhdellä.
DIV	divide	Jaa.
INC	increment	Kasvata yhdellä.
MUL	multiply	Kerro.
NEG	negate	Vaihda etumerkki.
SBB, SBC, SUBX	subtract with borrow/carry/extend	Vähennä muistinumeron kera.
SUB	subtract	Vähennä.

Peruslaskutoimitukset.

AND	and	Bitittäinen ja-operaatio.
ASL, SAL	arithmetic shift left	Siirrä bittejä oikealle jatkaen ylintä bittiä.
ASR, LSR, SHR	[arithmetic/logical] shift right	Siirrä bittejä vasemmalle.
EOR, XOR	exclusive or	Bitittäinen poissulkeva tai.
LSL, SHL	[logical] shift left	Siirrä bittejä oikealle jatkaen nollalla.
NOT	not	Käännä bitit päinvastaisiksi.
OR	or	Bitittäinen tai-operaatio.
ROL, RL, RCL	rotate [with carry] left	Pyöritä bittejä myötäpäivään [C-lipun kautta].
ROR, RR, RCR	rotate [with carry] right	Pyöritä bittejä vastapäivään [C-lipun kautta].

Bittioperaatiot.

Jossittelu

Korkean tason kielten `if`-lausetta vastaa konekielessä useimmiten ehdollinen hyppy. Esimerkiksi käsky nimeltä `beq`, `je` tai `jz` hyppää operandina annettuun muistiosoitteeseen, jos edellisen laskutoimituksen tulos oli nolla. Ennen hyppykäskyä käytetään usein vertailukäskyä `cmp`/`cp`, joka suorittaa vähennyslaskun tallentamatta tulosta mihinkään. Hyppykäskyt on yleensä nimetty vertailun näkökulmasta: jos vähennyslaskun tulos on nolla, niin luvut ovat yhtäsuuret (`e`/`eq`, *equal*).

Tieto laskutoimituksen tuloksesta tallentuu yleensä niinsanottuihin lippuihin, jotka ovat statusrekisterin bittejä. Aiemmin mainittu muistinumerolippu on yksi näistä. Ehdolliset hyppykäskyt tutkivat lippuja ja suorittavat hypyn, mikäli käskyn mukainen ehto täyttyy. Tyypillisiä lippuja ovat:

- Nollalippu (Z, zero), joka ilmaisee, oliko laskun tulos nolla.
- Etumerkkilippu (S, sign tai N, negative), joka vastaa rekisteriin mahtuneen tuloksen ylintä bittiä, joka on negatiivisilla luvuilla 1.
- Muistinumerolippu (C, carry), vastaa laskutoimituksesta yli jäänyttä bittiä.
- Ylivuotolippu (O tai V, overflow) asetetaan, kun muistinumerolippu ei riitä tuloksen jatkeeksi.

6502:ssa, X86:ssa ja 68K:ssa jokainen laskentakäsky vaikuttaa lippuihin. ARMissa lippuvaikutus ilmaistaan käskykohtaisesti käskysanan päätteellä `cc` (*condition codes*). ARMissa ei myöskään aina tarvita ehdollisia hyppyjä, sillä minkä tahansa käskyn suorituksen voi ehdollistaa. Esimerkiksi käsky `addeq` toimii kuten `add`, mutta se suoritetaan vain, jos nollalippu on ylhäällä.

Toisten tavarat pinoon talteen

Tavallinen ehdoton hyppykäsky on nimeltään esimerkiksi `jmp`, `bra` tai `b`, aliohjelmahyppy puolestaan `jsr`, `bsr`, `call` tai `bl`. Aliohjelmakutsut ottavat käskyosoittimen arvon talteen, jotta

BIT, BT, BTST, TEST	bit test	Vertaa yksittäisiä bittejä (ja-operaatio tallentamatta tulosta).
CLf	clear flag	Nollaa lippu (esim. C).
CMP, CP	compare	Vertaa (vähennä tallentamatta tulosta).
Scc, SETcc	set on condition	Aseta rekisterin arvo totuusarvon (esim. NE) mukaiseksi.
SEf, STf	set flag	Aseta lippu (esim. C).

Vertailu ja liput.

Bcc, Jcc	branch/jump on condition	Hyppää, jos ehto (esim. NE) täyttyy.
BL, BAL	branch and link	Hyppää aliohjelmaan, paluuosoite linkkirekisteriin.
DBcc, LOOP	decrement and branch, loop	Vähennä rekisterin arvoa ja hyppää jos ehto täyttyy.
JMP, JP, B, BRA	jump/branch	Hyppää muistipaikkaan.
JSR, JR, BSR	jump/branch to subroutine	Hyppää aliohjelmaan, paluuosoite pinoon.
RET, RTS	return from subroutine	Palaa aliohjelmasta pääohjelmaan.
SWI, INT, TRAP, BRK, SYSCALL	software interrupt, trap, break, system call	Suorita ohjelmallinen keskeytys.

Hyppykäskyt.

HLT	halt	Pysäytä suoritin (jää odottamaan keskeytystä).
NOP	no operation	Älä tee mitään.

Muita käskyjä.

CC, NC	no/clear carry	Muistinumero = 0
CS, C	carry set	Muistinumero = 1
EQ, E, Z	equal/zero	Luvut yhtäsuuret (nollalippu)
GT, G	greater [than]	Ensimmäinen > toinen
LT, L	less [than]	Ensimmäinen < toinen
NE, NZ	not equal/zero	Luvut erisuuret (nollalippu alhaalla)
NS, PL	no sign, plus	Tulos ei-negatiivinen (etumerkki = 0)
S, MI	sign, minus	Tulos negatiivinen (etumerkki = 1)
VC, NO	no/clear overlow	Ylivuotolippu alhaalla.
VS, O	overflow set	Ylivuotolippu ylhäällä.

Ehdot (käskyjen osana).

aliohjelmasta voidaan palata siihen kohtaan, mihin sitä kutsuttaessa jäätiin. Paluukäsky on tyypillisesti nimeltään `ret` tai `rts`.

Vanhemmat käskykannat tallentavat paluuosoitteen yleensä pinoksi kutsutulle muistialueelle. Riscit sen sijaan käyttävät rekisteriä, jonka aliohjelma pinoaa itse mikäli aikoo kutsua muita aliohjelmia. ARMin takaisinlinkittävä hyppykäsky on `bl` (*branch and link*). Linkkirekisteri on yleensä R14 ja käskyosoitin R15, joten aliohjelmasta palataan käskyllä `mov r15,r14`.

Pinoon tallennetaan muutakin kuin paluuosoitteita. Koska aliohjelmat käyttävät samoja rekistereitä kuin pääohjelmakin, on niiden arvoja usein tarpeellista raivata pinoon talteen. Lisäksi pinosta voidaan varata tilaa sellaisille paikallismuuttujille, jotka eivät mahdu rekistereihin. X86:ssa ja 6502:ssa on pino-osoittimeen sidotut push- ja pop/pull-käskyt, kun taas ARMissa ja 68K:ssa pinonkäsittelyyn käytetään tavallisia muistinkäsittelykäskyjä. ARMissa ja 68K:ssa on myös käskyt, joilla haluamansa rekisterijoukon voi tallentaa tai ladata yhdellä kertaa.

Isompien ohjelmakokonaisuuksien kasassa pitämiseksi on yleensä sovittu niinsanottuja kutsukäytäntöjä (*calling convention*). Nämä määrittelevät, miten pääohjelma ja aliohjelma välittävät parametrinsa ja paluuarvonsa, ja mitkä rekisterit aliohjelmalla on lupa sotkea.

Maailma on muistia

Suorittimen näkökulmasta koko ulkoinen maailma on muistia. Muisti jakautuu yleensä 8-bittisen tavun kokoisiin muistialkioihin, joista kullakin on oma numeerinen osoite.

Kun muistiin tallennetaan useista tavuista koostuvia lukuja, on tähän kaksi päätapaa. 68K käyttää laskevaa tavujärjestystä (*big-endian*), eli ensimmäiseen tavuun tulee luvun eniten merkitsevä osa. 6502 ja X86 puolestaan tallentavat alimmat bitit ensin, eli tavujärjestys on nouseva (*little-endian*). ARM toimii kummalla tavujärjestyksellä tahansa, joskin nouseva järjestys on nykyään yleisempi.

Yksinkertaisimmissa laitteissa fyysisillä RAM-, ROM- ja ohjauspiireillä on kiinteät alueet osoiteavaruudessa. VIC-20-ohjelmassa muistipaikkaan `$900F` kirjoittaminen vaikuttaa aina videopiirin värirekisteriin. Monimutkaisemmissa koneissa ohjelmalle näkyvää muistin rakennetta voi sen sijaan muuttaa.

Jos koneessa on enemmän muistia kuin osoiteavaruuteen mahtuu, esimerkiksi 6502-pohjaisessa koneessa yli 64 kilotavua, tarvitaan pankitusta (*banking*). Pankituksessa valitaan, mitkä osat kokonaismuistista näkyvät tietyillä muistiavaruuden alueilla. Nykykäyttöjärjestelmissä muistin näennäistä rakennetta sen sijaan muutellaan, jotta eri prosessit eivät pääsisi käsiksi niille kuulumattomille muistialueille. Samalla koodin oikeutta muuttaa suorittimen tilaa rajoitetaan siirtymällä sen suorittamisen ajaksi niinsanotusta valvojatilasta (*supervisor mode*) käyttäjätilaan (*user mode*).

Näennäismuistin ansiosta kaiken ohjelmalle näkyvän muistin ei tarvitse vastata fyysistä muistia. Jos osoiteavaruus on riittävän suuri, ohjelma voi pyytää käyttöjärjestelmää kuvaamaan vaikkapa koko kiintolevyn sisällön näennäismuistiin. Kun ohjelma yrittää lukea muistipaikkaa, joka ei ole fyysisessä muistissa, se aiheuttaa poikkeustilanteen, jonka käyttöjärjestelmä käsittelee hakemalla levyltä halutun kohdan fyysiseen muistiin. Ohjelman kannalta kone toimii siten, kuin koko levyn sisältö olisi jatkuvasti muistissa.

70-lukulaisilla suorittimilla muistin nopeus ei vielä muodosta pullonkaulaa. Esimerkiksi 6502:lla kannattaa käyttää nopeuskriittisessä koodissa muistinvaraisia taulukoita ja aukirullattuja silmukoita mikäli ne suinkin mahtuvat muistiin. Nykysuorittimilla laskuoperaatio saa sen sijaan olla jo todella monimutkainen, jotta sille kannattaisi laskea valmis taulukko. Aukirullauskaan ei enää kannata: sisäisten välimuistien ja älykkäiden liukuhihnojen ansiosta se todennäköisemmin hidastaa kuin nopeuttaa koodia.

```
        !to "skrolli.prg",cbm
        *=$0801     ; Ohjelman alkuosoite.

; Pakollinen BASIC-osuus 10 SYS2061 + loppunollat:
!byte $0b,$08,$0a,$00,$9e,$32,$30,$36,$31,0,0,0

        ldx #0      ; Laskuri (X) nollaan.

silmu   txa         ; X akkuun jotta saadaan
        and #15     ; laskettua X AND 15.
        tay         ; Tulos Y:hyn ja haetaan
        lda herja,y ; tavu osoitteesta herja+Y.

        sta $0400,x ; Kirjoitetaan se
        sta $0500,x ; näyttömuistin kullekin
        sta $0600,x ; 256-tavuiselle
        sta $0700,x ; lohkolle kohtaan X.

        inx         ; Kasvatetaan X:ää.
        bne silmu   ; Toistetaan kunnes pyörähti.

        rts         ; Palataan BASIC-tulkkiin.

herja   !scr "tilaa skrolli!! "
```

6502-esimerkki Commodore 64:lle. ACME-ristiinkääntäjän tuottama PRG-tiedosto käynnistyy suoraan vaikkapa VICE-emulaattorissa.

```
        bits 16         ; Nasm 16-bittiseen tilaan.
        org 0x100       ; COM-ohjelman alkuosoite.

        mov ax,0xb800   ; Näyttömuistin alkuosoite
        mov es,ax       ; .. segmenttirekisteriin.
        xor di,di       ; Kohdeosoitin nollaksi.

        mov ah,14       ; Väri AX:n ylempään tavuun.

silmu1  mov si,herja    ; Lähdeosoitin tekstin alkuun.
        mov cx,16       ; Silmukkalaskuriksi 16.

silmu0  lodsb           ; AL <- [DS*16+SI], SI kasvaa.
        stosw           ; AH*256+AL -> [ES*16+DI], DI +2.
        loop silmu0     ; CX vähenee, toistetaan kunnes 0.

        cmp di,80*25*2  ; Ollaanko käyty koko näyttö?
        jne silmu1      ; Jos ei, jatketaan toistoa.

        ret             ; Palataan komentotulkkiin.

herja   db "Tilaa Skrolli!! "
```

16-bittinen X86-esimerkki MS-DOSille. Ohjelma kääntyy NASMilla suoraan COM-tyyppiseksi ohjelmatiedostoksi.

Laitteita ohjaamaan

Tietokonelaitteistoon kuuluu oheispiirejä, joilla on omat ohjausrekisterinsä. 6502:ssa, 68K:ssa ja ARMissa nämä rekisterit näkyvät osana muistiavaruutta. X86 taasen käyttää tähän erillisiä I/O-portteja, joita käsitellään `in`- ja `out` -käskyillä.

Jotta suorittimen ei tarvitsisi jatkuvasti kysellä eli pollata laitteiden tiloja, on olemassa keskeytykset: laite voi lähettää suorittimelle keskeytyskutsun (IRQ, *interrupt request*), joka saa sen keskeyttämään senhetkisen toimintansa ja siirtymään käsittelyrutiiniin. Useimmat käyttöjärjestelmät suorittavat joitakin kymmeniä kertoja sekunnissa ajastinkeskeytyksen, jotta tietyt juoksevat asiat saadaan hoidettua.

Yksinkertaisimmillaan keskeytys ei juuri eroa aliohjelmakutsusta. Aliohjelman alkuosoite haetaan keskeytyksen tyypin ja mahdollisen numeron mukaan hyppytaulukosta. Nykykäyttöjärjestelmissä keskeytys myös siirtää suorittimen valvojatijaan. Vain valvojatilassa toimiva käyttöjärjestelmä voi käsitellä ulkoisia laitteita, ja sovellukset tekevät käyttöjärjestelmän apua tarvitessaan niinsanotun ohjelmallisen keskeytyksen (NMI, *non-maskable interrupt*).

Monta käskyä samaan aikaan

Yleisesti käytetyt käskykannat periytyvät vuosikymmenten takaa, mutta suorittimien toimintatavat ovat muuttuneet tänä aikana huomattavasti. Erityisesti rinnakkaisuutta on tullut lisää.

Perinteiset cisc-suorittimet ajavat vain yhtä käskyä kerrallaan. Käskyn suoritus jakautuu useisiin peräkkäisiin vaiheisiin, jotka on koodattu suorittimen sisäiseen mikrokooditaulukkoon. 6502-käskyn suoritus koostuu 2–8 vaiheesta, kun taas 8086:n jakolasku vie yli sata kellojaksoa. Ohjelmoija voi laskea koodinsa suoritusajan tällaisella suorittimella yksinkertaisesti summaamalla käskyjen viemät kellojaksot ja jakamalla tuloksen kellotaajuudella.

Riscin kantaviin ajatuksiin kuuluu, että yksinkertaisten käskyjen suoritusvaiheet voivat olla rinnakkaisia. Alkuperäisessä Archimedeen ARM-suorittimessa on kolmivaiheinen liukuhihna: suoritin tallentaa yhden laskutoimituksen tuloksen rekisteriin samaan aikaan, kun se laskee seuraavan käskyn laskutoimituksen ja lukee muistista sitä seuraavan käskyn.

Liukuhihnatekniikka tekee hypyistä suhteessa kalliita. Kun hyppykäsky toteutuu, joudutaan sen jälkeen tulevien käskyjen suoritusvaiheet hylkäämään. Ongelman ehkäisemiseen on useita tapoja. ARMin ehdollinen käskynsuoritus on yksi näistä: yhden tai parin käskyn suorittamatta jättäminen on edullisempaa kuin liukuhihnan tyhjennys. Edistyneempi tekniikka on haarautumisen ennakointi (*branch prediction*), jossa suoritin arvaa etukäteen, tuleeko hyppy toteutumaan, ja lataa käskyjä liukuhihnalle sen mukaan. Spekulatiivisessa suorituksessa puolestaan suoritetaan molemmat vaihtoehdot ja jätetään vain toisen vaikutukset voimaan.

Monissa suorittimissa on useita rinnakkaisia liukuhihnoja, eli ne suorittavat peräkkäisiä käskyjä aidosti samaan aikaan. Usein peräkkäiset käskyt ovat kuitenkin riippuvaisia toistensa tuloksista, joten ohjelmoijan tai suorittimen on hyvä järjestää käskyt siten, etteivät peräkkäiset käskyt käytä samoja rekistereitä. Suorittimen automatiikassa tällaisia tekniikoita ovat *out-of-order execution* ja *register renaming*.

X86 on tuottanut aivan omia haasteitaan suoritinsuunnittelijoille. 90-luvun puolestavälistä alkaen monimutkaiset X86-käskyt on yleensä pilkottu risc-tyylisiksi mikrokäskyiksi, joihin sitten sovelletaan edellämainittuja tekniikoita.

Erikoiskäskyjä erikoistehtäviin

Vaikka peruskäskykannoilla pystyykin jotenkuten tekemään kaiken, on niille

```
# Määritellään linkkeriä varten symboli _start,
# joka osoittaa ohjelman ajon alkukohtaan.

.globl _start
_start:

# Alustetaan silmukkalaskuri.

        movq $1024,%rbp     mov r8,#1024

# Suoritetaan järjestelmäkutsu write(1,herja,15),
# missä 1 on standardiulostulo ja 15 pituus.
# Write-kutsun numero on 64-bittisessä Linuxissa 1
# ja 32-bittisessä 4.

silmu:  movq $1,%rax        mov r7,#4
        movq %rax,%rdi      mov r0,#1
        movq $herja,%rsi    adr r1,herja
        movq $15,%rdx       mov r2,#15
        syscall             swi 0

# Vähennetään laskuria, hypätään jos ei nolla.

        decq %rbp           subcc r8,r8,#1
        jnz silmu           bne silmu

# Suoritetaan järjestelmäkutsu exit(0)

        movq $4,%rax        mov r7,#1
        xorq %rdi,%rdi      mov r0,#0
        syscall             swi 0

# Tulostettava merkkijono:

herja:  .string "Tilaa Skrolli! "
```

Kernel-kutsuja käyttävä Linux-esimerkki 64-bittiselle X86:lle (vasemmanpuoleiset käskyt) ja 32-bittiselle ARMille (oikeanpuoleiset). Ohjelma kääntyy kohdekoneella käskyllä gcc -nostdlib ohjelma.s -o ohjelma tai erikseen as-assembleria ja ld-linkkeriä kutsumalla.

monesti olemassa erityisiä laajennoksia, jotka nopeuttavat tietyntyyppisten tehtävien suorittamista.

Liukulukumatematiikkaa on vanhastaan tarvittu etenkin tieteellisessä laskennassa. Ideana on, että luvut esitetään käyttäen kantalukua ja ”nollamäärää” (eli mantissaa ja eksponenttia), jolloin niiden arvoalue on huomattavasti laajempi kuin kokonaisluvuilla. PC-suorittimiin alkoi tulla integroituja liukulukuyksiköitä 486-aikakaudella, mutta pelikonsoli- ja mobiilimaailmassa liukulukulaitteisto ei ole ollut vielä 2000-luvullakaan mikään itsestäänselvyys.

Kuva- ja äänidatan käsittelyn nopeuttamiseen on käytetty digitaalisia signaaliprosessoreita (DSP). 90-luvulla DSP-tyyppisiä SIMD-käskystöjä (*single instruction, multiple data*) alkoi ilmestyä myös perussuorittimiin. Esimerkkejä SIMD-laajennoksista ovat X86:n MMX ja SSE sekä ARMin NEON.

SIMD-käskyt operoivat nimensä mukaisesti usealla erillisellä data-alkiolla rinnakkain. Esimerkiksi MMX-käsky `paddb mm0,mm1` tulkitsee 64-bittisten multimediarekisterien MM0 ja MM1 arvot erillisten 8-bittisten tavujen riveinä laskiessaan ne yhteen. Käskyjä on myös esimerkiksi data-alkioiden uudelleenjärjestelyyn.

SSE:n ja NEONin rekisterit ovat 128-bittiset, ja alkiot voivat olla myös liukulukuja. SSE tukee liukuluvuille myös neliöjuuren kaltaisia monimutkaisia toimituksia, ja nyky-X86:ssa se onkin syrjäyttänyt vanhat X87-käskyt.

Etenkin mobiilimaailmassa yksiin kuoriin on integroitu valtava määrä erikoistunutta laskentalogiikkaa. Esimerkiksi Qualcommin Snapdragon 810 -piiri sisältää kahdeksan 64-bittistä ARM-suoritinydintä, joista kullakin on kolme erillistä liukuhihnaa ja NEON- ja liukulukulaajennokset. Lisäksi piirillä on 288-laskentayksikköinen grafiikkasuoritin, 32-bittinen DSP sekä eri radioprotokolliin erikoistuneita ohjainpiirejä. Samanaikaisia laskutoimituksia saattaa siis tapahtua taskussa enemmän kuin entisajan supertietokoneessa.

Ja sitten räpeltämään

Luontevimmat ja usein palkitsevimmat kohteet konekielitason askarteluun löytyvät yksinkertaisesta tietotekniikasta – esimerkiksi vanhoista kotimikroista, sulautetuista järjestelmistä ja Arduinon kaltaisista elektroniikka-alustoista. Laitteen toimintaan on näissä mahdollista sukeltaa yksittäisten bitinliikkeiden ja kellojaksojen tarkkuudella, ja suorittimen erityispiirteitä voi hyödyntää tavoilla, jotka katoavat korkean tason kielillä näkymättömiin. Pieniä laitteita konekieliohjelmoidaan useimmiten ristiinkääntäjillä (*cross-assembler*), ja apuna voi usein käyttää myös emulaattoreita. Valmiita oppaita kannattaa etsiä kohdekoneen mukaan.

Isommilla laitteilla helpoin tie konekielen pariin löytyy korkean tason kielten kääntäjistä: esimerkiksi GCC kääntää `-S`-vivulla lähdekooditiedoston assemblyksi, jota voi tutkia ja muokata itse. Kääntäjät tukevat myös inline-assemblyä, eli assembly-osien upottamista korkean tason kielen sekaan. Koodin nopeuttamista ei kannattane käyttää motivaattorina nykysuorittimien konekieliohjelmoinnin opetteluun – sen sijaan kannattaa kokeilla esimerkiksi mahdollisimman pienten ohjelmien kirjoittamista.

Assembly-kääntäjien ohella voi käyttää myös suorempia työkaluja. Konekielimonitorit ja debuggerit on tarkoitettu muistin ja muistinvaraisen ohjelman käsittelyyn lennossa. Heksaeditoreilla voi puolestaan tutkia ja muokata ohjelmatiedostoja suoraan, ja monet niistä osaavat esittää tiedoston sisällön myös assemblynä.

Tämä artikkeli oli tiivis pintakatsaus konekielen olemukseen. Sen tiedoilla voi yrittää tutkia assembly-koodia, mutta syvempään tutustumiseen on hyvä olla kattavat käskykanta- tai suoritinkohtaiset dokumentit. Tämän jälkeen konekieliohjelmoinnin saloihin pääseekin parhaiten sisään valitsemalla itselleen jonkin mieluisen projektin ja ohjelmoimalla itse.