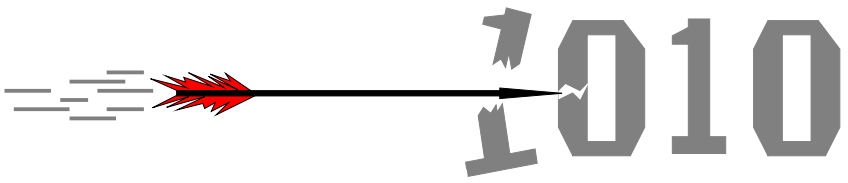
Tietotekniikan perusteet



Pasi Koikkalainen
Data-analyysin laboratorio

Jyväskylän yliopisto

Pekka Orponen
Tietojenkäsittelyteorian laboratorio

Teknillinen korkeakoulu

kevät 2002

Lukijalle

Tämä moniste on syntynyt Jyväskylän yliopistossa luennoitavan tietotekniikan approbatur-kurssin "Tietotekniikan perusteet" tarpeisiin. Tämän kolmen opintoviikon laajuisen johdantokurssin tavoitteena on perehdyttää oppiaineen ensimmäisen vuoden opiskelijat niihin yleisiin periaatteisiin ja tekniikoihin, joille nykyaikainen tietotekniikka rakentuu, sekä toimia ensimmäisenä johdatuksena tietojenkäsittelytieteellisiin kysymyksenasetteluihin.

Kurssin laajuus sen ensimmäisillä luentokerroilla vuonna 1998 oli 24 luentotuntia ja 24 niitä tukevaa harjoitustuntia. Kurssille osallistujat olivat joko suorittaneet ensimmäisen ohjelmointikurssin edeltävällä lukukaudella, tai suorittivat sitä samaan aikaan tämän kurssin kanssa. Vaikka ohjelmointitaito ei ole aivan välttämätön edellytys tämän monisteen asioiden ymmärtämiselle, se helpottanee monin paikoin suhteellisen tiiviin tekstin avautumista.

Monisteen ensimmäinen osa käsittelee nykyaikaisen tietokoneen rakennetta ja toimintaa tiedonesitysten ja logiikkapiirien suunnittelun tasolta von Neumann -tyyppisen tietokoneen arkkitehtuuriin ja konekäskyjen toteutukseen saakka. Monisteen toinen osa jatkaa tästä ohjelmistotekniikan ja tietojenkäsittelytieteen suuntaan yleisesityksillä ohjelmointikielistä, käyttöjärjestelmistä, tietokannoista, algoritmianalyysista ja tekoälystä. Kurssin tavoitteiden ja aikarajoitteiden mukaisesti kutakin aihepiiriä on tässä tarkasteltu vain noin yhden luentokerran laajuudessa: varsinaisen tietotekniikan alan ammattitaitoon johtavan käsittelynsä asiat saavat vasta opintojen myöhemmissä vaiheissa.

Jyväskylässä, 8. huhtikuuta 1999

Pasi Koikkalainen *Pekka Orponen*

Monisteen toiseen painokseen olemme korjanneet edellisessä painoksessa olleita virheitä. Lisäksi olemme tehneet lisäyksiä, mistä merkittävin on tietoliikenneosuuden laajentaminen omaksi luvukseen. Loppuun olemme lisänneet myös joukon kirjallisuuslähteitä, joiden kautta aiheeseen voi tutustua syvällisemmin.

Jyväskylässä, 6. huhtikuuta 2000

Pasi Koikkalainen *Pekka Orponen*

Kolmannessa painoksessa on edelleen korjailtu virheitä ja tehty pieniä täydennyksiä.

Jyväskylässä, 17. huhtikuuta 2001

Pasi Koikkalainen *Pekka Orponen*

Neljäs painos perustuu FT Timo Männikön työhön. Hän on käynyt monisteen läpi kiitettävällä tarkkuudella löytäen huomattavan määrän kirjoitusvirheitä ja muita puutteellisuuksia, jotka on nyt korjattu.

Kiitokset Timolle.

Jyväskylässä, 2. huhtikuuta 2002

Pasi Koikkalainen

Sisältö

Osa I

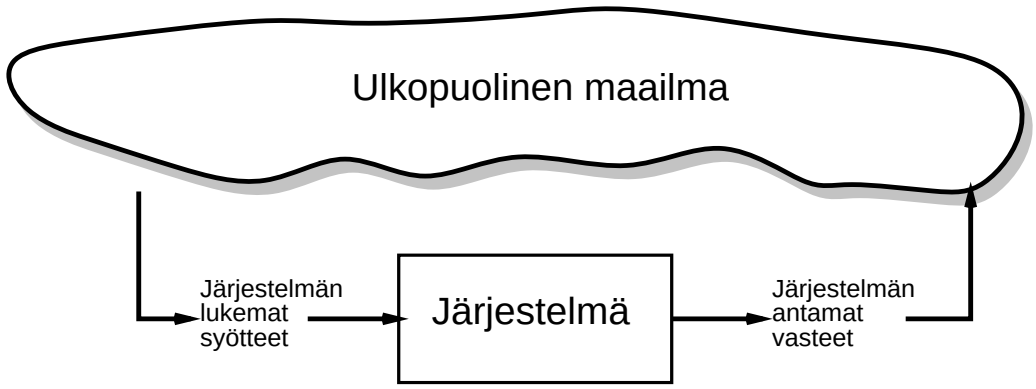
Tietokone

Luku 1

Digitaalinen järjestelmä

1.1 Järjestelmä

Järjestelmän (systeemin) käsite on varsin hyödyllinen pyrkiessämme ymmärtämään tietokoneen toimintaa. Se soveltuu kuvaamaan ihmisen rakentamia laitteita sekä yhteiskunnan tai luonnon osa-kokonaisuuksia. Järjestelmän perusajatuksen voi ymmärtää kuvan 1.1 avulla. Järjestelmä on muusta maailmasta erotettu osa, joka voi olla vuorovaikutuksessa vain määrättyjen syötteiden ja vasteiden avulla.



Kuva 1.1: *Järjestelmä on muusta maailmasta irrallinen osakokonaisuus, joka kommunikoi vain syötteiden ja vasteiden avulla.*

Myös ihmisen voi ymmärtää järjestelmäksi, joka aisteillaan vastaanottaa tietoa ym-

päristöstä ja tuottaa vasteinaan lihasten liikkeitä, joita muu maailma voi havainnoida (myös ääni on lihasten liikettä). Ihminen siten ymmärtää muun maailman tilaa ainostaan havaintojensa kautta ja voi tehdä oman tilansa ymmärrettäväksi omilla vasteillaan.

Tietokoneen näkökulmasta ihmisellä on toinenkin keskeinen järjestelmien ominaisuus, muisti, joka mahdollistaa monimutkaiset vuorovaikutussuhteet syötteiden ja vasteiden välillä. Näiden vuorovaikutussuhteiden ansiosta yhtä syötettä kohden on mahdollista tuottaa useita vasteita, syötteen ja vasteen välillä voi olla viivettä, vasteita voi tuottaa ilman syötettä, tai vasteet voivat jäädä kokonaan tulematta.

1.2 Syöte-vaste-käyttäytyminen ja aakkostot

Tietokone on pohjimmiltaan digitaalinen järjestelmä, joka muiden järjestelmien tavoin kommunikoi ulkopuolisen maailman kanssa lukemalla syötteitä ja palauttamalla vasteita. Tietokoneessa syötteiden muoto ja määrä ovat rajoitettuja ja vuorovaikutussuhteet tarkasti määrättyjä. Syötteiden ja vasteiden vuorovaikutussuhteen määrää järjestelmän sisäinen toimintasekvenssi, joka voidaan kuvata algoritmina eli joukkona toimintaohjeita.

Näin on myös muissa fyysisissä järjestelmässä. Ulkopuolinen havainnoitsija voi siis nähdä vain algoritmin toteutuksen järjestelmälle kohdistettuina **syötesignaaleina** (input) ja järjestelmän niihin antamina **vastesignaaleina** (output).

Abstraktilla tasolla voidaan mikä tahansa tehtävä spesifioida (määrätä) alkutilaa kuvaavina syötteinä ja toivottua lopputulosta kuvaavina vasteina. Se miten tämä saadaan aikaiseksi vaatii avukseen toimintaohjeen eli algoritmin, jonka avulla ilmaistaan miten syötettä käsitellään, jotta haluttuun lopputulokseen (vasteeseen) voidaan loppujen lopuksi päätyä. Esimerkkinä voi ajatella kahden luvun yhteenlaskua: syöte on nämä kaksi lukua ja vaste on näiden lukujen summa.

Tietotekniikassa on syöte-vaste-käyttäytymisen ymmärtäminen keskeistä. Mahdollisten erilaisten syötteiden määrä on rajoitettu (äärellinen), samoin kuin mahdollisten vasteiden määräkin. **Digitaalinen järjestelmä** on siis järjestelmä, jonka syöte-vaste-käyttäytyminen koostuu rajoitetusta määrästä syötesignaaleja ja vastesignaaleja. Jos käytössämme on esimerkiksi N kappaletta erilaisia syötteitä ja M kappaletta erilaisia vasteita, voidaan järjestelmä kuvata luettelemalla nämä vaihtoehdot joukkoina:

syötteiden joukko	$\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_N\}$, ja
vasteiden joukko	$\Omega = \{\omega_1, \omega_2, \ldots, \omega_M\}$.

Joukkoja Σ ja Ω kutsutaan **syöte-aakkostoksi** ja **vaste-aakkostoksi**. Aakkoston alkioita kutsutaan sen kirjaimiksi ja näistä järjestettyä kirjainjonoa **sanaksi**. Digitaalista järjestelmää voidaan toisissa yhteyksissä kutsua myös **diskreetiksi järjestelmäksi** tai **loogiseksi järjestelmäksi**.

Tietotekniikassa on pohjimmiltaan kyse muunnoksista äärellisestä määrästä lähtötietoja äärelliseen määrään lopputietoja.

1.3 Koodaukset

Käytännössä ei ole järkevää ajatella jokaista tietoalkiota omana symbolinaan. Näin on etenkin lukujen kanssa, joiden esittämisessä erilaisten symbolien määrä kasvaisi hyvin nopeasti järjettömän suureksi, vaikka lukujen määrää rajoitettaisiinkin käytännön ja tarpeen kannalta mahdollisimman pieneksi.

Tämän on havainnut jo muinainen ihminen kehittäessään mm. kymmenluku- ja kaksitoistalukujärjestelmät. Näissä isokin luku voidaan koodata pienemmällä joukolla symboleja, esimerkiksi symboleilla (merkeillä) "0", "1", "2", ... ,"9". Varsinaisia symboleita on siis vain kymmenen erilaista, mutta niitä yhdistelemällä voidaan tuottaa rajaton määrä koodattuja esityksiä, sanoja.

Yleisesti koodaus voidaan esittää siten, että aakkoston, joka sisältää L symbolia, voi koodata n erilaisen alisymbolin järjestetyllä joukolla (sanalla), jonka pituus (sananpituus) on p. Siis osa-alkioita tarvitaan vain p kappaletta, missä p on pienin kokonaisluku siten, että

$$p \geq \log_n L. \tag{1.1}$$

Esimerkiksi kymmenlukujärjestelmässä tuhat lukua: “0”, “1”, ..., “999” voidaan esittää kolmella symbolilla $\log_{10} 1000 = 3$. Se mistä luvusta esitys alkaa ja mihin se päättyy on valintakysymys. Yleensä valitaan kuten edellä, ensimmäiseksi luvuksi nolla, jolloin viimeinen luku on lukujen määrä vähennettynä yhdellä.

Negatiiviset luvut voidaan esittää, joko käyttämällä ylimääräistä “-” symbolia, tai jakamalla lukualue kahteen osaan, positiivisiin lukuihin ja negatiivisiin lukuihin, so-

pivasta kohdasta koodattua esitystä. Tietokoneessa käytetään jälkimmäistä menetelmää, mikä 2-kantaisessa lukuesityksessä vaatii yhden ylimäärisen bitin etumerkille. Reaaliluvuissa taas on tapana esittää luvun kokonaisosa ja desimaaliosa omana koodattuna lukunaan, mitä voi edelleen kehittää suurille luvuille siten, että luvun magnitudi (eksponentti) esitetään erikseen. Tässä yhteydessä tyydymme yksinkertaisuuden vuoksi kokonaislukujen esityksiin.

Taulukossa 1.1 on eräitä yleisimpiä koodauksia kokonaisluvuille, perustuen 10, 2, 16 ja 8-kantaisiin koodauksiin kokonaisluvuille.

10-kanta (desimaali)	**2-kanta (binääri)**	**16-kanta (heksadesimaali)**	**8-kanta (oktaali)**
000	00000	000	000
001	00001	001	001
002	00010	002	002
003	00011	003	003
004	00100	004	004
005	00101	005	005
006	00110	006	006
007	00111	007	007
008	01000	008	010
009	01001	009	011
010	01010	00A	012
011	01011	00B	013
012	01100	00C	014
013	01101	00D	015
014	01110	00E	016
015	01111	00F	017
016	10000	010	020
017	10001	011	021
018	10010	012	022
019	10011	013	023
020	10100	014	024
...	.....	...	...

Taulukko 1.1: Kokonaisluvut $0, \ldots, 20$ esitettynä erikantaisissa järjestelmissä.

Muunnokset eri lukujärjestelmien välillä voidaan tehdä merkeittäin vanhan järjestelmän symboleista uuteen järjestelmään. Esimerkiksi muunnos A-kantaisesta luvusta $Y = (y_{n-1}y_{n-2}\ldots y_0)$ uuteen B-kantaiseen lukuun $X = (x_{m-1}x_{m-2}\ldots x_0)$ voidaan tehdä käyttäen seuraavaa muunnoskaavaa:

$$X = \sum_{i=0}^{n-1} (y_i)_{(B)} A^i_{(B)}, \tag{1.2}$$

missä $A_{(B)}$ on alkuperäisen luvun kanta uuden B-järjestelmän lukuna, i on alkuperäisen numeron (merkin) indeksi (vähiten merkitsevä on 0, toinen 1, jne.) ja $(y_i)_{(B)}$ on alkuperäisen luvun i:s numero (merkki) uuden B-järjestelmän lukuna.

Esimerkiksi heksadesimaaliluku $A09$ on kymmenkantaisessa järjestelmässä merkeittäin esitettynä $\underbrace{10_{(10)}}_{i=2}, \underbrace{0_{(10)}}_{i=1}, \underbrace{9_{(10)}}_{i=0}$, eli 10-kantaisena lukuna se on

$$\begin{aligned} X &= 10 \times 16^2 + 0 \times 16^1 + 9 \times 16^0 \\ &= 10 \times 256 + 0 \times 16 + 9 \times 1 = 2569. \end{aligned} \tag{1.3}$$

Toisinpäin laskutoimitus saattaa olla hieman hankalampi ymmärtää, sillä se vaatii heksadesimaalilukujen yhteenlaskua (esim. 4+8=C) ja potenssiinkorotusta ($10_{(10)} = A_{(16)}$, $10^2_{(10)} = A^2_{(16)} = 64_{(16)}$ ja $10^3_{(10)} = A^3_{(16)} = 3E8_{(16)}$)

$$\begin{aligned} Y &= 2 \times A^3 + 5 \times A^2 + 6 \times A^1 + 9 \times A^0 \\ &= 2 \times 3E8 + 5 \times 64 + 6 \times A + 9 \times 1 \\ &= 7D0 + 1F4 + 3C + 9 \\ &= A09 \end{aligned} \tag{1.4}$$

Taulukossa 1.2 on muutamia lisäesimerkkejä koodausten tekemisestä yksityiskohtaisemmin merkinnöin.

Tietokoneen käyttämä lukuesitys on 2-kantainen lukujärjestelmä, joka yleisesti tunnetaan myös digitaalisen järjestelmän nimellä. Tämän etuna on lähinnä laitteistotekninen toteutus, sillä 2-tilaisia sähköisiä komponentteja on muita vaihtoehtoja helpompi toteuttaa. Tätä käsitellään tarkemmin luvussa 3.1.

Esimerkki 1. Muunnos 10-kantaisen lukujärjestelmän luvusta 103 2-kantaiseen:

$$
\begin{aligned}
&103_{(10)} \\
&= 1_{(10)} \times 100_{(10)} + 0_{(10)} \times 10_{(10)} + 3_{(10)} \times 1_{(10)} \\
&= 1_{(2)} \times 1100100_{(2)} + 0_{(2)} \times 1010_{(2)} + 11_{(2)} \times 1_{(2)} \\
&= 1100111_{(2)}.
\end{aligned}
$$

Esimerkki 2. Muunnos 16-kantaisen lukujärjestelmän luvusta 31A 2-kantaiseen:

$$
\begin{aligned}
&31A_{(16)} \\
&= 3_{(16)} \times 100_{(16)} + 1_{(16)} \times 10_{(16)} + A_{(16)} \times 1_{(16)} \\
&= 11_{(2)} \times 100000000_{(2)} + 1_{(2)} \times 10000_{(2)} + 1010_{(2)} \times 1_{(2)} \\
&= 1100011010_{(2)}.
\end{aligned}
$$

Esimerkki 3. Muunnos 8-kantaisen lukujärjestelmän luvusta 77 10-kantaiseen:

$$
\begin{aligned}
&77_{(8)} \\
&= 7_{(8)} \times 10_{(8)} + 7_{(8)} \times 1_{(8)} \\
&= 7_{(10)} \times 8_{(10)} + 7_{(10)} \times 1_{(10)} \\
&= 63_{(10)}.
\end{aligned}
$$

Esimerkki 4. Muunnos 2-kantaisen lukujärjestelmän luvusta 10101 10-kantaiseen:

$$
\begin{aligned}
&10101_{(2)} \\
&= 1_{(2)} \times 10000_{(2)} + 0_{(2)} \times 1000_{(2)} + 1_{(2)} \times 100_{(2)} + 0_{(2)} \times 10_{(2)} + 1_{(2)} \times 1_{(2)} \\
&= 1_{(10)} \times 16_{(10)} + 0_{(10)} \times 8_{(10)} + 1_{(10)} \times 4_{(10)} + 0_{(10)} \times 2_{(10)} + 1_{(10)} \times 1_{(10)} \\
&= 21_{(10)}.
\end{aligned}
$$

Taulukko 1.2: Esimerkkejä koodausten tekemisestä erikantaisten lukujärjestelmien välillä.

Luku 2

Operaatiot digitaalisessa järjestelmässä

Symbolien lisäksi on pystyttävä määräämään myös järjestelmän toiminnot. Yleisellä tasolla toiminto on kuvaus syötesymboleista vastesymboleihin.

Esimerkiksi kahden luvun a ja b yhteenlasku $c = a + b$ voitaisiin esittää hieman erikoisen tuntuisena symbolimuunnoksena jokaisesta mahdollisesta syöteparista (a, b) jokaiseen mahdolliseen vastaukseen c. On siis kyse kuvauksesta

$$f : \Sigma \to \Omega$$

mahdollisten syötteiden (summattavat numeroparit) joukosta Σ mahdollisten vasteiden (lukujen summat) joukkoon Ω. Digitaalisessa järjestelmässä nämä kuvaukset määrätään tarkalleen kaikkien mahdollisten symbolien välille.

Edelläkerrotun pohjalta on mahdollista, joskin yleisesti järjetöntä, määrätä lukujen yhteenlasku taulukoituna kullekin symbolien yhdistelmälle erikseen:

```
Jos (a=0 ja b=0) niin c=0
Jos (a=0 ja b=1) niin c=1
Jos (a=1 ja b=0) niin c=1
Jos (a=1 ja b=1) niin c=2
jne.
```

Tarvitaankin tapa esittää kuvaukset yleisemmin, suppealla joukolla perusoperaatioita ja suhteellisen pienellä määrällä symboleja.

2.1 Loogiset funktiot ja totuustaulut

Osoittautuu, että koodauksen käyttö symbolien esittämisessä yksinkertaistaa myös operaatioita, sillä nämä voidaan tehdä usein varsin kätevästi merkeittäin (tai numeroittain, jos kyseessä on laskuoperaatio).

Esimerkiksi kymmenlukujärjestelmän yhteenlaskussa $302 + 647$ voitaisiin laskea ensin $2 + 7$, tämän jälkeen $0 + 4$, ja lopuksi $3 + 6$, jolloin yhteenlaskun perusoperaatio tarvitsee määritellä vain kahden 10-arvoisen symbolin välille. Tämän jälkeen perusoperaatiota käytetään kullekin merkkiparille erikseen[1].

Kun syötteet ja vasteet esitetään merkeittäin tai numeroittain 10-kantaisessa muodossa, on kyseessä nk. *looginen funktio* muotoa

$$f : \{0, 1, 2, \ldots, 9\}^p \rightarrow \{0, 1, 2, \ldots, 9\}. \tag{2.1}$$

Siis p:stä 10-arvoisesta symbolista päädytään yhteen 10-arvoiseen symboliin, missä p on syötteiden määrä. Nyt n-arvoisen operaation toiminnan täydellinen taulukointi vaatii n^p riviä. Siis esimerkkimme tapauksessa taulukointiin tarvittaisiin 10^2 riviä, koska p on kaksi (symbolit a ja b).

Tämä ei kuitenkaan ole yksinkertaisin tapa, sillä digitaalisessa, 2-kantaisessa järjestelmässä kuvauksia tarvitaan vain 2^p. Tämä kuvaus on muotoa

$$f : \{0, 1\}^p \rightarrow \{0, 1\}. \tag{2.2}$$

Kun lisäksi huomataan, että kaikki loogiset funktiot voidaan koostaa kahden merkin funktioista ($p = 2$), niin taulukon pienuuden vuoksi toisistaan riippumattomia operaatioita on vain muutama. Kaikki monimutkaisemmat ja useampia syötteitä käyttävät operaatiot voidaan esittää näiden perusoperaatioiden (JA, TAI, EI) avulla.

Tämä joukko yksinkertaisia laskusääntöjä on nimeltään Boolen algebra. Käytännössä yksinkertaisuudesta on se hyöty, että operaatiot on mahdollista toteuttaa muutaman sähköisen komponentin avulla. Loogisella tasolla Boolen algebran operaatiot (JA, TAI, EI) voidaan esittää myös taulukon 2.1 muodossa.

[1] Lisähuomautuksena edelliseen mainittakoon, että kaikki operaatiot ovat todellakin määritettävissä merkeittäin, mutta se ei ole välttämättä aina mielekästä. Tämän seikan tarkempi tarkastelu joudutaan tässä yhteydessä sivuuttamaan.

x1	**x2**	**x1** AND **x2**
0	0	0
1	0	0
0	1	0
1	1	1

x1	**x2**	**x1** OR **x2**
0	0	0
1	0	1
0	1	1
1	1	1

x	NOT **x**
1	0
0	1

Taulukko 2.1: AND, OR ja NOT loogisten operaatioiden totuustaulukot.

Kaksiarvoinen logiikka on siis varsin perusteltu vaihtoehto tietokoneiden toimintojen kuvaamiseen. Muitakin vaihtoehtoja on toki harkittu ja eräissä toiminnoissa sovelletaankin 3-arvoista logiikkaa, jonka loogiset funktiot ovat muotoa

$$f : \{0, 1, 2\}^p \rightarrow \{0, 1, 2\}. \tag{2.3}$$

Käytännön sovelluksissa kolmiarvoisen logiikan perusteena on käytetty tarvetta epätietoisuuden esittämiseen (0=epätosi, 1=epävarma, 2=tosi). Toisaalta tämä saadaan toteutettua myös sopivaa koodausta käyttämällä kaksiarvoisella logiikalla.

2.2 Boolen algebra

Tässä yhteydessä totuustaulujen käyttö riittää tietokoneen toiminnan ymmärtämiseen. Yleissivistyksen vuoksi lienee paikallaan ymmärtää myös hieman Boolen algebraa, joka muodostaa pohjan laajemmalle laskennan teorialle.

Yksinkertaistettuna se on binääristen (kaksiargumenttisten) operaatioiden $\wedge$ (konjunktio, AND), $\vee$ (disjunktio, OR), sekä yksiargumenttisen operaation $\overline{x}$ (NOT x) kautta rakennettu laskentamalli, joka yleensä käsittelee binäärilukuja, mutta yleistyy muihinkin esityksiin. Sen avulla voidaan esittää mikä tahansa digitaalisen systeemin syöte-vaste-käyttäytyminen rakennettuna muutaman perustavaa laatua olevan lainalaisuuden, aksiooman, varaan. Aksioomista on johdettavissa kaikki muut mahdolliset lainalaisuudet, jolloin voidaan välttää sinänsä havainnollinen, mutta raskas totuustaulujen käyttö.

Jos käytettävissä on kolme lukua a, b ja c, voidaan Boolen algebran lait esittää seuraavasti:

$A1$: $a \wedge (b \vee c) = a \wedge b \vee a \wedge c$, $A2$: $a \vee b \wedge c = (a \vee b) \wedge (a \vee c)$,
$A3$: $a \vee 0 = a$, $A4$: $a \wedge 0 = 0$,
$A5$: $a \vee 1 = 1$, $A6$: $a \wedge 1 = a$,
$A7$: $\overline{1} = 0$, $A8$: $a \vee \overline{a} = 1$,
$A9$: $a \wedge \overline{a} = 0$, $A10$: $\overline{(\overline{a})} = a$,

missä se mikä pätee $\wedge$:lle pätee myös $\vee$:lle, kun samalla vaihdetaan 0 ja 1 keskenään. Tämän takaavat nk. De Morganin lait, mitkä ovat hyödyllisiä laskukaavoja monissa tilanteissa. Formaalimmin lait ovat

$$D1: \overline{a \vee b} = \overline{a} \wedge \overline{b}, \quad D2: \overline{a \wedge b} = \overline{a} \vee \overline{b},$$

jotka voidaan todistaa oikeiksi esimerkiksi totuustaulujen avulla.

2.3 Carry-yhteenlaskin

Jatkamme esitystä yhteenlaskuesimerkillä, jonka avulla voidaan ymmärtää monia muitakin tietokoneen perustoimintoja eli käskyjä. Tarkoituksena on tehdä kahden luvun yhteenlasku merkeittäin, esimerkiksi bitti kerrallaan, jos aakkoston kirjaimisto on 2-kantainen.

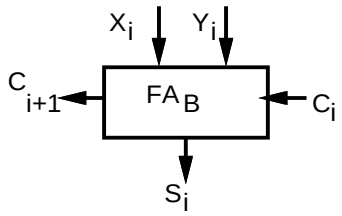
Ymmärtääksemme lukujen yhteenlaskua mielivaltaisessa lukujärjestelmässä ja tietokoneen piiritasolla katsomme miten kaksi lukua X ja Y voidaan laskea yhteen käyttäen algoritmista lähestymistapaa. Molemmat luvut koostuvat B-kantaisesta järjestelmästä ja lukujen yksittäisiä numeroita (merkkejä) indeksoidaan kuten edellä indeksillä i. Esimerkiksi 10-järjestelmän luvussa 951 merkki $x_0 = 1$, $x_1 = 5$ ja $x_2 = 9$. Yhteenlaskualgoritmi $S = Y + X$ voidaan nyt esittää merkeittäin

$$s_i = [\![x_i + y_i + c_i]\!]_B, \tag{2.4}$$

missä $[\![.]\!]$ tarkoittaa, että yhteenlaskun tuloksena syntyvästä luvusta otetaan vain viimeinen (vähiten merkitsevä) merkki (esim. $[\![14]\!]_{10} = 4$), ja missä c_i on nk. ylivuotomerkki (carry-merkki). Se on määritelty edellisen merkin $(i-1)$ laskutoimituksen avulla

$$c_i = \lfloor (x_{i-1} + y_{i-1} + c_{i-1})/B \rfloor, \tag{2.5}$$

missä $\lfloor . \rfloor$ tarkoittaa saadun luvun katkaisemista kokonaisluvuksi B-kantaisessa järjestelmässä. Esimerkiksi kymmenjärjestelmässä $\lfloor (9+3+0)/10 \rfloor = \lfloor 12/10 \rfloor = 1$. Operaatiot s_i ja c_i muodostavat nk. Carry-yhteenlaskimien peruskomponentin, joka on esitetty kuvassa 2.1, ja joita yhdistämällä voidaan suorittaa laskutoimitus mielivaltaisen suurille numeroille.



Kuva 2.1: *Carry yhteenlaskimen peruskomponentti yhdelle merkille.*

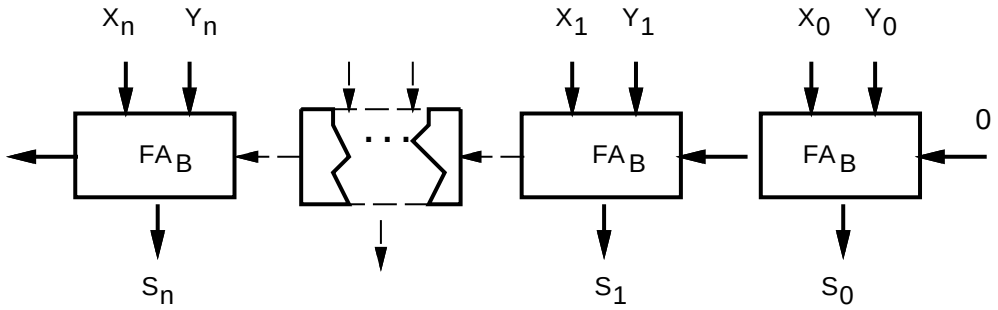
Useamman merkin yhteenlaskualgoritmi voidaan kirjoittaa seuraavasti, kun ensimmäiselle (vähiten merkitsevälle) merkille oletetaan $c_0 = 0$.

1. Alussa merkki $i = 0, c_i = 0$
2. $s_i = [\![x_i + y_i + c_i]\!]_B$
3. $c_{i+1} = \lfloor (x_i + y_i + c_i)/B \rfloor$
4. Jos luvussa on merkkejä jäljellä: $i = i + 1$, jatka kohdasta 2.

Edellämainittu algoritmi voidaan havainnollistaa suhteellisen helposti käyttämällä kuvan 2.2 mukaista Carry-laskimien sarjaa. Tämä on helposti toteutettavissa myöhemmin esiteltävillä logiikkapiireillä.

Se mitä toimituksen jälkeiselle viimeiselle c_i:lle tehdään on määrittelykysymys. Mikäli se on nollasta poikkevava, tiedetään sen perusteella, että valittu lukuesityksen pituus on liian lyhyt esittämään tehdyn laskutoimituksen tulosta. Useimmiten tämä tarkoittaa virheilmoituksen antamista tai muita erikoistoimenpiteitä.

Taulukossa 2.2 on muutamia esimerkkejä Carry-yhteenlaskimen käytöstä.



Kuva 2.2: *Kahden n-numeroisen luvun yhteenlasku Carry-yhteenlaskimilla.*

Esimerkki 1. 10-järjestelmän lukujen 1035 ja 2956 yhteenlasku.

```
                5+6 =    1  C=1
      10x(5+3+C)    =  90  C=0
     100x(0+9+C)    = 900  C=0
  + 1000x(1+2+C)    =3000  C=0
  ----------------------------
                 ==  3991
```

Esimerkki 2. 16-järjestelmän lukujen A03 ja 02F yhteenlasku.

```
                3+F =  2   C=1
      10x(0+2+C)    = 30   C=0
   + 100x(A+0+C)    =A00   C=0
  ----------------------------
                 ==  A32
```

Esimerkki 3. 2-järjestelmän lukujen 1110 ja 0111 yhteenlasku.

```
                0+1 =    1 C=0
      10x(1+1+C)    =   00 C=1
     100x(1+1+C)    =  100 C=1
    1000x(0+1+C)    = 0000 C=1
  +10000x(0+0+C)    =10000 C=0
  ----------------------------
                  == 10101
```

Taulukko 2.2: Esimerkkejä Carry-yhteenlaskimen käytöstä eri lukujärjestelmissä.

Luku 3

Fyysinen toteutus ja logiikkapiirit

Palaamme nyt digitaalisen järjestelmän määritelmään, jonka mukaan se on fyysinen järjestelmä, jonka määrää rajallinen (äärellinen) joukko syöte- ja vastesignaaleja. Tähän kuvaukseen sopii hyvin suuri joukko reaalimaailmasta löytyviä esimerkkejä. Kannaltamme onkin oleellista, että voimme vaikuttaa järjestelmän tekemään muunnokseen syötesignaaleista vastesignaaleiksi. Periaatteessa mikä tahansa fyysinen järjestelmä, jonka tekemän signaalimuunnoksen pystymme määräämään, mahdollistaa universaalin laskukoneen tekemisen. Eräät nykyään erikoisena pidettävät ratkaisut, kuten kvanttimekaniikka, optinen tietojenkäsittely tai DNA-molekyyleihin perustuva laskenta saattavatkin tulla kyseeseen tulevaisuuden tietokoneissa. Toteutustekniikan valinta ei kuitenkaan vaikuta tietokoneen loogiseen toimintaperiaatteeseen. Tulevaisuuden tietokone perustuukin suurella todennäköidyydellä perinteiseen tietojenkäsittelyn teoriaan.

Tämän päivän tekniikka perustuu elektronisiin transistoreihin. Transistori on todellisuudessa jatkuva-arvoinen komponentti, jossa toisella jännitelähteellä voidaan ohjata toisen jännitteen läpikulkua komponentin kautta. Käytännössä toiminta jaetaan kynnystämällä kahteen tilaan, joiden avulla voidaan rakentaa yksinkertaisia kaksitilaisia kytkimiä: kytkin on joko auki tai kiinni. Tätä kautta päädytään binäärilukujärjestelmään tietokoneen sisäisessä tiedon esityksessä. Kaksikantainen järjestelmä on valittu käytännön syistä. Sen avulla elektronisten komponenttien hallinta on osoittautunut kaikkein helpoimmaksi epätäsmällisyyttä ja häiriöitä sisältävässä reaalimaailmassa. Mitään teoreettista estettä useampi kuin kaksitilaisten elektronisten kytkimien tekemiseen ei ole.

Toisaalta tarkkojen fyysisten piirien tekeminen, joissa lämpötila, jännitteen pienet vaihtelut, mikroskooppisen pienet roskat tai luonnon taustasäteily eivät vaikuttai-

si piirien vastearvoihin, on mahdotonta. Mitä enemmän analogiapiirejä kytketään yhteen, sitä suurempi on näiden virheiden yhteisvaikutus, ja lopputuloksena saatu arvo voi olla enemmän sattuman kuin piirin alkuperäisten syötteiden tulosta.

2-arvoisissa piireissä virheliikkeille jää paljon tilaa, mutta niitäkin tapahtuu. Tietokone ei siis ole aina oikeassa, kuten seuraava lainaus osoittaa.

> **Scientific American, Feb. 1980, p. 70:**... *radiation failures (presumably from background radiation) causing random failures 3,000/million hours of operation in a 256k charge-coupled device.*

Nykyisellä tekniikalla on muutamia muihin tekniikkoihin nähden ylivertaisia ominaisuuksia:

- Se mahdollistaa systeemitason ratkaisut, joissa järjestelmä voidaan suunnitella itsenäisistä komponenteista, joita voidaan yhdistellä suuremmiksi kokonaisuuksiksi. Esimerkiksi analogiaelektroniikan alueella tähän ei ole pystytty yleisellä tasolla.
- Edellämainittu kohta on pohjana hierarkkiselle suunnittelulle, jossa alemman tason ratkaisuja käytetään rakennuspalikkoina korkeammalle tasolle.
- Valmistustekniikka on kehittynyt niin, että suuria kokonaisuuksia pystytään integroimaan yhteen piiriin.

Tie elektronisista komponenteista tietokoneeseen on nyt (hieman mielivaltaisesti) luokiteltavissa seuraaviin tasoihin

1. Vastukset, diodit, transistorit.
2. Logiikkapiirit.
3. Multiplekserit, demultiplekserit, laskimet, jne.
4. Kiikut (flip-flop), rekisterit ja muistipiirit.
5. Prosessorin osat (viipaleet), erikoispiirit, ja prosessorit.
6. Keskusyksikkö, väylä, muisti ja oheislaitteet.

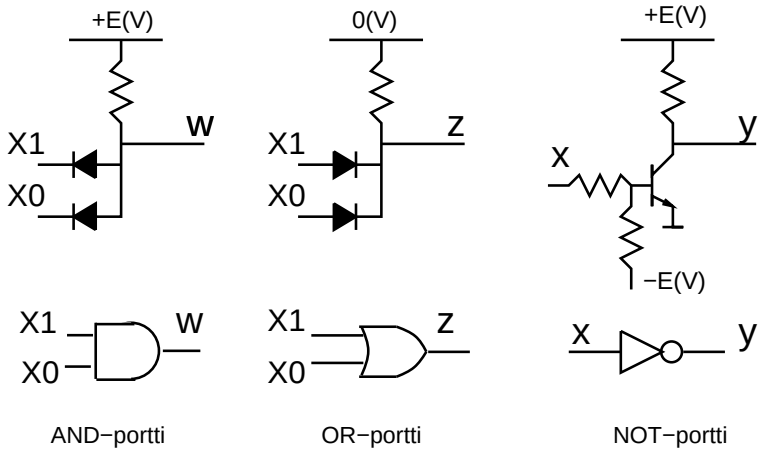
Tietokoneen toimintaperiaatteet on mahdollista esittää varsin pienellä joukolla peruslogiikan operaatioita, joilla on myös suora tulkinta sähköisinä piireinä, kun signaalit esitetään 2-kantaisessa järjestelmässä. Usein puhutaankin logiikkapiireistä, tai lyhyemmin pelkästään piireistä.

Logiikkapiirejä tarvitaan tekemään signaalien valintaa, koodauksia, tiedon välitystä ja tiedon tallentamista. Niiden käytöllä on etuna myös graafinen tulkinta, jolloin toiminnan voi havainnollistaa kuvin, ilman matemaattista esitystä.

Tässä yhteydessä ohitamme suurimman osan kombinatoristen piirien teoriasta, keskittyen vain niihin komponentteihin, jotka ovat tarpeellisia tietokoneen toiminnan esittelemiseksi.

3.1 Peruslogiikat (AND-OR-NOT)

Tietokoneen toiminnan kannalta logiikan fyysinen toteutus tai transistorien toiminnan ymmärtäminen ei ole oleellista. Asiasta kiinnostuneille tuotakoon ilmi, että AND-/, OR-/ ja NOT-operaatioiden määräämiseen tarvitaan yksinkertaisimmillaan muutama vastus, diodi ja transistori, esimerkiksi kuten kuvassa 3.1 on esitetty.

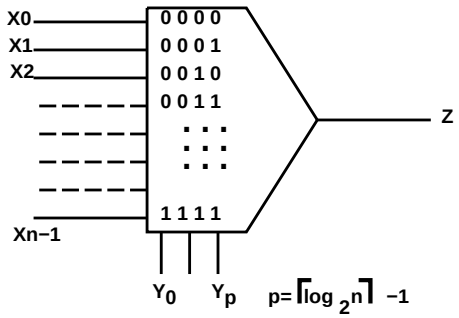


Kuva 3.1: *Peruslogiikkapiirien (porttipiirien) toteutus diodeilla, vastuksilla ja transistoreilla.*

Tässä sähköiset signaalit `+E(V)` ja `0(V)` koodataan 1:ksi ja 0:ksi, jolloin piirien tilat vastaavat aiemmin esitettyjä logiikkataulukkoja. Käytännössä logiikka tehdään hieman eri tavalla, johtuen siitä, että vastusten ja diodien runsas käyttö integroiduissa piireissä ei johda tehokkaimpaan lopputulokseen tilankäytön, nopeuden tai tehonkäytön kannalta.

3.2 Valitsimet (multiplekserit)

Eräs tärkeimmistä peruspiireistä on *valitsin* eli *multiplekseri*, joka on esitetty kuvassa 3.2. Sen avulla jokin syötteistä $x_0, x_1, \ldots, x_{n-1}$ johdetaan vasteeksi z. Siis z saa saman arvon kuin jokin x_i. Se mikä syöte vasteeseen johdetaan määrätään ohjausarvoilla $y_0, y_1, \ldots y_{\lceil \log_2 n \rceil - 1}$, missä $\lceil \cdot \rceil$ tarkoittaa luvun pyöristämistä ylöspäin seuraavaan kokonaislukuun. Esimerkiksi $\lceil \log_2 5 \rceil = \lceil 2.32 \rceil = 3$.

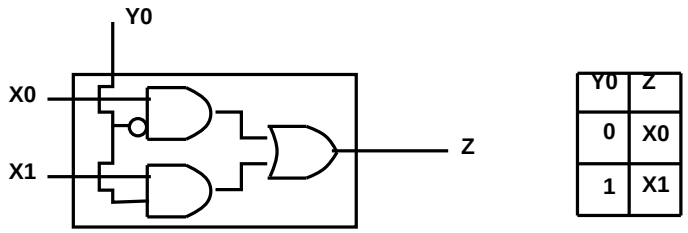


Kuva 3.2: *Valitsimen periaate. Se valitsee yhden syötteistään annetun ohjauksen määräämänä.*

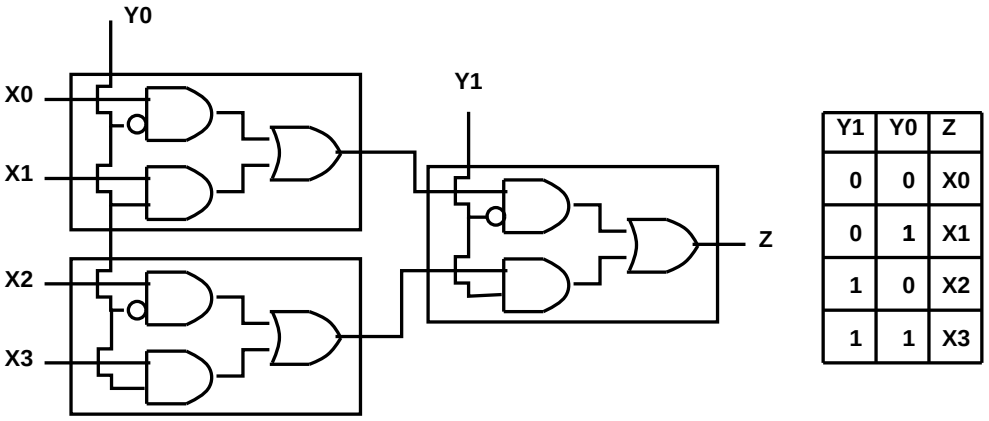
Kyseessä on siis aiemmin esitetty koodaus, jossa n symbolia voidaan esittää käyttäen $\lceil \log_2 n \rceil$ bittiä. Nyt tätä tietoa käytetään valintakriteerinä. Piirin fyysinen toteutus käyttäen kuvan 3.1 peruslogiikkoja on hyvin yksinkertainen, kuten kuvasta 3.3 käy ilmi. Tässä kaksi totuusarvoa x_0 ja x_1 ovat syötteinä, joista toinen johdetaan ohjaussignaalin y_0 arvosta riippuen vasteen z arvoksi. Totuustaulu kuvassa kertoo saman asian.

Valitsimen peruspiirien luonteeseen kuuluu, että suurempia piirejä voidaan koostaa käyttäen pienempiä osasia apuna. Kuvassa 3.4 on esitetty kuinka $2 \rightarrow 1$-valitsimia hyväksikäyttämällä voidaan koostaa suurempi $4 \rightarrow 1$-valitsin.

Kannattaa myös huomata, että kuvan 3.4 esittämän periaatteen laajentaminen ei välttämättä johda tehokkaimpaan ratkaisuun tarvittavien logiikkapiirien lukumäärän tai logiikan "nopeuden" suhteen. Toisaalta se on varsin helppo menetelmä mielivaltaisten koodausverkkojen rakentamiseen.



Kuva 3.3: $2 \rightarrow 1$-*valitsimen totetutus AND-, NOT- ja OR-operaatioilla, missä ympyrä* $\circ$ *edustaa NOT-operaatiota.*



Kuva 3.4: $2 \rightarrow 1$-*valitsimen laajentaminen suuremmaksi* $4 \rightarrow 1$-*valitsimeksi.*

Valitsin matemaattisena kuvauksena

Asiasta kiinnostuneille mainittakoon, että Boolen algebran avulla valitsin voidaan esittää myös matemaattisena kuvauksena $z = f(y_0, y_1, \ldots y_p,\ x_0, x_1, \ldots x_{n-1})$, joka yksinkertaisemmassa $2 \rightarrow 1$-tapauksessa on

$$z = (x_0 \wedge \overline{y_0}) \vee (x_1 \wedge y_0),$$

ja yleisemmässä $N \rightarrow 1$-tapauksessa kuvan 3.4 periaatetta yleistäen

$$z = (x_0 \wedge \overline{y_0} \wedge \overline{y_1} \wedge \ldots \wedge \overline{y_p}) \vee (x_1 \wedge y_0 \wedge \overline{y_1} \wedge \ldots \wedge \overline{y_p}) \vee \ldots \vee (x_{n-1} \wedge y_0 \wedge y_1 \wedge \ldots \wedge y_p),$$

missä $p = \lceil \log_2 N \rceil - 1$. Tämä voidaan tiivistää monellakin tavalla, esimerkiksi muotoon

$$z = \bigvee_{i=0}^{n-1} x_i \wedge \left[\bigwedge_{k=0}^{p} \overline{(i_{(2)})_k \otimes y_k} \right],$$

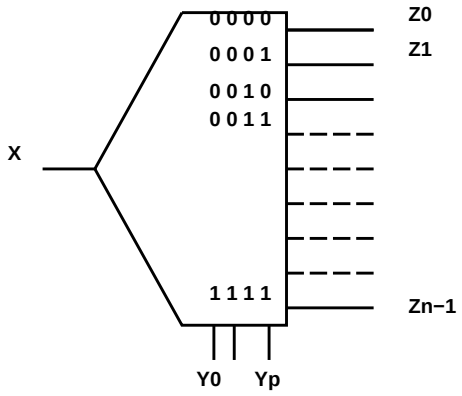
missä operaatio $a \otimes b$ on nk. XOR-funktio $(a \wedge \overline{b}) \vee (b \wedge \overline{a})$ ja $(i_{(2)})_k$ tarkoittaa kokonaisluvun i binääriesityksen k:nnetta bittiä.

Asiaan perehtymättömälle edellä annettu matemaattinen kuvaus saattaa näyttää turhankin "kryptiseltä". Kokeneempi lukija saattaa tunnistaa osan $[i \otimes Y]$ luvun $Y = (y_0, y_1, \ldots y_p)$ “maskaamiseksi” bitti kerrallaan, eli tunnistamiseksi annetun bittijonon $i = (i_0, i_1, \ldots i_p)$ avulla. Maskioperaatio antaa arvokseen 1, jos $Y = i$ ja 0 muuten. Kaava voidaan siis lukea selkokielisenä muodossa

$$z = x_i \quad \text{jos} \quad i = Y.$$

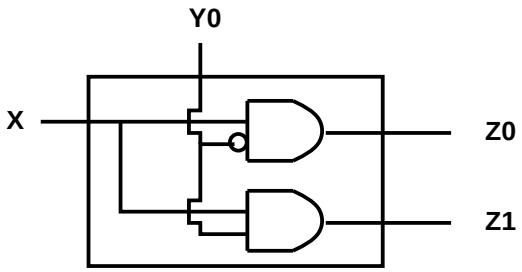
3.3 Haaroittimet (demultiplekserit)

Valitsimen vastine, joka toimii ikäänkuin toisinpäin, on *haaroitin* eli *demultiplekseri*. Se ohjaa kuvassa 3.5 esiintyvän syötteen x yhteen vasteista $z_0, z_1, \ldots, z_{n-1}$. Kuten multiplekserin tapauksessa, niin tässäkin tapauksessa valinta on koodattu ohjaussignaaleihin $y_0, y_1, \ldots, y_{\lceil \log_2 n \rceil - 1}$, joita on $\lceil \log_2 n \rceil$ kappaletta.



Kuva 3.5: *Haaroittimen periaate. Syöte johdetaan yhteen vasteeseen.*

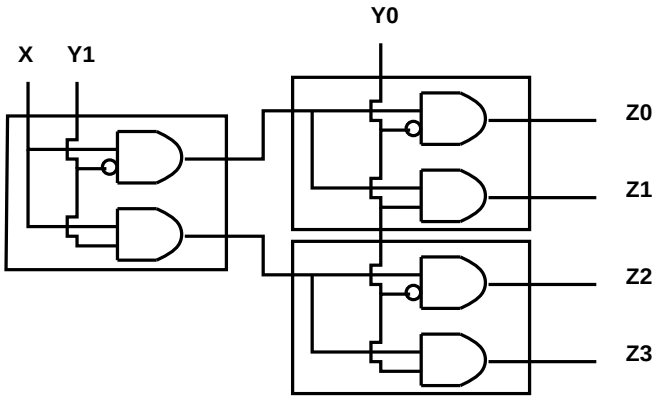
Kuten valitsimen tapauksessa, on myös haaroittimen toteutus varsin yksinkertainen (kuva 3.6). Se voidaan rakentaa $1 \rightarrow 2$-tapauksessa kahdella AND-piirillä sekä yhdellä NOT-piirillä. Myös laajennus $1 \rightarrow N$-piiriksi onnistuu kuten aiemminkin, käyttämällä useita piirejä yhdessä kuvan 3.7 osoittamalla tavalla.



Y0	Z0	Z1
0	X	0
1	0	X

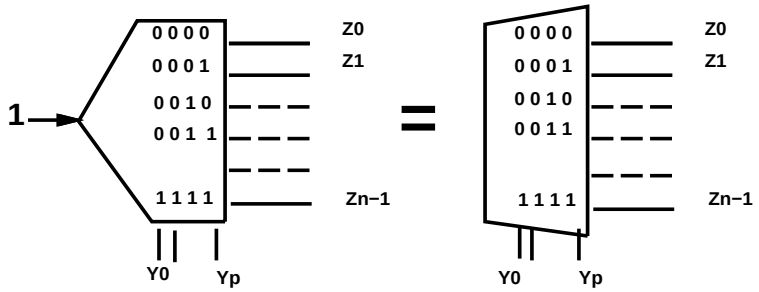
Kuva 3.6: $1 \rightarrow 2$-*haaroittimen rakenne ja totuustaulu.*

Usein haaroitinta käytetään koodauksen purkamiseen, siirettäessä tietoa n:n komponentin valinnasta, käyttäen mahdollisimman pientä määrää $\lceil \log_2 n \rceil$ fyysisiä johdotuksia. Tällöin haaroittimen tehtävänä on muuttaa koodaus siten, että n komponentista yksi saa signaalin 1, ja muut saavat arvon 0, jolloin haaroittimen syöte x on aina 1. Tällöin syötettä x ei merkitä erikseen, vaan käytetään kuvan 3.8 tapaa komponentin piirtämiseen.



Y1	Y0	Z0	Z1	Z2	Z3
0	0	X	0	0	0
0	1	0	X	0	0
1	0	0	0	X	0
1	1	0	0	0	X

Kuva 3.7: *Haaroittimen laajentaminen peruspiirejä yhdistelemällä* $1 \rightarrow 4$*-piiriksi.*



Kuva 3.8: *Haaroitin koodauksen purkajana, jolloin syöte* X *on aina 1.*

Haaroitin matemaattisena kuvauksena

Myös haaroitin voidaan esittää matemaattisena kuvauksena, joka on muotoa $z_i = f_i(y_0, y_1, \ldots y_p, x)$. Yksinkertaisemmassa $1 \rightarrow 2$-tapauksessa se on

$$\begin{cases} z_0 & = x \wedge \overline{y_0} \\ z_1 & = x \wedge y_0 \end{cases}$$

ja yleisemmässä $1 \rightarrow N$-tapauksessa tiivistettynä, esimerkiksi

$$z_i = x \wedge \left[\bigwedge_{k=0}^{p} \overline{\left(i_{(2)}\right)_k \otimes y_k} \right],$$

missä merkinnät ovat samat kuin aiemmin. Kyseessä on jälleen bittijonon tunnistuksen avulla tehtävä valinta, joka tarkoittaa seuraavaa

$$z_i = \begin{cases} x & \text{jos } \ i = Y \\ 0 & \text{muuten.} \end{cases}$$

Luku 4

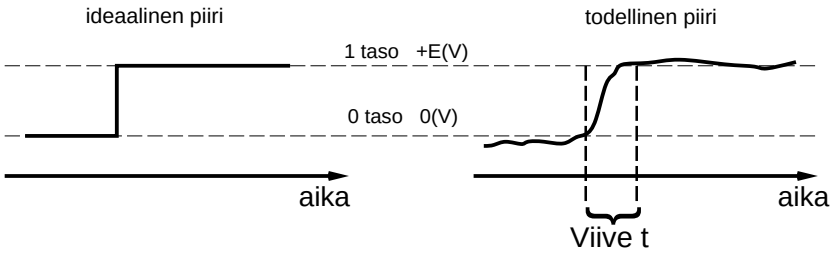
Diskreetti aika ja tietokoneen muisti

Aika on tärkeä käsite tietokoneen toiminnan kannalta. Kaikissa fyysisissä järjestelmissä toimintojen suorittamiseen, muunnokseen syötteistä vasteisiin, kuluu tietty aika. Lisäksi jatkuvasti tilaansa muuttava järjestelmä on hankalasti hallittava. Muutokset on siis pystyttävä pysäyttämään hetkellisesti, mikä on diskreetin digitaalisen järjestelmän keskeisimpiä ominaisuuksia. Tähän tarkoitukseen tarvitaan muistia, jonka tarkoitus on pitää signaali määrätyn ajan muuttumattomana.

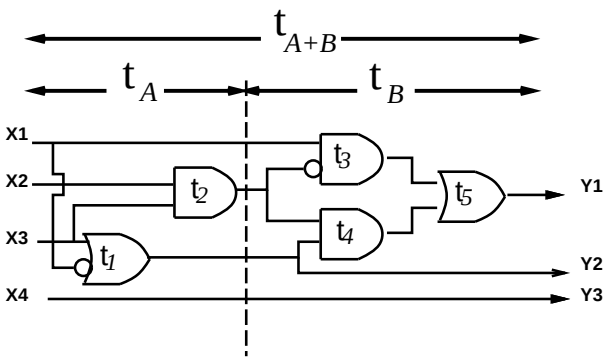
Fyysisellä tasolla aika tulee vastaan tarkastellessa sähköisten piirien viiveitä. Transistorien ym. komponenttien toiminnassa kuluu tietty viiveaika t ennenkuin vastearvot asettuvat määrättyihin tasoihin riittävän luotettavasti. Kaksiarvoiselle logiikalle tätä havainnollistetaan kuvassa 4.1. Näemme siinä vasteen vaihtumisen tilasta 0 tilaan 1, sekä ideaalisessa että todellisen kaltaisessa tilanteessa. Toimintaa hidastava viive t on määrättävä niin pitkäksi, että sen jälkeen olemme varmoja signaalin uudesta tilasta.

Jos tarkastelemme kuvan 4.2 laajempaa piiriä, niin viiveiden t_A ja t_B (jotka on laskettu yhteen komponenteille ennen ja jälkeen katkoviivaa) määräämänä signaalit y_1, y_2 ja y_3 asettuvat luotettavasti vasta kokonaisajan t_{A+B} jälkeen. Jos verrataan signaaleja keskenään, niin vasteen y_3 arvoon voidaan luottaa heti, kun syötteet x_1, x_2, x_3 ja x_4 ovat asettuneet. Viiveen t_A jälkeen voidaan luottaa vasteen y_2 arvoon, kun taas vasteeseen y_1 voidaan luottaa vasta ajan $t_{A+B} = t_A + t_B$ jälkeen.

Tietokoneissa viiveitä käsitellään monellakin tavalla, mutta toiminnan kannalta kaikkein mielenkiintoisin on rekisterien, eli muistin käyttö, jossa tulevat signaalit tallennetaan väliaikaisesti ja lähetetään eteenpäin samalla ajanhetkellä. Tätä kutsutaan synkronoinniksi. Kun kaikkia tietokoneen toimintoja synkronoidaan samoilla ennal-



Kuva 4.1: *Ideaalisen logiikan ja todellisen toteutuksen välinen ero havainnollistettuna.*



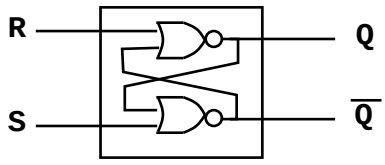
Kuva 4.2: *Kombinatorisen piirin viive on sarjaan kytkettyjen komponenttien viiveiden summa. Tämä määrää piirin suurimman käyttönopeuden. Kokonaisviive on* $t_A + t_B$*, missä* $t_A = \max\{t_1, t_2\}$ *ja* $t_B = \max\{t_3, t_4\} + t_5$.

ta määrätyillä ajanhetkillä, puhutaan kellosta ja kellotaajuudesta. Se siis kertoo millä tahdilla informaatio komponettien välillä kulkee.

Kellotetun järjestelmän toteutus vaatii, että laitteessa ei saa olla yhtään kellotetuilla rekistereillä (muisteilla) jakamatonta osaa, jossa viive on pidempi kuin koneen kellotaajuus, mikä selittää sen miksi suurempi kellotaajuus on myös kalliimpi ratkaisu. Monessa laitteessa onkin useita kelloja, jolloin tärkeimmät osat voivat toimia muita nopeammin, ja muissa säästetään kustannuksissa.

4.1 Rekisterit ja kiikut

Yksinkertaisin peruspiiri, jolla voi rakentaa muistin, on kiikku (engl. flip-flop). Hieman hassunkurinen nimi kuvaa piirin ominaisuutta säilyttää ja vaihtaa tilaansa tietyillä ohjauksilla. Kiikkuja on muutamia erilaisia. Kuvassa 4.3 on nk. RS-kiikku (Set-Reset-kiikku) ja sen totuustaulu. Lähtö `Q` ja sen negaatio $\overline{\texttt{Q}}$ määräytyvät asetussignaalien `S` ja `R` sekä piirin edellisen tilan mukaan. Tämä on tärkeää, sillä tilan säilyvyyden kautta piirillä on kyky toimia muistina.



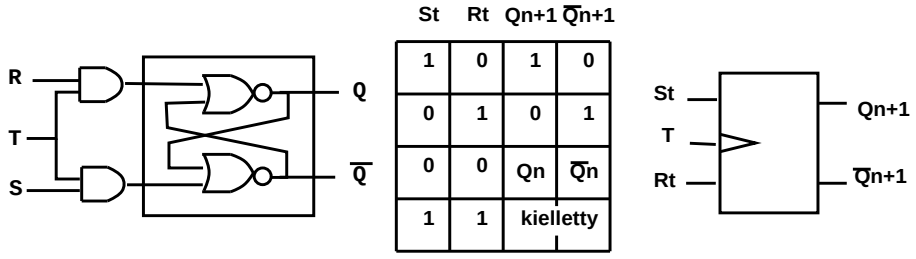
S	R	Q	$\overline{\texttt{Q}}$
1	0	1	0
0	1	0	1
0	0	Ei muutu	
1	1	kielletty	

Kuva 4.3: *NOR (NOT-OR)-piireillä toteutetun RS-kiikun rakenne ja totuustaulu.*

Toiminta, joka on kuvattu myös totuustaulussa, on seuraava. Asettamalla hetkellisesti `S=1, R=0` asettuu piiri viiveen jälkeen tilaan `Q=1`. Vastaavasti syötteellä `S=0, R=1` siirtyy piiri tilaan `Q=0`. Näin piirille saadaan asetettua muistiin joko 0 tai 1. Kun `R=0, S=0`, pitää piiri nykyisen tilansa. Tapauksessa `R=1, S=1` sekä `Q=0` että $\overline{\texttt{Q}} = \texttt{0}$, eli vaste ei ole ennakoitavissa. Tila `R=1, S=1` onkin nk. RS-kiikun kielletty tila, johon joutuminen tulee estää järjestelmän suunnitteluvaiheessa.

Kellotetuissa järjestelmissä piirillä on ylimääräinen syöte `T`, joka tulee erilliseltä kellopiiriltä, ja vaihtelee vuorotellen nollan ja ykösen välillä. Toiminnan ajatuksena on, että edelläkuvatut muutokset tapahtuvat aina yhden kellojakson aikana. Toiminta

tälle RST-kiikulle (Kuva 4.4) on siis sama kuin aiemminkin, mutta nyt ajan merkitys on helpompi esittää: aika-askeleella $n+1$ on vaste $\mathtt{Q}_{n+1}$ määrättävissä R:n, S:n ja edellisen aika-askeleen arvon $\mathtt{Q}_n$ perusteella.



Kuva 4.4: *Synkronoitu RS-kiikku eli RST-kiikku, totuustaulu sekä kiikkua kuvaava symboli.*

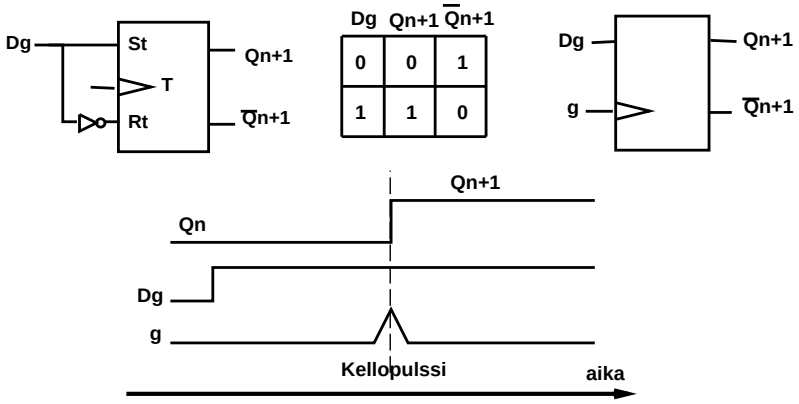
Muistiominaisuuden kannalta käyttöön riittää R ja S signaaleja yksinkertaisempikin asetus, eli viimeisen syötearvon muistaminen seuraavaan kellopulssiin asti. Tämän toteuttava D-kiikku on esitetty totuustauluineen kuvassa 4.5. On varsin helppo nähdä, että D-kiikku on toteutettavissa esimerkiksi RST-kiikun avulla. Piirin toiminta on kellopulssin g määräämä, mitä on havainnollistettu kuvan 4.5 alalaidassa. Kellopulssin vaihtuessa tallentaa piiri sisääntulonsa Dg arvon ja pitää sen niin kauan, kunnes seuraava pulssi g saapuu sisääntuloon.

D-kiikulla voidaan toteuttaa sinänsä yksinkertainen mutta nopea muisti, *rekisteri*. Se on edelläkuvatun kaltainen muistipaikka yhdelle tai useammalle bitille. Kytkemällä kiikkuja yhteen voidaan koota suurempia rekisterejä, kuten kuvassa 4.6, jossa rekisteriin mahtuu 6-bitin sana.

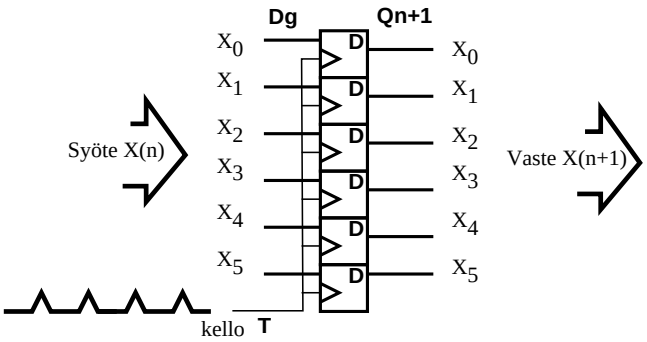
Jos mikropiireistä tahdotaan tehdä nopeampia saman valmistusprosessin puitteissa, on piirin osia jaettava rekistereillä yhä pienempiin osiin. Tällöin kellotaajuutta voidaan kasvattaa alittamatta komponenttien pienimpiä sallittuja viiveitä.

4.2 Muistipiirit

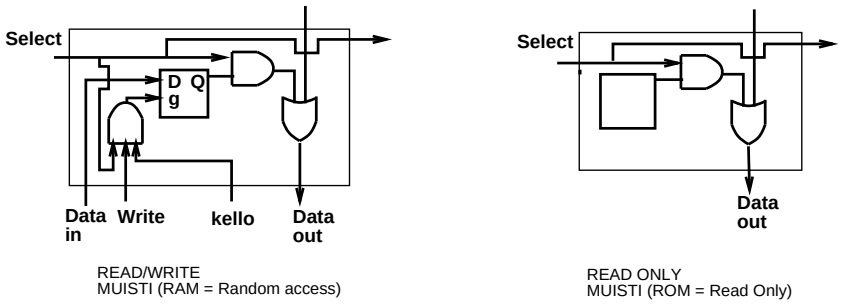
Tietokoneen todellista muistia rakennettaessa kiikkujen käyttö on tuhlausta, sillä muistirakenteita voidaan toteuttaa halvemmallakin tekniikalla. Periaatteessa muistin toiminta voidaan toki esittää käyttäen kiikkuja, kuten kuvassa 4.7.



Kuva 4.5: *D-kiikun toiminta. Se lukee syötteen arvon* `Dg` *muistiinsa, kun* `g`*-sisääntulossa on pulssi (ykkönen) ja säilyttää arvon samana seuraavan pulssin saapumiseen asti.*



Kuva 4.6: *Rekisterimuisti kuudelle bitille D-kiikkujen avulla toteutettuna.*



Kuva 4.7: *RAM- ja ROM-tyyppisten yhden bitin muistisolujen periaatteellinen toiminta.*

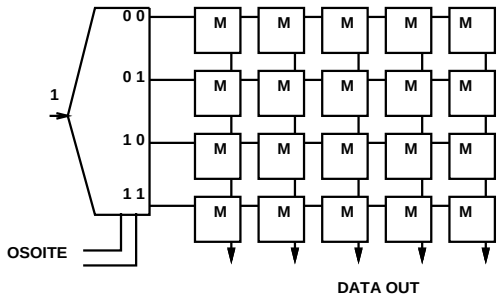
Muisteja on kahta tyyppiä. "Read only" eli ROM-muisti on rakennettu siten, että tallennusta ei voi tehdä ilman erikoistoimenpiteitä, mutta kerran tallennettu informaatio säilyy periaatteessa pysyvästi. Sensijaan "Random access" eli RAM-tyyppistä muistia voi käyttää joko kirjoittamiseen tai lukemiseen. Niinpä muistisolulla onkin ylimääräisenä sisääntulona signaali `Write`, joka kertoo, että nykyinen muistisolun sisältö (mikäli solu on valittu, `Select=1`) korvataan arvolla `Data in`.

Varsinainen muistipiiri koostuu suuremmasta ryhmästä muistisoluja (englanniksi memory cell), jotka soveltuvat yhden bitin tallentamiseen. Solut kootaan matriisiksi, jossa on tietty määrä rivejä ja sarakkeita kuten kuvassa 4.8. Organisointi järjestetään siten, että muistia voidaan lukea rivi kerrallaan. Rivin valinta suoritetaan johtamalla halutulle riville signaali `Select=1` ja muille riveille nolla. Käytännössä valinta on aina koodattu, jolloin muistipiirin sisällä on haaroitin, joka muuttaa binääriluvun haetun rivin signaaliksi kuvan 4.8 tavalla. Rivin järjestysnumeroa kutsutaankin osuvasti osoitteeksi.

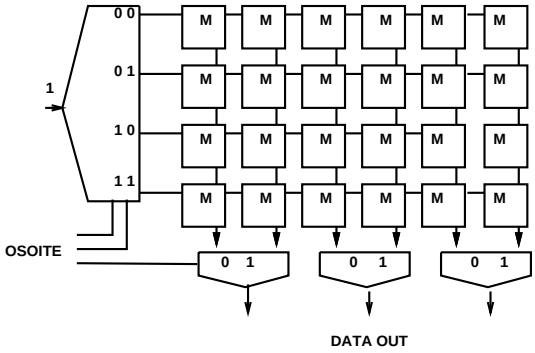
Rivin sisältö saadaan muistin vasteeksi (ulostuloksi) yhdistämällä kaikki saman sarakkeen muistisolut TAI-operaatiolla. Näin sarakkeen vasteeksi (`Data out`) tulee valitun rivin sisältämän muistisolun arvo.

Muistipiireistä on helppo koota valitsimia ja haaroittimia yhdistelemällä erilaisia muistiorganisaatioita. Tätä voidaan tehdä valmistusteknisistä syistä myös muistipiirien sisällä, kuvan 4.9 tapaan, missä 8×3 bitin muisti on toteutettu 4×6-muistimatriisin ja kolmen $2 \rightarrow 1$-valitsimen avulla.

Vastaavasti kuvassa 4.10 on kaksi sisäiseltä organisaatioltaan 8×3-bitin muistia, jotka on yhdistetty yhdeksi 16×3 bitin muistiksi. Eli käyttöön on saatu 16 muisti-

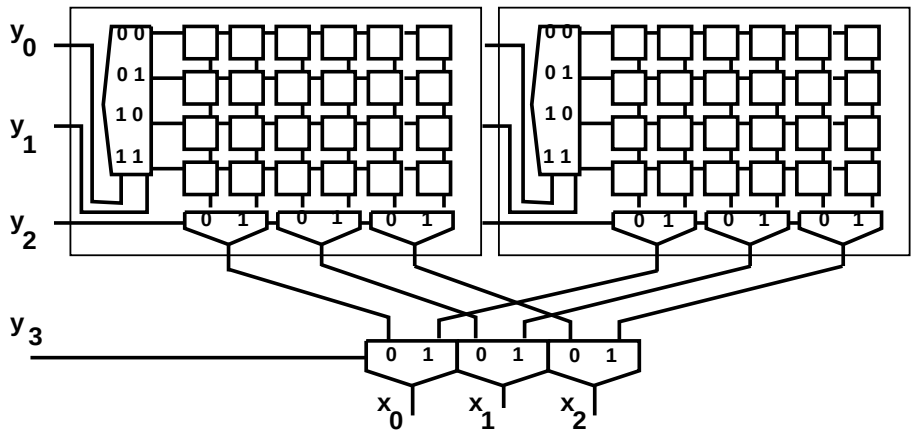


Kuva 4.8: *Muistisolujen yhdisteleminen yhdeksi* 4×5 *bitin muistipiiriksi, josta muistisanan (rivin) valinta tehdään haaroittimen avulla.*



Kuva 4.9: *Muistisolujen yhdisteleminen yhdeksi* 8×3 *bitin muistipiiriksi.*

paikkaa 3-bitin mittaisille sanoille.



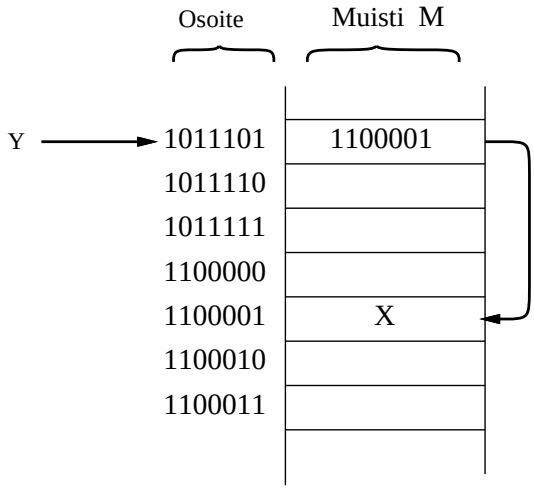
Kuva 4.10: *Muistipiirien yhdisteleminen käyttäen valitsimia.*

Muistin käytöstä ja osoittimista

Muistin esittäminen ja osoittaminen formaalilla notaatiolla tehdään yleensä yksinkertaisesti muodossa $X = M[Y]$, mikä tarkoittaa muistin M osoitteessa Y olevan sanan sijoittamista vasteeksi X. Muisti on siis tästä näkökulmasta samanlainen syöte-vaste-järjestelmä kuin mitä aiemmin on käsitelty.

Tietokoneohjelmissa muistia käytetään usein myös epäsuorasti, osoittamalla $X = M[M[Y]]$, jolloin vasteeksi X tulee osoitteessa $M[Y]$ oleva muistisana, kuten kuvassa 4.11.

Epäsuoran osoittamisen avulla voidaan toteuttaa monia tietojenkäsittelyn keskeisiä algoritmeja ja tietorakenteita, esimerkiksi aliohjelmien kutsut ja linkitetyt listat. Hyödyllisyys perustuu siihen, että ohjelmoitaessa Y on yleensä määrättävä ennalta, jolloin ohjelman suorituksen aikana tapahtuva, kulloisestakin tilanteesta riippuva muistihaku käy vaikeaksi. Sen sijaan muistin $M[Y]$ sisältöä voidaan vaihdella, jolloin myös haettava tieto X vaihtuu osoitteen $M[Y]$ myötä.

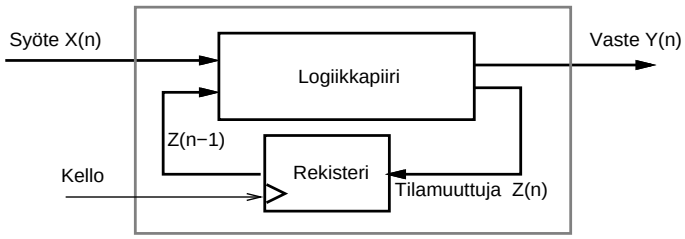


Kuva 4.11: *Epäsuoran muistiosoituksen periaate. Muistipaikan Y sisältö osoittaa, mistä haettava tietoalkio X löytyy: $X = M[M[Y]]$. Useissa ohjelmointikielissä tämänkaltaista muuttujaa Y kutsutaankin osoitinmuuttujaksi.*

Luku 5

Automaatit

Luvussa 1.1 esitellyllä järjestelmällä on mahdollista toteuttaa mikä tahansa yksikäsitteisesti määrätty toiminto (operaatio) syötteistä vasteiksi. Pelkkä syöte-vastekäyttäytyminen ilman sisäistä muistia on kuitenkin varsin rajoittunutta. Lähes kaikissa mielenkiintoisissa järjestelmissä on sisäinen muisti, jonka avulla toimintaa voidaan ohjata paitsi saadun syötteen, niin myös aiempien syötteiden tai annettujen vasteiden vaikutuksesta. Tällöin järjestelmään on lisättävä takaisinkytkentä, jonka kautta järjestelmän tilaa kuvaava vasteinformaatio, tilamuuttuja(t), johdatetaan takaisin syötteeksi kuvan 5.1 tavalla.



Kuva 5.1: *Takaisinkytketyn digitaalisen järjestelmän periaate.*

Takaisinkytkennän avulla voi järjestelmä ohjata tulevia toimintoja omilla vasteillaan, eli toimia *automaattisesti* ilman ulkopuolisten syötteiden suoraa vaikutusta. Logiikassa tämän kaltaista järjestelmää kutsutaan sekvenssilogiikaksi ja se on kaikkien *automaattien* perusta. On tyypillistä, joskaan ei välttämätöntä, että automaatti käy ulkopuolisen syötteen vastaanottamisen jälkeen läpi useita sisäisiä tiloja ennen

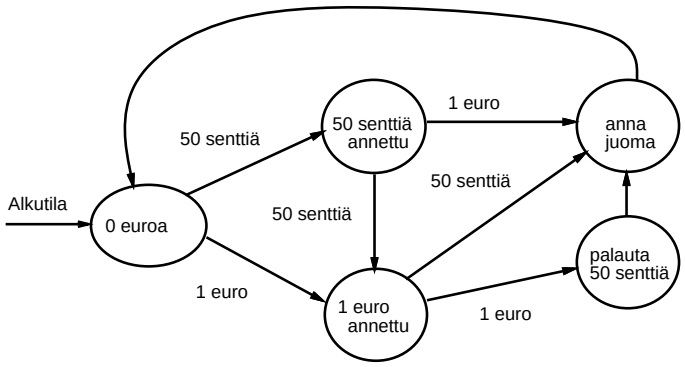
vasteen antamista. Tilasiirtymä tapahtuu aina kellopulssin saapuessa, jolloin automaatin tekemiin syöte-vaste-toimintoihin voi kulua useampia kellopulsseja.

Automaatti voidaan toteuttaa helposti muistipiirien avulla, mitä tässä käytetään askeleena kohti tietokoneen toiminnan ymmärtämistä. Tosin käytännössä toteutukset eivät ole niin pelkistettyjä kuin luvussa annetaan ymmärtää.

Automaatilla on keskeinen osuus tietokoneen toiminnassa. Se mahdollistaa tietokoneen kenties olennaisimman osan, ohjausyksikön toteutuksen. Sen vuoksi onkin hyvä käydä automaatin toimintaa läpi hieman huolellisemmin, esimerkin valossa.

5.1 Esimerkki: Juoma-automaatti

Esimerkkinä on yksinkertaistettu juoma-automaatti, jonka toiminta on esitetty kuvassa 5.2. Automaatti toimii sekä euron että 50 sentin kolikoilla ja juoma maksaa 1,50 euroa. Rahoja voi syöttää missä järjestyksessä hyvänsä. Kun rahaa on tarpeeksi, palauttaa automaatti mahdolliset ylimääräiset rahat sekä antaa juoman. Kuvassa automaatille on esitetty viisi erilaista toimintatilaa, mutta kuten myöhemmin käy ilmi, voidaan haluttu toiminta toteuttaa minimissään kolmella tilalla.



Kuva 5.2: *Yksinkertainen juoma-automaatti.*

Automaatin saamia ulkopuolisia syötesignaaleja ovat rahat ja vasteita ovat palautetut rahat sekä annettu juoma. Näin syötteitä on yksi ja vasteita kaksi kappaletta, jotka voidaan koodata esimerkiksi seuraavasti:

Muuttuja	Arvo	Selitys
Syöte	X=0,	kun syötetty raha on 50 senttiä,
Syöte	X=1,	kun syötetty raha on euro,
Vaste1	Y_0=1,	kun rahaa palautetaan 50 senttiä,
Vaste1	Y_0=0,	kun rahaa ei palauteta,
Vaste2	Y_1=1,	kun annetaan juoma,
Vaste2	Y_1=0,	kun ei anneta juomaa.

Koska automaatin toiminta määräytyy sen aiemman toiminta- ja syötehistorian perusteella, on olennaista, että automaatilla on muistissa tieto toiminnan nykytilasta. Sen on pystyttävä muistamaan missä tilassa se kulloisellakin ajanhetkellä oli, ja toimittava tilan edellyttämällä tavalla. Kuvan 5.2 tilat kattavat kaiken tarpeellisen, kun lähtötilana on, että koneessa ei ole rahaa sisällä.

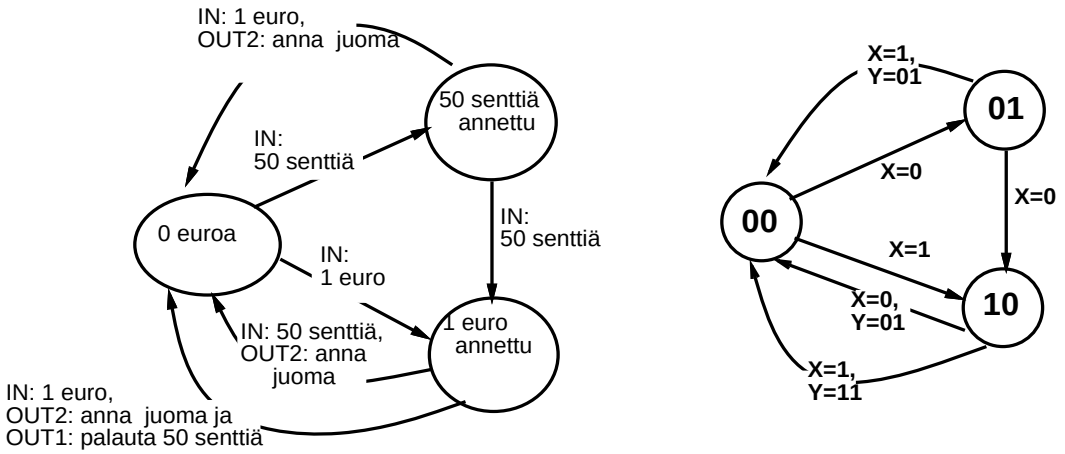
Kussakin kuvan 5.2 tilassa voidaan määrätä vaste sekä syötteiden määräämänä seuraava tila johon automaatti siirtyy. Näin saadaan taulukko 5.1. Tiloissa 4. ja 5. on huomattava, että ne eivät ota vastaan syötettä vaan tilasiirtymä tapahtuu suoraan, kun tilaan liittyvät toiminnat (palautus, juoman anto) on saatu totetutettua. Automaatti toimii nyt siten, että automaatin tila kertoo suoraan minkälaisen vasteen automaatti antaa. Tätä kutsutaan *Mooren* automaatiksi.

Tila	Vaste 1	Vaste 2	Syöte	Seuraava tila
1. (0 €)	0 (ei palautusta)	0 (ei juomaa)		
			0 (50 c)	2.
			1 (1€)	3.
2. (50 c annettu)	0 (ei palautusta)	0 (ei juomaa)		
			0 (50 c)	3.
			0 (1 €)	5.
3. (1 € annettu)	0 (ei palautusta)	0 (ei juomaa)		
			0 (50 c)	5.
			1 (1 €)	4.
4. (palauta 50 c)	1 (palauta 50 c)	0 (ei juomaa)	-	5.
5. (anna juoma)	0 (ei palautusta)	1 (anna juoma)	-	1.

Taulukko 5.1: Mooren automaatti: juoma-automaatin toimintatilat ja tilasiirtymät.

Esimerkin tapauksessa syötteettömien tilojen käyttö ei ole aivan välttämätöntä. On mahdollista koodata juoma-automaatti käyttäen kolmea tilaa, jolloin sen tarvitsee muistaa ainoastaan ne tilat, joissa vaaditusta 1,50 summasta vielä puuttuu rahaa.

Muissa tapauksissa syötesignaaleiden toiminta on ennalta määrättyä. Kun lisäksi logiikan toteutusta silmälläpitäen tilan numero ja automaatin vasteet esitetään binäärilukuina, voidaan automaatin toiminta tiivistää kuvan 5.3 mukaiseksi.



Kuva 5.3: *Automaatin toteutus kolmella tilalla.*

Kuvasta 5.3 huomataan, että kustakin tilasta voi seurata erilaisia vasteita. Tällöin vaste sidotaan suoraan annettuun syötteeseen, eli se riippuu sekä automaatin tilasta että annettavasta uudesta syötteestä. Korostaaksemme tätä on tiloja kuvaava taulukko nyt kirjoitettu hieman erilaiseen järjestykseen (taulukko 5.2). Automaattia, jonka vasteet määräytyvät sekä tilan että syötteen perusteella kutsutaan *Mealy*-automaatiksi.

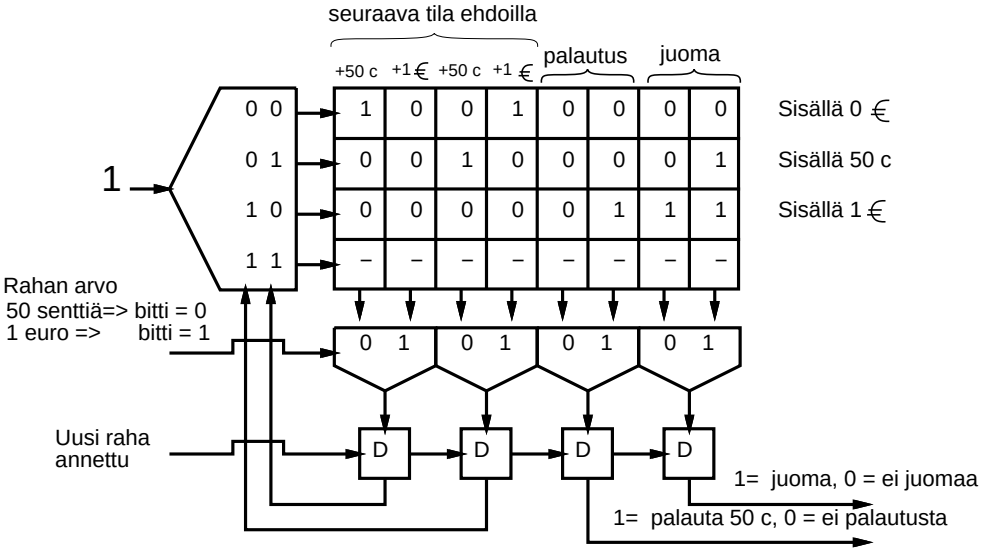
5.2 Automaatin toteutus muistipiirillä

Automaatti on helppo toteuttaa esimerkiksi ROM-muistilla. Periaatteessa toteutus tehdään kirjoittamalla tilasiirtymätaulukko suoraan automaatin muistiin ja johdottamalla osa muistin ulostuloista tekemään seuraavan rivin valintaa. Yksinkertaisimmillaan jokainen taulukon rivi on siis rivi muistipiirissä. Käytännössä saattaa olla helpompaa tehdä valintaa rivien ja sarakkeiden koodauksen välillä. Esimerkiksi taulukon 5.2 Mealy-automaatti voidaan kirjoittaa ROM-piirin muistiin kuvan 5.4 tavalla.

Tässä toteutustavaksi on valittu 4×8 bitin muisti, jossa viimeistä (osoitteella "11")

Tila n	Syöte X	Vaste Y_0	Vaste Y_1	$n+1$
00 (0 €)				
	0 (50 c)	0 (ei palautusta)	0 (ei juomaa)	**01**
	1 (1 €)	0 (ei palautusta)	0 (ei juomaa)	**10**
01 (50 c annettu)				
	0 (50 c)	0 (ei palautusta)	0 (ei juomaa)	**10**.
	1 (1 €)	0 (ei palautusta)	1 (anna juoma)	**00**
10 (1 € annettu)				
	0 (50 c)	0 (ei palautusta)	1 (anna juoma)	**00**
	1 (1 €)	1 (palauta 50 c)	1 (anna juoma)	**00**

Taulukko 5.2: Mealy-automaatti: juoma-automaatin toimintatilat ja tilasiirtymät.



Kuva 5.4: *Mealy-automaatin toteutus muistin avulla.*

valittavaa riviä ei hyödynnetä. Tilan valinta on sijoitettu neljään ensimmäiseen sarakkeeseen. Näissä on seuraavan tilan osoitteet kummankin syötteen (1 € tai 50 c) tapaukselle. Tilan valinta tapahtuu uuden syötteen ohjaamalla valitsimella. Neljään viimeiseen sarakkeeseen on koodattu vasteiden arvot, jotka siis myöskin ovat uuden syötteen ohjaamia. Automaatin toiminta on synkronoitu D-kiikkujen avulla, joita kellotetaan aina kun uusi raha on annettu koneelle.

5.3 Automaatin toteutus rekistereillä ja porttipiireillä

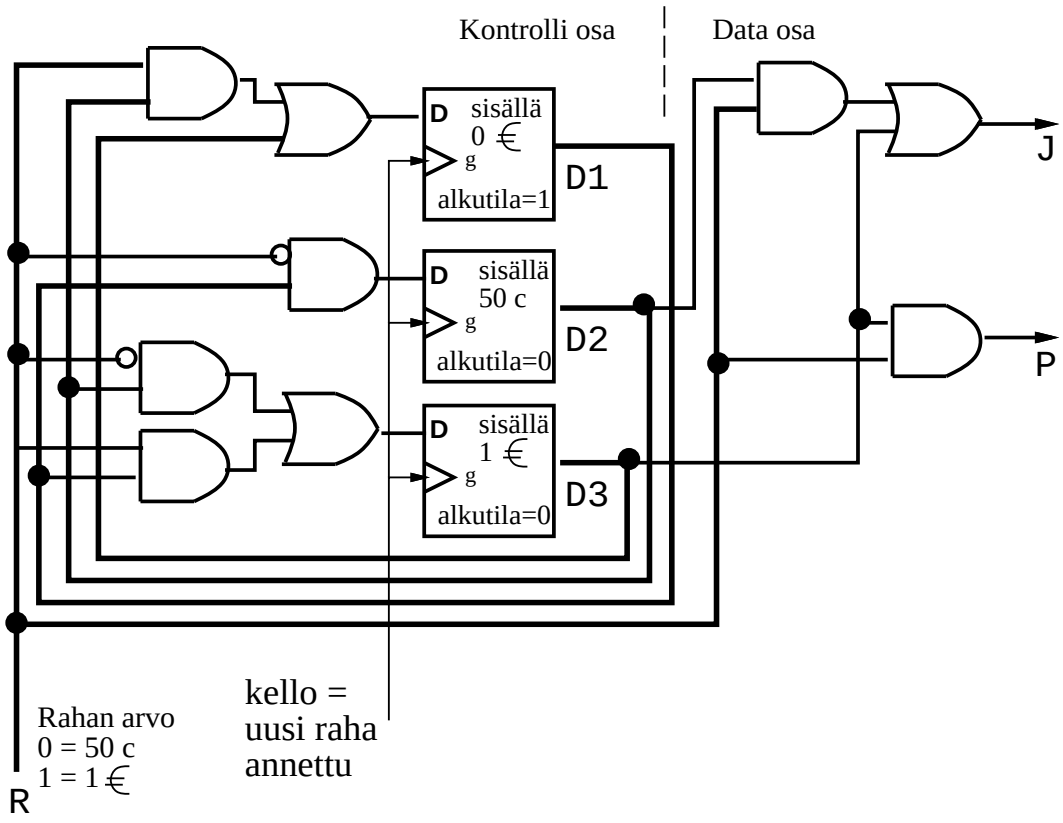
Automaatin voi toteuttaa myös suoraan D-kiikkujen ja porttipiirien avulla, mutta ratkaisu ainakin näyttää monimutkaisemmalta kuin ROM-piirin avulla saavutettu. Toisaalta tarvittavien transistorien määrä on huomattavasti pienempi. Looginen esitys tilasiirtymistä voidaan kirjoittaa suoraan ja toteuttaa suhteellisen suoraviivaisesti.

Käyttämällä merkintöjä D1,D2 ja D3 tiloille (D-kiikuille), R rahan arvolle, J juomalle ja P palautukselle, voidaan toteutuksen looginen rakenne tiivistää taulukon 5.3 mukaisesti. Tilasiirtymät lasketaan synkronoidusti jokaisella kerralla n kun uusi raha laitetaan koneeseen. Tässä koodaus on tehty siten, että kolme kiikkua on varattu esittämään Mealy automaatin kolmea tilaa.

Kiikku	Arvon laskenta
$\mathtt{D1}_{n+1}$	$= \mathtt{D3}_n \vee (\mathtt{D2}_n \wedge \mathtt{R}_n)$
$\mathtt{D2}_{n+1}$	$= \mathtt{D1}_n \wedge \overline{\mathtt{R}}_n$
$\mathtt{D3}_{n+1}$	$= (\mathtt{D1}_n \wedge \mathtt{R}_n) \vee (\mathtt{D2}_n \wedge \overline{\mathtt{R}}_n)$
$\mathtt{J}_n$	$= \mathtt{D3}_n \vee (\mathtt{D2}_n \wedge \mathtt{R}_n)$
$\mathtt{P}_n$	$= \mathtt{D3}_n \wedge \mathtt{R}_n$

Taulukko 5.3: Juoma-automaatin looginen toiminta.

Kuvassa 5.5 on esitetty eräs toteutus loogisille yhtälöille. Tila tunnistetaan siten, että ainoastaan yhdessä kiikussa on muistissa ykkönen, muiden arvojen ollessa nolla. Ykkönen siirtyy uuden rahan tullessa loogisten funktioiden määräämällä tavalla. Esimerkiksi tilasta “sisällä 1 €” siirrytään molemmilla syötteillä (50 c tai 1 €) tilaan"sisällä 0 €". Samalla muutetaan tilan “sisällä 1 €” arvo nollaksi ja tilan "sisällä 0 €"arvo ykköseksi.



Kuva 5.5: *Juoma-automaatin logiikan mahdollinen toteutus D-kiikkujen avulla. Viivan paksuuksilla ei ole merkitystä, ne on tehty tulkinnan helpottamiseksi.*

Katkoviivan vasemmalla puolella on kuvattu logiikka millä tilasta siirrytään toiseen eli toiminnan kontrolliosuus. Katkoviivan oikealla puolella on tiedon käsittelyn (datan) osuus, eli miten syötteistä tulee vasteita. Esimerkiksi siirryttäessä tilasta "sisällä 1 €" tilaan "sisällä 0 €" annetaan juoma.

5.4 Abstrakti tietokone automaattina

On hyvä ymmärtää, että teorian näkökulmasta tietokone on varsin yksinkertainen. Monimutkaisuus syntyy vasta, kun ryhdytään etsimään teknisiä ratkaisuja tietokoneen rakentamiseksi siten, että lopputulos on taloudellisesti kannattava ja käyttäjien toiveita vastaava. Onneksi suuri osa tietojenkäsittelyopin ja tietotekniikan sisällöstä perustuu abstraktin tietokoneen mallille, joka on riippumaton teknisistä toteutustavoista. Näin teorian päätulokset eivät vanhene ajan kuluessa yhtä nopeasti kuin reaalimaailman tietokoneet ja niiden ohjelmistot. Tämän vuoksi esittelemme seuraavassa abstraktin tietokoneen perusperiaatteen.

Kuvassa 5.6 automaatin sisäinen ja ulkoinen maailma on tiivistetty binäärilukumuuttujiin X, Y, Z. Muuttuja Y on tietokoneen ulkomaailmasta saama binäärisana, syöte, jonka sisältö voi vaikuttaa tietokoneen toimintaan. Muuttujan X arvo määrää tietokoneen (sisäisen) tilan, ja muuttuja Z on tietokoneen antama vaste.

Toiminta voidaan nyt tiivistää (esimerkiksi) seuraaviin kahteen kuvaukseen muuttujien arvojoukkojen välillä:

$$\textit{tilanvaihto}, \quad f_1 : X \times Y \mapsto X',$$

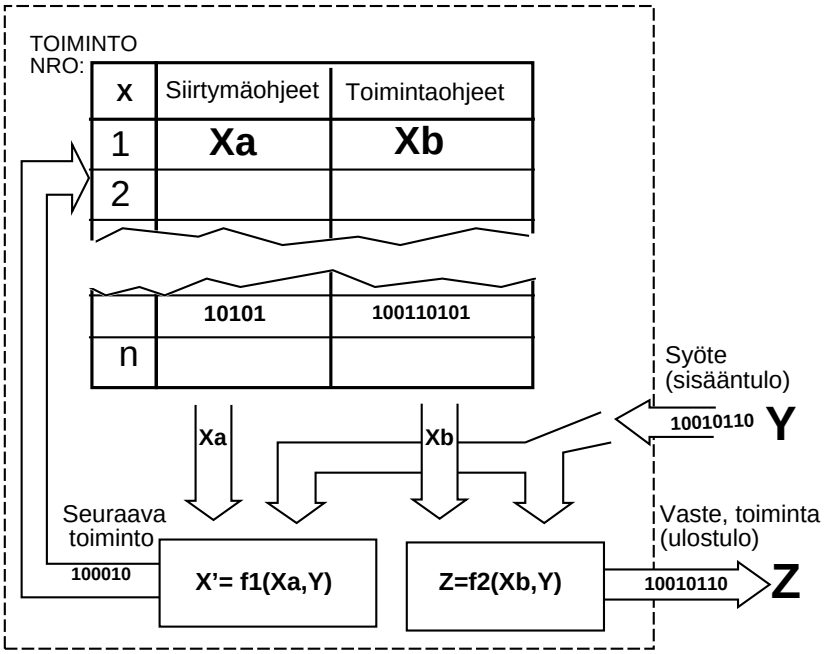
missä kone vaihtaa tilasta X tilaan X', ja

$$\textit{vasteen laskenta}, \quad f_2 : X \times Y \mapsto Z,$$

missä vaste Z määräytyy tilan X ja syötteen Y vaikutuksesta.

Tietokone on pohjimmiltaan universaali laskentakone, jonka periaatteellinen toiminta, kun toteutuksesta ei tarvitse huolehtia, voidaan tulkita automaattina, joka lukee syötteitä ja palauttaa vasteita. Tietyssä mielessä kaikki tietokoneet ovatkin kuvan 5.6 kaltaisia automaatteja, kunhan käsitettä ensin laajennetaan hieman.

Yksinkertaistettuna tietokoneelta vaadittava universaalisuus tarkoittaa, että automaatin on pystyttävä vastaanottamaan syötteenään toisen mielivaltaisen automaa-



Kuva 5.6: *Tietokoneen ydin: kontrolliyksikkö on esitettävissä automaatin tilataulukkona.*

tin kuvaus ja ryhdyttävä toimimaan sen kaltaisesti. Tietokoneen ja automaatin keskeisin ero onkin, että kaikki automaatit eivät ole universaaleja, eli eivät täytä tietokoneen määritelmää. Automaatti toimii ulkoisen havainnoijan kannalta suppealla, ennalta ohjelmoidulla tavalla, kun taas tietokoneelle voi toimintaohjeen lähettää syötteenä, joka määrää minkälainen koneen syöte-vaste-käyttäytyminen tulee tämän jälkeen olemaan.

5.5 Turingin kone

Eräs yksinkertainen malli tietokoneelle (mutta ei ainoa mahdollinen) on nk. Turingin kone. Sen kehitti englantilainen matemaatikko Alan Turing vuonna 1936. Turingin työ oli keskeisessä roolissa myös toisen maailmansodan aikana, sillä hänen suunnittelemiensa lakentalaitteiden avulla liittoutuneet onnistuvat murtamaan saksalaisten Enigma-salakirjoituskoneen koodin ja seuraamaan vihollisen radioliikennettä.

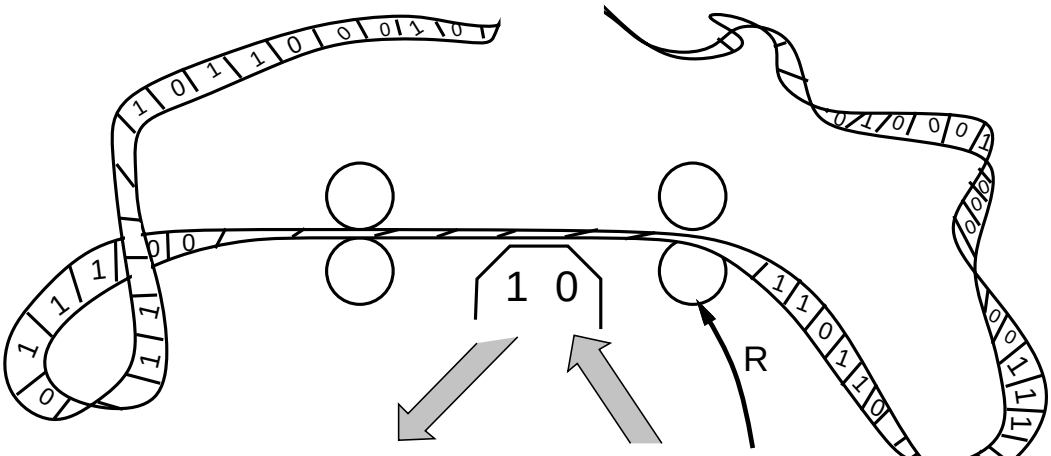
Alan Turingin tavoitteena oli selvittää vastaus nk. Hilbertin kymmenenteen ongelmaan:

Onko olemassa yleistä ratkaisualgoritmia, jolla voi ratkaista minkä tahansa matemaattisen ongelman ?

Turing onnistui kehittämään täysin yleisen laskentakoneen periaatteen, jolla jokainen ratkaistavissa oleva tehtävä voidaan periaatteessa mekanisoida. Toisaalta algoritmia mielivaltaisen loogisen ongelman ratkaisun löytämiseen ei Turingin koneella, eikä millään muullakaan tavalla saavuteta. Turingin todistus matemaattisten ongelmien yleiselle ratkeamattomuudelle liittyy itävaltalaisen Kurt Gödelin vuonna 1931 todistamaan tulokseen jonka mukaan

on olemassa tosia matemaattisia lauseita, joita ei kuitenkaan pystytä todistamaan tavanomaisia matemaattisia päättelysääntöjä käyttäen.

Vaikka tulos tuntuu masentavalta, niin maailmassa on myös hyvin paljon ongelmia, jotka ovat ratkaistavissa. Siten Turingin ideoima universaali laskentakone on osoittautunut käytännössä hyödylliseksi. Universaalisuus sanoo, että mikä tahansa tietokone voidaan toteuttaa Turingin koneella ja päinvastoin. Ehtona on kuitenkin, että laskenta-aikaan ja muistin määrään ei kiinnitetä huomiota. Kaikki nykyaikaiset tietokoneet voidaankin siis nähdä, jos niin halutaan, Turingin koneina. Niiden avulla voidaan ratkaista ja löytää ratkaisuja monenlaisiin käytännössä esiintuleviin matemaattis-loogisiin ongelmiin. Turingin teoreeman arvo on ymmärryksessä, että matemaattis-loogisen tietokoneen kyvyilläkin on rajat.



Tila	Syöte (luku)		Seuraava tila	Vaste (kirjoitus)	Nauhan siirto
0	0	→	0	0	R
0	1	→	13	1	L
1	0	→	65	1	R
1	1	→	1	0	R
2	0	→	0	1	R.STOP
2	1	→	66	1	L
3	0	→	37	0	R
259	1	→	0	0	R.STOP

Kuva 5.7: *Turingin koneen periaate: ääretön nauha, luku ja kirjoitusoperaatiot, sekä joukko sisäisiä tiloja.*

Turingin koneen perusperiaate on yksinkertainen, kuten käy ilmi kuvasta 5.7. Kirjallisuudessa koneesta on esitetty suuri määrä erilaisia variaatioita, joista seuraavassa esitetään yksi. Toiminnan kuvaamiseen tarvitaan seuraavat käsitteet.

- Joukko sisäisiä tiloja, joissa tehdään ensin lukuoperaatio ja tämän valitsemana kirjoitusoperaatio, nauhan siirtäminen ja seuraavaan tilaan siirtyminen.
- Äärettömän pitkä nauha, joka toimii muistina. Muistipaikoissa on joko 1 tai 0, (ja käytännössä myös erikoismerkki "blankko", joka osoittaa ne nauhan osat, jotka eivät ole käytössä). Muistipaikka voidaan lukea tai siihen voidaan kirjoittaa, kun lukupää on kyseisen muistipaikan kohdalla.
- Lukuoperaatio, jossa luetaan nauhalla oleva binäärinen merkki (1 tai 0).
- Kirjoitusoperaatio, jossa kirjoitetaan nauhalle merkki (1 tai 0).
- Nauhan siirto-operaatio, jossa siirretään nauhaa joko vasemmalle (L) tai oikealle (R).
- Lopetuskomento, STOP, jolloin Turingin kone pysähtyy ja vastaus voidaan lukea. Lopetuskomento on tärkeä, sillä tehtävä on ratkaistavissa ainoastaan, jos Turingin koneen suoritus tulee päättymään.

Näitä käyttäen yksi Turingin koneen komento on muotoa:

Tila_i, $\text{Luettu}_{(1/0)} \rightarrow \text{Tila}_{i+1}$, $\text{Kirjoita}_{(0/1)}$, $\text{Nauha}_{(L/R)}$

Esimerkiksi komentopari:

$0, 1 \rightarrow 13, 1, L$
$0, 0 \rightarrow 1, 0, R$

Tarkoittaa:

"Tilassa 0, jos luetaan 1 kirjoitetaan nauhalle 1, siirretään nauhaa vasemmalle (L), ja siirrytään tilaan 13."

"Tilassa 0, jos luetaan 0 kirjoitetaan nauhalle 0, siirretään nauhaa oikealle (R), ja siirrytään tilaan 1."

Jos kone pysäytetään, lisätään komennon loppuun STOP merkki, esim.

$259, 1 \rightarrow 0, 0, R.STOP$

On tärkeää huomata, että jokainen Turingin koneen tilataulukko voidaan koodata edelleen nollina ja ykkösinä. Tämä tehdään esimerkiksi siten, että kukin tila esitetään binäärijonona, jota seuraa syötteen arvo (0 tai 1), jota seuraavat siirtymien koodaukset (R=0, L=1, EI STOP=0, STOP=1). Näinollen jokainen mahdollinen

tilataulukko on itsessään binäärijono, eli kokonaisluku. Kun koodaus tehdään siten, että kaikki mahdolliset jonot ovat käytössä, niin jokainen jono (kokonaisluku) voidaan tulkita tilataulukoksi eli Turingin koneeksi: T_{Nro}. Tarkempi käsittely sivuutetaan tässä yhteydessä, mutta esimerkiksi Turingin koneet $0, 1, 2, \ldots, 12$ ovat:

T_0 : 0,0 → 0,0,R; 0,1 →0,0,R
T_1 : 0,0 → 0,0,R; 0,1 →0,0,L
...
T_{12} : 0,0 → 0,0,R; 0,1 →0,0,R; 1,0→0,0,R

Suurin osa numerokoodeista on koneen kannalta "järjettömiä", mutta toiset lukevat ja kirjoittavat nauhaa, muuntaen nauhalla olevia merkkejä toisiksi. Esimerkiksi T_3 ja T_{11} ovat järjellisiä, joskin hyvin vaatimattomia:

T_3 : 0,0 → 0,0,R; 0,1 →0,0,STOP

Turingin kone numero 3 siirtää nauhaa oikealle kunnes se kohtaa ykkösen, jonka se muuttaa nollaksi. Tämän jälkeen kone pysähtyy. Kone 11 on edellistä hölmömpi:

T_{11} : 0,0 → 0,0,R; 0,1 →0,1,STOP

Se pysähtyy kohdattuaan ensimmäisen ykkösen ja ei tee sen perusteella mitään.

Mielenkiintoiseksi ajatusleikki muuttuu, kun huomataan, että on olemassa universaali Turingin kone. On siis mahdollista kirjoittaa tilataulukko (binäärijono), joka ymmärtää nauhalla olevat merkit toisen koneen tilataulukoksi ja käyttäytyy aivan kuten nauhalle kirjoitettu "ohjelmoitu" Turingin kone käyttäytyisi. Tämän koneen tilataulukko on binäärijono, jonka pituus olisi, koneen tyypistä ja koodauksesta riippuen, lyhyimmillään 1-6 tämän luentomonisteen sivua, mikä tässä yhteydessä sivuutetaan. Lisäksi on selvää, että jos on olemassa yksi universaali Turingin kone, on olemassa muitakin.

Tämänkaltaista ohjelmoitavaa Turigin konetta kutsutaan yksinkertaisemmin (universaaliksi) tietokoneeksi. Kaikki tämän päivän tietokoneet kuuluvat tähän kategoriaan ja ovat (jos muistia on riittävästi) ohjelmoitavissa simuloimaan toisiaan.

Turingin kone on teoreettinen konstruktio, joka ei kerro miten tietokone saadaan järkevästi toteutettua. Kestikin yli kymmenen vuotta ennenkuin rakennettiin ensimmäinen universaali tietokone, EDSAC (1949)[1]. Suurimpana erona todellisessa tietokoneessa suhteessa Turingin koneeseen on muistin käsittely, josta seuraa myös monia vaatimuksia tiedon siirtoon ja tallentamiseen, minkä vuoksi joudumme tutustumaan mm. jaettuihin resursseihin ja tietokoneväyliin.

[1] Enismmäisenä tietokoneena pidetty ENIAC (1946) ei tosiasiassa ollut universaali.

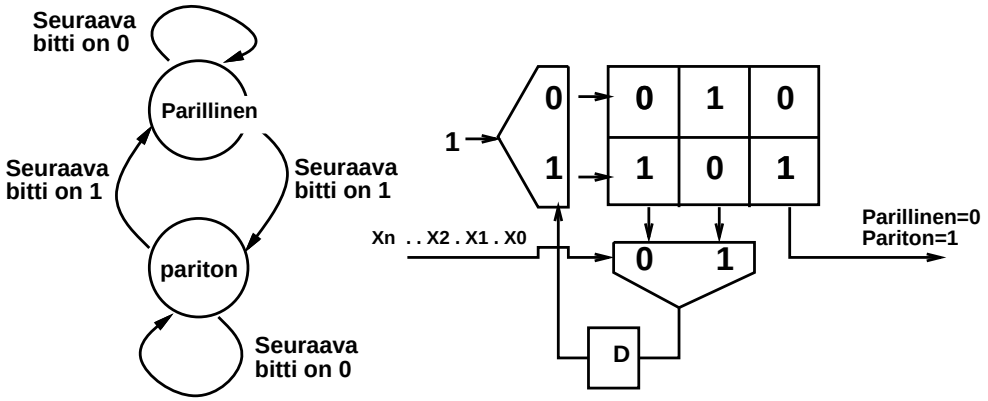
Esimerkkitehtävä: pariteettitarkistin

Tehtävänä on suunnitella automaatti, joka kertoo sarjamuotoisesti (bitti kerrallaan) syötetyn bittijonon (esimerkiksi 8 bitin sanan) pariteetin, eli onko 1-bittejä parillinen vai pariton määrä.

Esimerkkitehtävän vastaus

Tehtävä on varsin helppo, kunhan huomaa, että automaatti tarvitsee vain kaksi tilaa: parillinen ja pariton. Alussa automaatti on tilassa parillinen. Automaatti vaihtaa tilaansa aina kun vastaanotetaan 1-bitti.

Tila	Syöte	Vaste	Seuraava tila
0. (parillinen)	0	0 (parillinen)	0. (parillinen)
0. (parillinen)	1	0 (parillinen)	1. (pariton)
1. (pariton)	0	1 (pariton)	1. (pariton)
1. (pariton)	1	1 (pariton)	0. (parillinen)



Pariteettitarkistimen tilakaavio ja toteutus 2×3 *ROM:lla.*

Luku 6

Kommunikointi, jaetut resurssit ja väylät

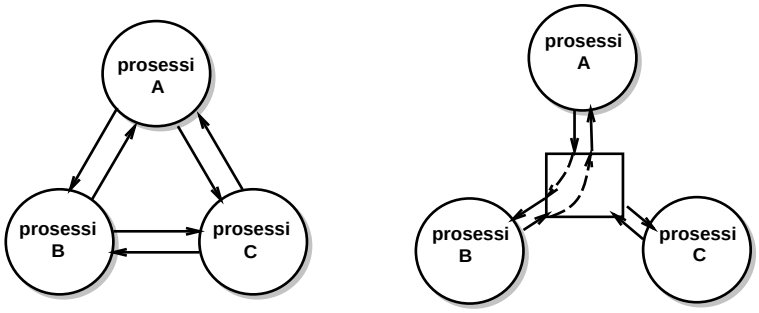
Kommunikointi on tiedonvälityksen perusta. Järjestelmien näkökulmasta se tapahtuu syöte- ja vastesignaalien muodossa. Sen merkitys digitaalisessa järjestelmässä on helppo käsittää analogiana ihmisten väliseen tiedonsiirtoon. Kommunikoinnin perusmekanismien ymmärtäminen tuleekin tietotekniikassa vastaan monella tasolla. Niin myös tässä monisteessa: käyttöjärjestelmien ja tietoliikenteen osalta luvuissa 9 ja 11, sekä tietokoneen sisäisissä toimintaperiaatteissa luvussa 7.

6.1 Jaetut resurssit

Yleisellä tasolla kommunikointi on itsenäisten rinnakkaisten prosessien, komponenttien, olioiden tai ohjelmien välistä tiedonsiirtoa. Keskeistä on, että tätä varten on olemassa mekanismi tai tietoväylä, jonka avulla informaatio siirretään. Tässä yhteydessä sivuutamme tiedonsiirron erikoiskysymykset ja keskitymme tarkastelemaan kommunikointia tietokoneen toteutuksen kannalta. Olennaisia seikkoja ovat tällöin käsitteet:

1. *Jaettu resurssi*, mikä tarkoittaa usean kommunikoivan prosessin (komponentin) yhteisesti käyttämää tiedonsiirtokanavaa.
2. *Prosessien synkronointi*, ei tapaa turvata jaetun resurssin käyttö niin, että viestit eivät mene sekaisin.

Jaetun resurssin käyttö on yleensä perusteltua, koska muuten kommunikoinnin toteuttamisen kustannukset käyvät liian kalliiksi. Tätä on havainnollistettu kuvassa 6.1, missä yhteisen tiedonvälityksen käyttö vähentää kommunikointikanavien määrää. Samalla ongelmaksi tulee resurssin jakaminen käytön kannalta järkevästi.



Kuva 6.1: *Kolmen itsenäisen prosessin kommunikointi joko suorilla yhteyksillä tai käyttäen jaettua resurssia.*

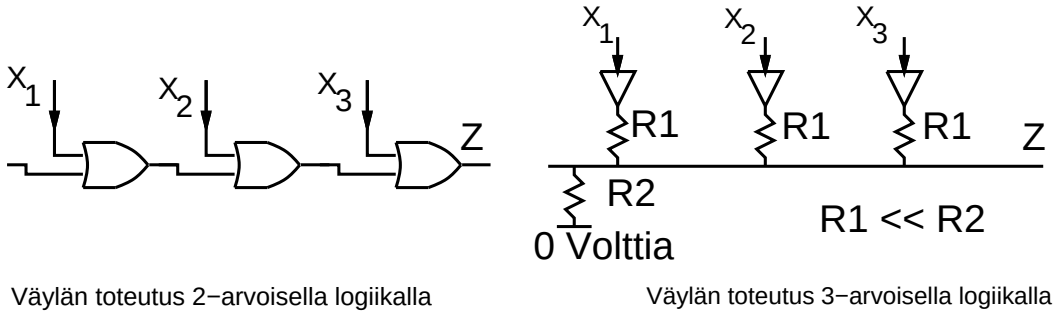
Rinnakkaisten prosessien synkronointiin on olemassa monia mekanismeja, jotka liittyvät läheisesti käyttöjärjestelmien suunnitteluun. Näillä rajataan tai ohjataan jaetun resurssin käyttöä sopivin kriteerein, esimerkiksi takaamalla, että kaikki kommunikointia tarvitsevat osapuolet pääsevät kommunikoimaan tietyn ajanjakson sisällä.

6.2 Tietokoneväylä

Tietokone olisi mahdollista rakentaa johdottamalla jokainen tietokonearkkitehtuurin komponentti suoraan toisiin komponentteihin. Käytännössä tämä on järjetöntä, sillä johdotusten määrä n:lle komponentille tulisi olemaan suuruusluokkaa n^2. Se muodostaisi nopeasti suurimman osan tietokoneen toteuttamisen kustannuksista, kun komponenttien määrä on kohtalaisen suuri. Erityisen hyödytöntä johdottaminen on von Neumann "perus"arkkitehtuurissa (EDVAC-arkkitehtuurissa), jossa vain kaksi komponenttia kommunikoi samanaikaisesti keskenään.

Tietokoneväylä on kahden tai useamman komponentin jakama kommunikointiresurssi, johon yksi komponentti voi kirjoittaa, ja johon kirjoitetun informaation voi lukea useampi komponentti. Väylän periaatteellisia ratkaisuja on useita. Eräänä yksinkertaisena vaihtoehtona sitä voitaisiin ajatella usean kirjoittavan komponentin

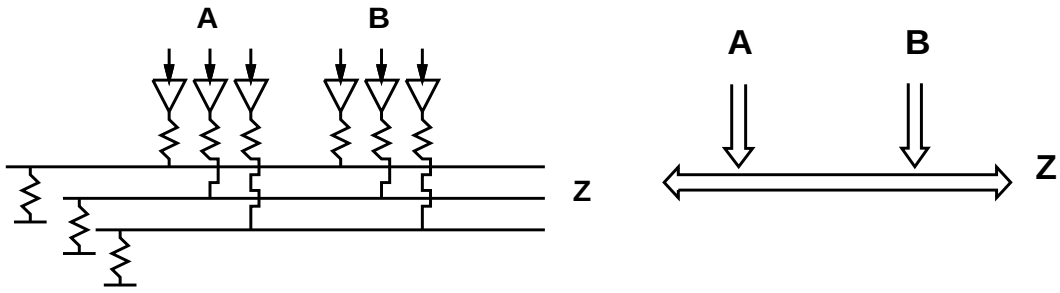
muodostamaksi TAI-piiriksi (kuva 6.2). Pitämällä huolta, että ainostaan yksi komponenteista kirjoittaa kerrallaan, tulee kirjoittavan komponentin antaman signaalin arvo (0 tai 1) signaalilinjan Z-arvoksi. Tämä voidaan johtaa edelleen useammalle komponentille syötteeksi.



Kuva 6.2: *Eräs yksinkertainen (mutta ei ainoa) väylän toteutusperiaate. Ilman yhtään 1:stä syötteissä, ohjautuu Z-signaali 0-tasoon. Yhdenkin syötteen ollessa 1-tasoinen, nousee signaalilinjan Z jännite 1-tasolle.*

Porttipiireillä rakennetun väylän ongelmana on, että uuden komponentin lisääminen vaatii aina signaalijohdon katkaisemisen. Tällöin lisälaitteiden esimerkiksi mikrotietokoneen grafiikkakortin lisääminen tulee huomattavan hankalaksi. Käytännössä väylällä käytetäänkin muusta logiikasta poikkeavia 3-arvoisia komponentteja, joilla on tilojen 0 ja 1 lisäksi tila "ei merkitystä". Näin signaalilinjalla oleva komponentti voi toimia myös kuin sitä ei olisi edes kytketty linjalle. Kuvan 6.2 oikeanpuoleinen versio lienee kaikkein yksinkertaisin toteutus. Siinä signaalilinja on kytketty 0-tasoon suuremmalla vastuksella ja diodeilla ohjattuihin tuloihin selvästi pienemmillä vastuksilla, jolloin toiminta on kuten TAI-piirillä.

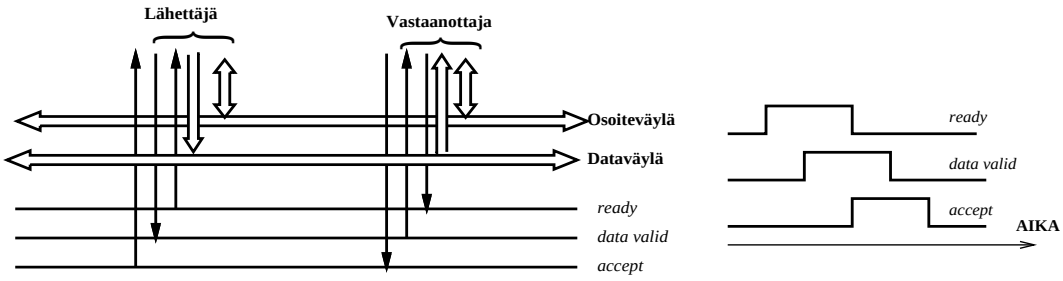
Väylien ensisijainen tarkoitus on siirtää useamman bitin mittainen data-sana tai osoite komponentilta toiselle. Tällöin signaalilinjoja on useampia yhdessä, mikä esitetään yleensä leveinä, vahvennettuina nuolina, kuten kuvassa 6.3. Nuolien suunta kertoo informaation siirtosuunnan, ja kaksisuuntainen nuoli tarkoittaa kaksisuuntaista (luku tai kirjoitus) liikennettä.



Kuva 6.3: *Usean bitin väylärakenteen esittäminen graafisesti.*

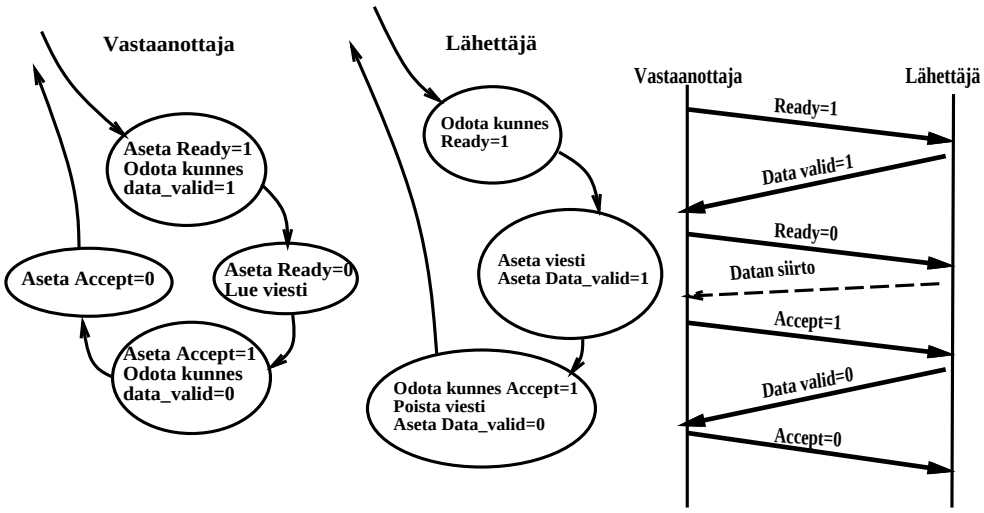
6.3 Kommunikointi ja kättelyprotokolla

Kuten edellä kerrottiin, vaatii jaettujen resurssien käyttö sitä käyttävien komponenttien synkronointia. Tässä yhteydessä käsittelemme aihetta varsin pintapuolisesti esittelemällä nk. kolmen signaalin kättelyprotokollan tietokoneväylän käyttöä varten. Väylää käyttää sekä informaation tuottaja että kuluttaja. Ohjaussignaalien avulla on tarkoitus synkronoida näiden välinen kommunikointi siten, että informaatio siirtyy ilman ongelmia paikasta toiseen. Tarkoitusta varten tarvitaan kolme uutta signaalilinjaa *ready*, *data valid* ja *accept* (kuva 6.4).



Kuva 6.4: *Kolmen signaalilinjan kättelyprotokolla toteutettuna fyysisillä signaalilinjoilla.*

Kommunikoinnin vaiheittaisen etenemisen voi esittää monella tavalla. Kuvan 6.4 oikeassa laidassa on kommunikointi esitetty signaalilinjojen ajoitusten avulla, mikä on tarkemmin esitetty taulukon 6.1 vaiheittaisena "hand-shake"-algoritmina. Saman voi esittää myös kahtena vuorovaikutteisena automaattina kuvan 6.5 mukaisesti.



Kuva 6.5: *Kommunikointi kahtena vuorovaikutteisena Mooren automaattina ja signaalien tapahtumasekvenssikaaviona.*

6.4 Kommunikointi usean lähettäjän kanssa

Kun lähettäjiä on useampia, tarvitaan lisäksi tieto siitä mikä lähettäjä on kyseessä. Nyt toimintaprotokolla riippuu siitä määrääkö vastaanottaja heti alkuvaiheessa mikä lähettäjistä on valittu, vai pyrkivätkö lähettäjät kilpailemaan vastaanottajan huomiosta. Ensimmäinen tapaus on yksinkertaisempi, mutta kummassakin tapauksessa tarvitaan datan siirtoon tarkoitetun väylän lisäksi osoiteväylä, johon vastaanottaja tulee asettamaan kommunikointiin luvan saaneen lähettäjän osoitteen. Toisessa tapauksessa (kuva 6.6) tarvitaan lisäksi uusi ohjaussignaali *request to send* (lähetyspyyntö).

Mikäli vastaanottaja määrää lähettäjän, tapahtuu se siten, että vastaanottaja asettaa lähettäjän osoitteen osoiteväylälle. Tämän jälkeen vastaanottaja havaitsee oman osoitteensa osoiteväylällä ja kommunikointi tapahtuu kuten aiemminkin *hand-shake*-algoritmin määräämänä.

Mikäli vastaanottaja ei tiedä mikä lähettäjistä olisi valmis lähettämään, on periaatteessa kaksi mahdollisuutta.

Pollaus, missä vastaanottaja kierrättää vuoron perään kaikkien mahdollisten lä-

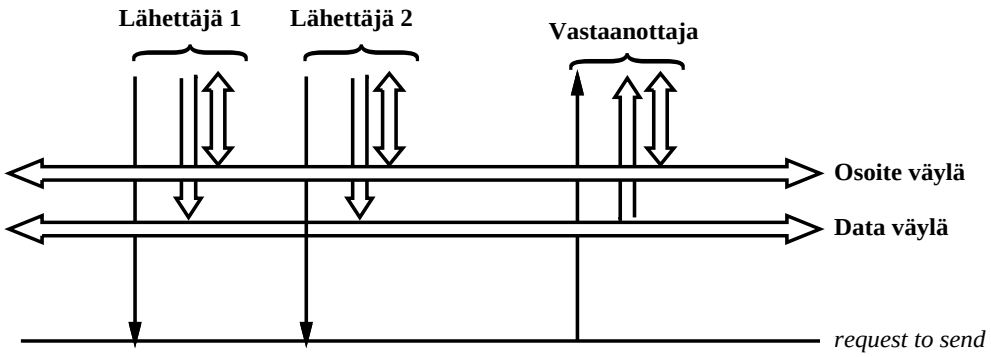
Algoritmi: hand-shake

1. *Vaihe: Valmis vastaanottamaan.* Alussa vastaanottaja asettaa signaalilinjan *ready=1* tiedoksi siitä, että se on valmis vastaanottamaan väylälle kirjoitettavan tiedon. Tämä tarkoittaa myös sitä, että väylä on (sen tulee olla) täysin lähettäjän ohjattavissa.
2. *Vaihe: Data asetettu.* Lähettäjä on huomannut signaalin *ready=1* ja asettaa väylän signaalien arvoiksi (asettaa väylälle) datasanan. Kun signaalitasot ovat asettuneet, lähettäjä asettaa signaalin *data valid=1* tiedoksi, että data on luettavissa väylältä.
3. *Vaihe: Luku alkaa.* Vastaanottaja on huomannut signaalin *data valid=1* ja alkaa lukea väylän signaalien arvoja. Alussa se asettaa *ready=0* tiedoksi, että se ei ole valmis uuteen kommunikointiin.
4. *Vaihe: Luku loppuu.* Lukutoimituksen tehtyään vastaanottaja asettaa tiedon siirtotapahtuman päättymisestä signaalilla *accept=1.*
5. *Vaihe: Lähettäjän lopetus.* Lähettäjä havainnoi signaalin *accept=1* merkiksi, että dataa ei tarvitse pitää väylällä. Se asettaa *data valid=0*, kun väylän arvojen asetus loppuu.
6. *Vaihe: Vastaanottajan lopetus.* Vastaanottaja havainnoi *data valid=0* signaalin ja asettaa *accept=0* tiedonsiirron lopullisen päättymisen merkiksi.

Taulukko 6.1: Algoritmi lähettäjän ja vastaanottajan kommunikointiin tietokoneväylällä.

hettäjien osoitteita osoiteväylällä, tarkastaen jokaisen osoitteen jälkeen onko signaali *request to send* asetettu ykköseksi. Apuna lähettäjä voi käyttää *ready* signaalia, joka tuottaa pulssin uuden osoitteen vaihtuessa. Kun lähettäjä, joka on valmis lähettämään, havaitsee signaalin *ready* ja oman osoitteensa, se asettaa signaalin *request to send=1.* Tämän havaittuaan vastaanottaja lopettaa pollauksen ja aloittaa *hand-shake*-algoritmin avulla tapahtuvan tiedonsiirron.

Keskeytys (kilpailu). Tässä lähettäjät kilpailevat vastaanottajan huomiosta lähettämällä signaalin halustaan kommunikoida. Jos signaalikanavat ovat jaettuja, on mahdollista että kutsut tapahtuvat useammasta osoitteesta saman aikaisesti. Tällöin tapahtuu yhteentörmäys (collision), ja kommunikointi jou-



Kuva 6.6: *Usean lähettäjän ja yhden vastaanottajan kommunikointi sekä lähettäjän valintaa tukeva ohjaussignaali.*

dutaan mahdollisesti peruuttamaan. Yleensä tietokoneissa on erillinen keskeytyslogiikka ulkopuolelta tulevien pyyntöjen hallintaan, jossa eri laitteilla voi olla eri prioriteetti kommunikointia varten. Yhteentörmäyksen tapahtuessa suurimman prioriteetin laite pääsee kommunikoimaan.

Luku 7

Von Neumann-arkkitehtuuri

Tämän luvun otsikko on hieman harhaanjohtava. Vaikka John von Neumann on vaikuttanut suuresti nykyaikaisen tietokoneen rakenteeseen, syntyi perusarkkitehtuuri pitkälti ryhmätyönä EDVAC-tietokonetta suunniteltaessa. Von Neumannin kunniaksi on toki luettava, että hän määrätietoisesti selkeytti ja kiteytti ratkaisut niihin perusperiaatteisiin, jotka me nykyään tunnemme. Koska tietokonearkkitehtuuri tunnetaan kaikkein laajimmin von Neumann-arkkitehtuurina, käytämme sitä myös tässä yhteydessä.

Tiivistettynä von Neumann-arkkitehtuuri jakautuu seuraaviin komponentteihin:

1. *Kontrolliyksikkö* (CU, control unit, sequentializer) on automaatti, joka toteuttaa koneen ymmärtämät primitiiviset käskyt, konekäskyt, joiden avulla mielivaltaisia laajempia ohjelmia pystytään toteuttamaan. Kontrolliyksiköstä lähtevät ohjaussignaalit määräävät muiden keskeisten yksikköjen välisen tiedonsiirron ja valitsevat suoritettavat operaatiot.

2. *Aritmeettis-looginen yksikkö* (ALU) sisältää toteutukset tärkeimmistä toiminnoista, kuten yhteen- ja vähennyslaskut, loogiset AND-/, OR-/, ja NOT-operaatiot, jne.

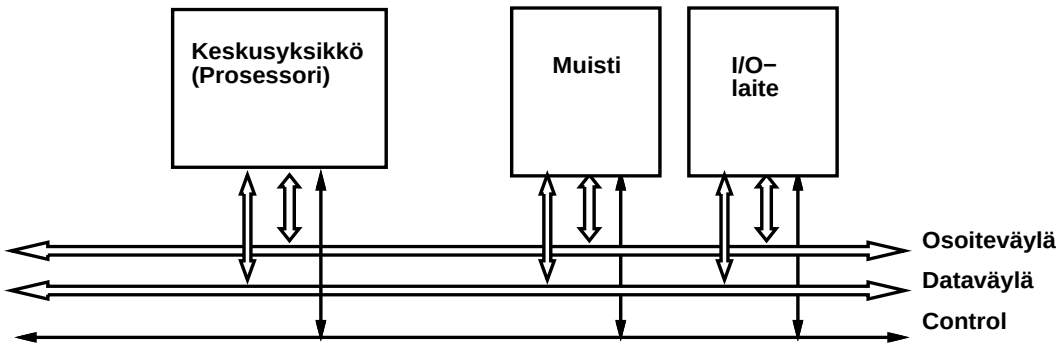
3. *Rekisterit* ovat nopeaa muistia kontrolliyksikön ja aritmeettis-loogisen yksikön välittömässä läheisyydessä. Niihin tallennetaan loogisten operaatioiden syötteet ja vasteet, ennen ja jälkeen muuhun muistiin siirtämistä.

4. *Muisti ja I/O* (input-output)-laitteet kattavat koneen nk. I/O- ja muistiavaruuden. Se on joukko osoitteita, joihin on sijoitettu toimintalaitteiden rekiste-

rejä tai muistia. Tietokoneen toiminta kiteytyy suuressa määrin hakusilmukkaan, jossa kontrolliyksikkö siirtää muistista tietoa rekistereihin, ja valitsee seuraavan toimintonsa tämän muistista siirretyn informaation määräämänä.

7.1 Esimerkkiarkkitehtuuri

Tarkastelemme von Neumann-arkkitehtuurin tietokonetta esimerkin kautta. Koneen rakenne on esitettu kuvissa 7.1 ja 7.2. Siinä on data-, osoite- sekä kontrolliväylät muistin ja oheislaitteiden liittämistä varten (kuva 7.1). Koneen sananpituus on 8 bittiä[1], mikä tarkoittaa, että komennot on koodattu 8 bitin mittaisiin sanoihin, mikä on myös dataväylän leveys. Osoiteväylän leveys on yleensä dataväylää suurempi. Esimerkissämme se on 16 bittiä.



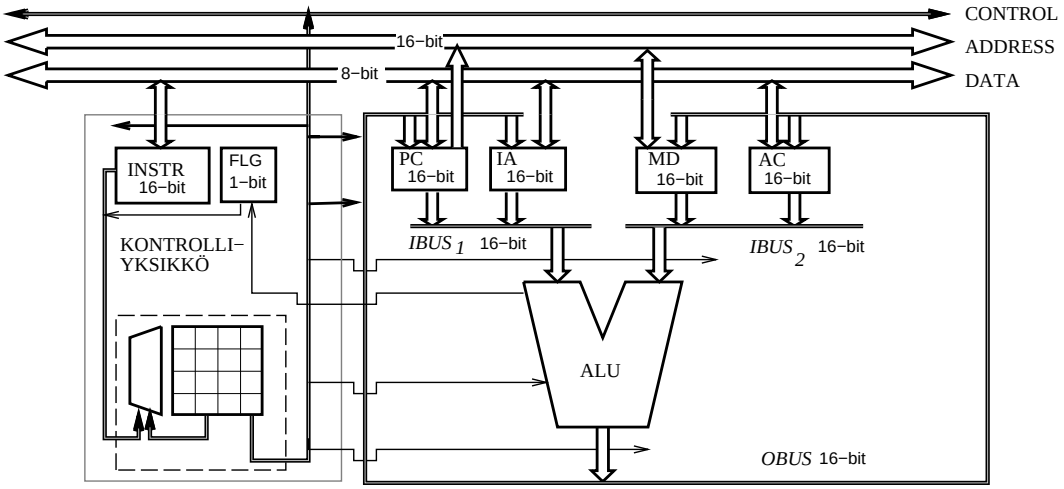
Kuva 7.1: *Tietokone koostuu väylällä toisiinsa kytketyistä keskusyksiköstä, muistista sekä I/O-laitteista.*

Koneen keskusyksikkö (kuva 7.2) huolehtii informaation siirron ohjauksesta muistin ja oheislaitteiden välillä. Se kykenee myös suorittamaan joukon aritmeettis-loogisia toimenpiteitä, kuten yhteenlaskua ja Boolen funktioita sekä siirtelemään tietoa rekisterien välillä. Keskusyksikkö jakaantuu edelleen kuvan 7.2 mukaisesti *aritmeettis-loogiseen yksikköön* (ALU), *kontrolliyksikköön* ja joukkoon *rekisterejä*. Lisäksi siinä on muutama sisäinen väylä rekisterien ja aritmeettis-loogisen yksikön väliseen kommunikointiin.

[1] Alan kehityksestä mainittakoon, että ensimmäisessä Intelin mikroprosessorissa sananpituus oli 4 bittiä, ja moderneissa tietokoneissa se on 64 bittiä.

Koneemme kontrolliyksikkö on automaatti, joka vaihtaa tilaansa edellisen tilan sekä *INSTR*- ja *FLG*-rekistereistä saamiensa syötteiden määräämänä. Se ohjaa sekä kontrolliväylän signaaleja että tiedonsiirtoa rekisterien ja väylien välillä. Lisäksi se valitsee kulloisenkin operaation, jonka ALU suorittaa, esimerkiksi kahden rekisterin sisällön yhteenlaskun tai niiden bittien välisen AND-funktion.

Koneen rekisterit ovat usein yhtä monen bitin levyisiä kuin koneen osoiteväyläkin, eli tapauksessamme rekistereihin mahtuu yksi 16 bitin sana. Lisäksi esimerkkimme keskusyksikön kolme sisäistä väylää $IBUS_1$, $IBUS_2$ ja *OBUS* ovat myöskin 16 bittiä leveitä. Edellisten lisäksi koneessa on tyypillisesti muutamia 1 bitin erikoisrekisterejä, lippuja, joiden arvo riippuu mahdollisesti edellisen aritmeettis-loogisen operaation tuloksesta. Esimerkissämme tätä edustaa 1-bitin rekisteri-*FLG*.



Kuva 7.2: *Keskusyksikön rakenne.*

Koneen ohjelmoinnin kannalta on oleellista, miten sen rekisterit on organisoitu. Esimerkissämme rakenne on varsin helppokäyttöinen, koska kaikki rekisterit on kytketty siten, että niiden ja dataväylän välillä pystyy siirtämään suoraan informaatiota. Rekistereillä on usein ennaltamäärättyjä käyttötarkoituksia, jotka esimerkkimme tapauksessa on esitelty tarkemmin taulukossa 7.1. Formaalisti rekisterien ja väylien käytön voi esittää muuttujien sijoituksena. Esimerkiksi kaksi operaatiota

```
      AC= OBUS
IA[0..7]= DATA,
```

tarkoittavat *OBUS* väylällä olevan sanan siirtoa *AC*-rekisteriin ja dataväylällä olevan 8 bitin sanan siirtoa *IA*-rekisterin bittipaikkoihin $0 \ldots 7$.

INSTR-rekisteri on kontrolliyksikön sisäisessä käytössä. Siihen tallennetaan suoritusvaiheessa oleva komento.
PC-rekisteri eli "Program Counter" on viittaus muistiosoitteeseen, josta löytyy kulloinkin suoritettava (tai seuraava) käsky. PC-rekisteri on kaikkein keskeisin käskynhaun osalta.
MD-rekisteri on tarkoitettu osoitteen väliaikaiseen tallentamiseen tai käsittelyyn. Muuten sitä voidaan käyttää yleiskäyttöisenä rekisterinä.
IA-rekisteri on yleiskäyttöinen. Siihen tallennetaan tyypillisesti aritmeettis-loogisen operaation parametri.
AC-rekisteri on yleiskäyttöinen. Tyypillisesti aritmeettis-loogisen operaation tulos tallennetaan siihen.
FLG-rekisteri on yhden bitin kokoinen ja saa arvonsa suoraan ALU:n tekemän laskutoimituksen perusteella siten, että se sisältää arvon 1, jos ALU:n tekemän laskutoimituksen arvo on nolla, muuten FLG=0. Rekisterin avulla voidaan toteuttaa ehdollisia toimintoja, sillä se on kytketty suoraan kontrolliyksikön syötteisiin.

Taulukko 7.1: Esimerkkiarkkitehtuurin keskusyksikön rekisterien merkitys.

Esimerkkimme aritmeettis-loogiseen yksikköön (ALU) informaatio voi siirtyä samanaikaisesti $IBUS_1$ ja $IBUS_2$ väyliltä, joihin informaatio saadaan *PC*- ja *IA*-rekistereistä ja vastaavasti *AC*- ja *MD*-rekistereistä. ALU:sta saatavan operaation tulos voidaan johdottaa mihin tahansa edellämainituista rekistereistä *OBUS*-väylän välityksellä.

7.2 Osoiteavaruus (muistiavaruus, I/O-avaruus)

Tietokoneen osoiteavaruus (muistiavaruus) tarkoittaa niiden osoitteiden joukkoa, joihin koneella pystyy viittaamaan. Nykyaikaisissa koneissa osoitteiden joukko on usein jaettu etukäteen, mahdollisesti jopa käyttäen eri väyliä, erikseen ulkopuolisille laitteille, datamuistille ja ohjelmamuistille. Esimerkkimme tapauksessa kaikki osoitettavat laitteet ja muistipaikat on sijoitettu samaan osoiteavaruuteen. Käytännössä

osoitteen bittien määrä onkin tärkeämpi kuin koneen sananpituus, sillä lyhyt sananpituus ei rajoita koneen toimintaa muuten kuin toteutuksen nopeuden suhteessa sen kalleuteen[2]. Osoiteavaruus määrää suoraan kuinka paljon muistia tai oheislaitteita voidaan koneeseen laittaa, kuten taulukosta 7.2 käy ilmi. Moderneissa koneissa osoiteavaruus on 32:sta 64:ään bittiä.

Osoitteen koko	Osoitteiden (muistipaikkojen) määrä	epävirallinen lyhenne
8 bittiä	256	-
10 bittiä	1024	1 Kilo
16 bittiä	65536	64 Kilo
20 bittiä	1048576	1 Mega
30 bittiä	1073741824	1 Giga
32 bittiä	4294967296	4 Giga
40 bittiä	1099511627776	1 Tera
50 bittiä	1125899906842624	1 Peta
60 bittiä	1152921504606846976	1 Exa
64 bittiä	18446744073709551616	16 Exa

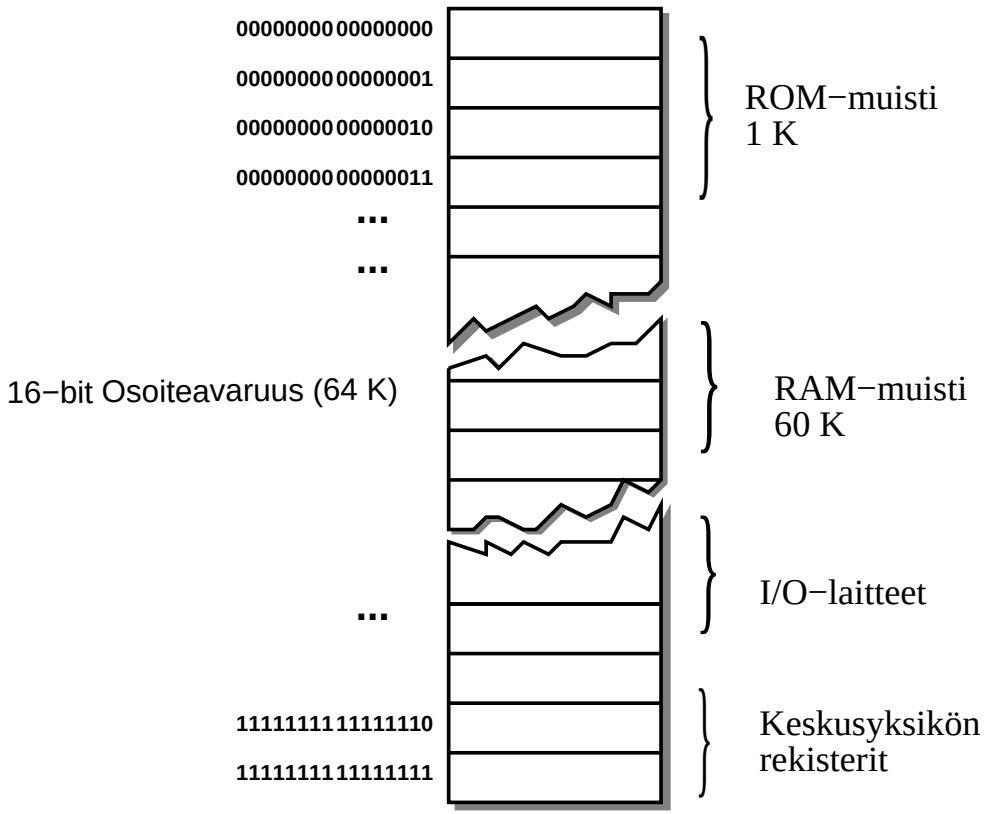
Taulukko 7.2: Mahdollisten muistipaikkojen määrä eri osoitteen pituuksilla.

Osoiteavaruuteen (kuva 7.3) sijoitetaan keskusyksikön kanssa kommunikoivat muut tietokoneen osat. Siinä on yksilöllinen osoite jokaiselle tietokoneen komponentille, jota voidaan osoittaa väylän avulla. Näinollen tilaa täytyy olla jokaiselle muistipaikalle sekä oheislaitteen rekisterille, johon informaatiota voidaan siirtää. Käytännössä myös osa keskusyksikön rekistereistä on sijoitettu osoiteavaruuteen, mikä helpottaa tiettyjen ohjelmointitehtävien tekemistä.

On tyypillistä, että osoitteessa 0 on ROM-muistia ja tietokone on suunniteltu siten, että käynnistyksen jälkeen ensimmäinen käsky suoritetaan juuri tästä osoitteesta. Mikrotietokoneiden omistajat tuntevat nämä ensimmäiset käskyt BIOS-ohjelmana, minkä tarkoitus on valmistaa kone käyttöjärjestelmä ohjelman suorittamista varten. Ennalta ohjelmoidun ROM-muistin lisäksi tulee RAM-muistia ohjelmien ja datan säilytystä varten, tyypillisesti ROM-muistin jatkeena. Monesti (mutta ei aina) I/O-laitteet ja keskusyksikön rekisterit sijoitetaan muistin toiseen äärilaitaan, jossa ne ovat ikäänkuin paremmassa turvassa muistiavaruutta käsitteleviltä ohjelmilta.

Edelläkerrotun lisäksi nykyaikaisissa tietokoneissa on monia kehittyneitä muistinhallintaominaisuuksia. Esimerkiksi prosessoritasolla voidaan määrätä osia osoiteava-

[2]Vielä nykyäänkin käytetään 1 bitin sananleveyden prosessoreja tietyissä erikoissovelluksissa.



Kuva 7.3: *Esimerkki 16 bitin osoiteavaruuden sijoittelusta.*

ruudesta suojatuksi, jolloin virheellinen (tai luvaton) ohjelma ei pääse muuttamaan koneen kannalta elintärkeitä tietoja (rekisterejä, ohjelmakoodeja).

7.3 Tietokoneen käskykanta

Seuraavassa esitellään kuvitteellisen tietokoneen käskykanta. Vaikka käskykanta on esitetty kokonaisuudessaan, ei tässä yhteydessä ole välttämätöntä sisäistää jokaisen käskyn olemusta. Tärkeämpää on saada käsitys siitä miten käskyn suoritus tietokoneessa tapahtuu, siis ymmärtää toiminta ainakin yhden käskyn osalta.

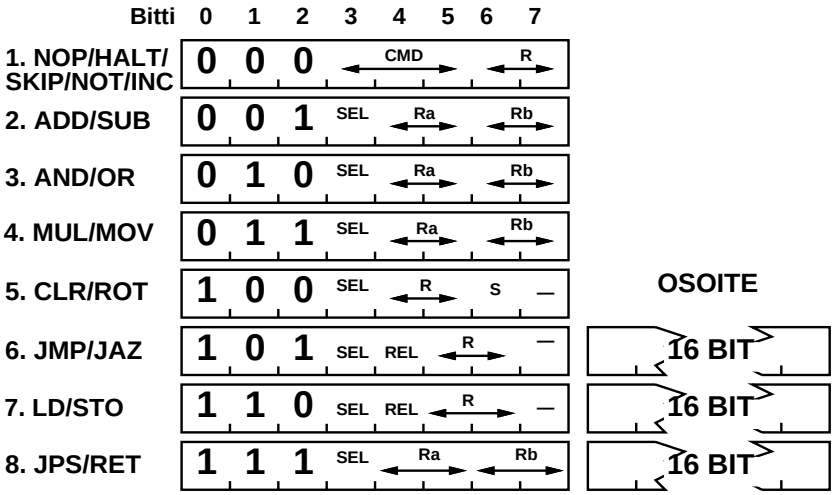
Tietokoneen kontrolliyksikköön on rakennettu joukko toimintasekvenssejä, jotka yksikkö suorittaa automaattisesti käskyrekisterin *INSTR* ja lippurekisterin *FLG* sisällön ohjaamana. Toimintasekvenssien joukko muodostaa koneen *käskykannan*, missä yksittäisen käskyn (komennon) avulla määrätään jokin perusoperaatio, kuten kahden rekisterin sisällön yhteenlasku. Kullakin käskyllä on oma yksilöllinen tunnisteensa, joka on koodattu osaksi käskysanaa. Tunnisteen lisäksi käskyyn voi liittyä joukko parametreja, jotka kertovat mistä informaatio siirretään ja minne se tallennetaan. Esimerkkimme koneessa käskykanta on äärimmäisen yksinkertaistettu. Peruskoodeja on ainostaan kahdeksan, mutta useimmat niistä ovat todellisuudessa samaan käskysanaan pakattuja erityyppisiä komentoja. Näin erilaisia konekäskyjä on kaikkiaan 23 kappaletta ja ne on esitetty kokonaisuudessaan taulukoissa 7.4, 7.5 ja 7.6.

7.3.1 Käskyjen koodaus

Jotta koneen sananpituutta voitaisiin käyttää mahdollisimman tehokkaasti, on käskyt tapana koodata tiivistettyyn muotoon. Esimerkillemme tämä koodaus on esitetty kuvassa 7.4. Useimmat käskysanoista ovat 8 bittiä, eli ne mahtuvat sellaisenaan koneen dataväylälle. Kolme viimeistä koodia esiintyy kahdessa muodossa, joista toinen on 8 bittiä ja toinen 8+16 bittiä. Useamman sanan käsky vaatii luonnollisesti pidemmän suoritusajan, koska sen haku muistista ja sinne kirjoitus vaativat useamman peräkkäisen vaiheen.

Edistyneimmille lukijoille huomautettakoon, että yksinkertainen koneemme ei ole pinokone, eli pinoa (STACK) ei ole toteutettu laitteistotasolla. Käytännössä ero näkyy lähinnä aliohjelmakutsujen toteutuksessa konekielitasolla.

Useissa käskyissä käytetään parametreina yhtä tai useampaa rekisteriä, missä sym-



Kuva 7.4: *Esimerkki käskyjen koodauksesta.*

bolit R, Ra ja Rb edustavat mitä tahansa rekistereistä IA, PC, MD ja AC. Nämä on koodattu kuvan 7.4 esittämillä kahdella biteillä taulukon 7.3 mukaisesti.

rekisteri R, Ra tai Rb	**bitti** x_0x_1
IA	0 0
PC	0 1
MD	1 0
AC	1 1

Taulukko 7.3: Rekisterin tunnistimien koodaus kahdella bitillä.

7.4 Käskyjen haku ja suoritus

Von Neumann (EDVAC)-tyyppisen tietokoneen toiminta kiteytyy käskynhakusekvenssiin, joka on toteutettu keskusyksikön sisäisenä automaattina. On selvää, että edellä esiteltyjen toimintojen määräämän tilakoneen toteutus on huomattavasti monimutkaisempi, kuin luentomonisteessa aiemmin esitelty juoma-automaatti. Tilakoneen perusidea, eli käskynhakusekvenssi on kuitenkin suhteellisen helposti ku-

vattavissa muutamalla tilalla, mikä antaa hyvän pohjan tietokoneen toimintojen ymmärtämiselle.

Yleinen käskynhakusekvenssi muistista keskusyksikköön (kontrolliyksikön automaattisesti suorittamana) on seuraava:

i) Hae komento: Komento on muistissa ohjelmarekisterin, PC, osoittamassa muistipaikassa, josta se haetaan väylän kautta INSTR-rekisteriin.
ii) Suorita komento: Nyt INSTR-rekisterin sisältö ohjaa kontrolliyksikön toimintaa, esimerkiksi saaden aikaan tiedonsiirtoa rekisterien välillä, tai laskutoimituksen, jossa rekistereissä oleva informaatio lasketaan yhteen aritmeettis-loogisessa yksikössä ja tallennetaan uudestaan johonkin rekisteriin.
iii) Seuraava komento: Ohjelmarekisteri siirretään osoittamaan muistipaikkaan, josta seuraava komento löytyy, tyypillisesti PC=PC+1.

Kun väylärakenne on tiedossa, voidaan edellinen esittää yksityiskohtaisemmin. Seuraavassa on pääpiirteissään kuvattu esimerkkikoneemme eri yksikköjen välinen kommunikointi käskynhakusekvenssin aikana.

1. Aseta: ADDRESS-väylä = PC:n arvo.
2. Aseta: CONTROL-väylälle *Ready=1*.
3. Odota kunnes: CONTROL-väylällä *data valid=1*.
4. Aseta: CONTROL-väylälle *Ready=0*.
5. Aseta: rekisteri INSTR[*0-7*]=DATA-väylän sisältö.
6. Aseta: CONTROL-väylälle *accept=1*.
7. ... jatko riippuu INSTR[*0-7*] rekisterin bittien arvoista.
8. Aseta: PC=PC+1.

Vaiheet 1 . . . 3 voidaan tulkita seuraavasti. Ennen käskyn suoritusta sen sijainti muistissa (osoite) on tallennettuna ohjelmalaskuriin (PC). Ensimmäiseksi kontrolliyksikkö asettaa kyseisen osoitteen osoiteväylälle ja kertoo muistipiirille kontrollisignaalien avulla, että dataväylä on vapaa ottamaan vastaan kyseisen muistiosoitteen sisällön.

Vaihe 4 on muistipiirin suorittama. Muistipiiri asettaa osoitteesta löytyvän datan, joka on haettavan käskyn 8 ensimmäistä bittiä, dataväylälle. Tämän jälkeen muistipiiri kertoo, että luku on asetettu väylälle.

Vaiheet 5 ja 6 vastaanottavat käskyn. Kontrolliyksikkö siirtää dataväylän sisällön INSTR rekisterin 8 ensimmäiseen bittiin ja kertoo muistipiirille, että muistipaikan luku on suoritettu.

Vaihe 7 on käskyn varsinainen suoritus. Toimintasekvenssin jatko riippuu siitä minkälainen komento on kyseessä (INSTR-rekisterin bittien 0-7 määräämänä).

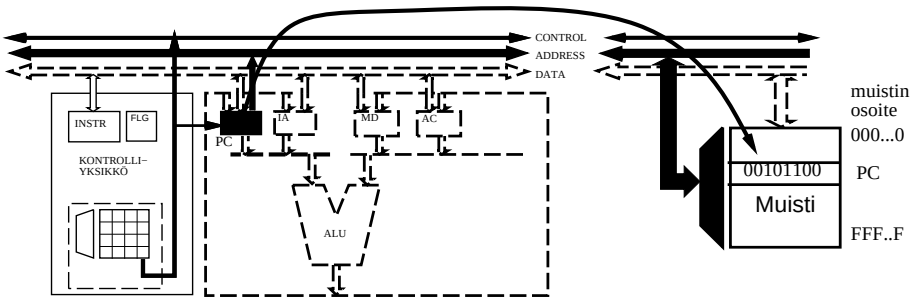
Lopuksi ohjelmalaskuri (PC) asetetaan osoittamaan seuraavaa muistissa olevaa käskyä.

7.5 Esimerkki: ADD-käskyn suoritus

Tarkastelemme esimerkin vuoksi käskyä ADD: AC=AC+IA, joka löytyy muistista PC-rekisterin soittamasta paikasta M[PC]. Muistipaikassa oleva käsky on on koodattu kahdeksalla bitillä muotoon `00101100`, mikä siis tarkoittaa esitystä

$$AC = AC + IA \quad \equiv \quad \underbrace{\texttt{0010}}_{ADD}\,\underbrace{\texttt{11}}_{AC}\,\underbrace{\texttt{00}}_{IA}.$$

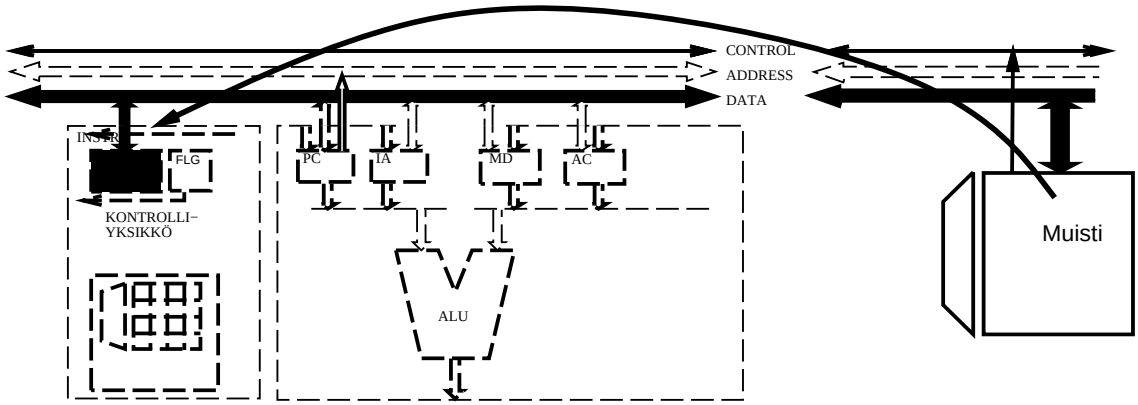
Käskyn ensimmäinen vaihe on sen hakeminen INSTR-rekisteriin. Osoite, eli PC:n arvo kerrotaan muistipiirille kuvan 7.5 mukaisesti.



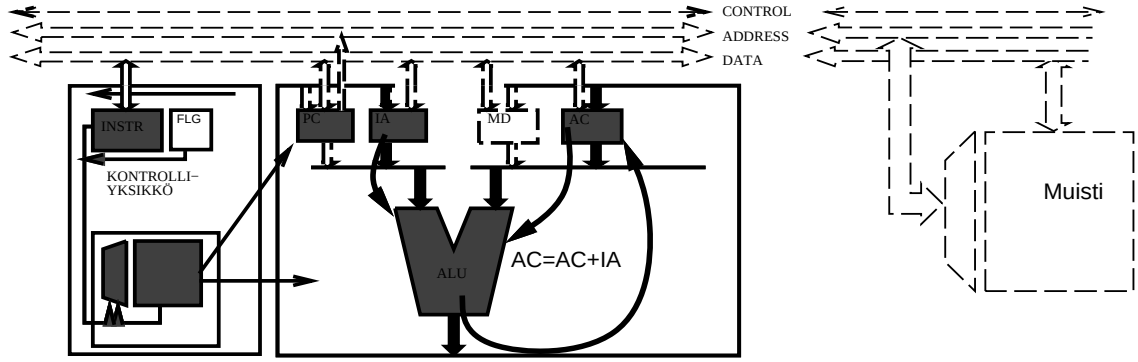
Kuva 7.5: *Käskyn osoite ohjelmalaskurista osoiteväylälle ja edelleen muistiin.*

Seuraavassa vaiheessa muistissa oleva käsky siirretään dataväylälle ja siitä edelleen INSTR-rekisteriin kuvan 7.6 mukaisesti.

Tämän jälkeen kontrolliyksikkö toimii INSTR-rekisterin sisällön ohjaamana ja suorittaa automaattisesti laskuoperaation ohjaamalla ALU:n toimintaa ja rekisterien välistä tiedonsiirtoa. Tämä on esitettynä kuvassa 7.7. Operaation jälkeen ohjelmalaskuria kasvatetaan yhdellä ohjelma-askeleella.



Kuva 7.6: *Käskyn ADD haku muistista dataväylälle ja siirto INSTR rekisteriin*



Kuva 7.7: *Yhteenlaskuoperaation suoritus ja ohjelmalaskurin (PC) arvon kasvattaminen.*

7.6 Esimerkkikoneen käskykanta

Taulukoissa 7.4, 7.5 ja 7.6 on esitetty koneemme täydellinen käskykanta, mikä on tarkoitettu lähinnä täydentäväksi materiaaliksi. Käskykantaan on valittu tyypilliset käskyt, jotka löytyvät pienin variaatioin lähes jokaisesta prosessorista.

Komento	$b_0b_1b_2b_3$	b_4b_5	b_6b_7	selitys
1. NOP	0000	00	-,-	*NO OPERATION, käsky alkaa ja päättyy välittömästi.*
2. HALT	0000	01	-,-	*STOP, käsky alkaa muttei koskaan pääty.*
3. SKIP	0000	10	-,-	*seuraavan käskyn yli hypätään, (PC=PC+1).*
4. NOT R	0000	11	R	*Rekisterin R negaatio:* $R[i] = \overline{R[i]}$.
5. INC R	0001	00	R	*Kasvata rekisterin R arvoa yhdellä: R=R+1.*
6. ADD Ra,Rb	0010	Ra	Rb	*Yhteenlasku rekistereille Ra ja Rb: Ra=Ra+Rb.*
7. SUB Ra,Rb	0011	Ra	Rb	*Vähennyslasku rekistereille Ra ja Rb : Ra=Ra-Rb.*
8. AND Ra,Rb	0100	Ra	Rb	*Looginen AND rekistereille Ra ja Rb :* $Ra[i]=Ra[i] \wedge Rb[i]$.
9. OR Ra,Rb	0101	Ra	Rb	*Looginen OR rekistereille Ra ja Rb :* $Ra[i]=Ra[i] \vee Rb[i]$.
10. MUL Ra,Rb	0110	Ra	Rb	*Rekisterien Ra ja Rb kertolasku : Ra=Ra * Rb.*
11. MOV Ra,Rb	0111	Ra	Rb	*Rekisterien kopiointi : Ra=Rb.*
12. CLR R	1000	R	-,-	*Nollaa rekisteri R : R=0.*
13. ROT R,S	1001	R	S,-	*Kierrä rekisteriä R yksi bitti vasempaan R[i]=R[i+1], R[15]=R[0], jos S=0 tai oikeaan R[i]=R[i-1], R[0]=R[15], jos S=1.*

Taulukko 7.4: Esimerkkiarkkitehtuurin peruskäskyt. Rekisterien koodaus kahdella bitillä noudattaa taulukkoa 7.3.

Komento	$b_0b_1b_2b_3b_4$	b_5b_6	b_7	selitys
14. JMP <os>	10100	-,-	-	*Absoluuttinen hyppy osoitteeseen: PC=<os>.*
15. JMP R	10101	R	-	*Hyppy rekisterin määräämään osoitteeseen: PC=R.*
16. JAZ <os>	10110	-,-	-	*Ehdollinen hyppy osoitteeseen: jos AC=0 niin PC=<os>, muuten PC=PC+1.*
17. JAZ R	10111	R	-	*Ehdollinen hyppy rekisterin määräämään osoitteeseen: jos AC=0 niin PC=R, muuten PC=PC+1.*
18. LD R,<os>	11000	R	-	*Hae sana muistista rekisteriin: R=M[<os>].*
19. LDi R,<os>	11001	R	-	*Hae sana epäsuorasti muistista rekisteriin: R=M[M[<os>]].*
20. STO R,<os>	11010	R	-	*Tallenna rekisteri muistiin: M[<os>]=R.*
21. STOi R,<os>	11011	R	-	*Tallenna rekisteri eräsuorasti muistiin: M[M[<os>]]=R.*

Taulukko 7.5: Esimerkkiarkkitehtuurin osoitetta hyödyntävät käskyt. Rekisterien koodaus kahdella bitillä noudattaa taulukkoa 7.3. Osoite <os> on 16 bittinen.

Komento	$b_0b_1b_2b_3$	b_4b_5	b_6b_7	selitys
22. JPS Ra, Rb <os>	1110	Ra	Rb	*Hyppy aliohjelmaan, jolloin nykyinen PC ja rekisterit tallennetaan parametrien välitystä varten:* *M[OSOITE+1]=Ra,* *M[OSOITE+2]=Rb,* *M[OSOITE]=PC+1,* *PC=OSOITE+3.*
23. RET Ra, Rb <os>	1110	Ra	Rb	*Paluu aliohjelmasta, jolloin rekisterit ja PC palautetaan:* *PC=M[OSOITE],* *Ra=M[OSOITE+1],* *Rb=M[OSOITE+2].*

Taulukko 7.6: Käskyt, joilla helpotetaan aliohjelmien kutsumista ja niihin paluuta.

Osa II

Tietojenkäsittelyn menetelmiä

Luku 8

Ohjelmointikielet ja ohjelmointi

8.1 Konekielet ja korkean tason ohjelmointikielet

Kuten edellisissä luvuissa on esitetty, nykyaikainen von Neumann (oik. EDVAC) -tyyppinen tietokone on yleiskäyttöinen laskulaite, joka voidaan saada suorittamaan mikä tahansa mekaanisesti toteutettavissa oleva laskutehtävä asettamalla koneen keskusmuisti oikeaan alkutilaan, so. tallentamalla muistiin tehtävän suorittamisohjeet sisältävä *konekieliohjelma*. Kullakin tietokonetyypillä on oma konekielensä, jotka tosin ovat yleensä pääpiirteiltään samanlaisia, koska myös tietokoneiden perusarkkitehtuuri on edelleen pääpiirteissään sama kuin ensimmäisissä EDVAC-tyyppisissä koneissa 50 vuotta sitten. Valmistusteknologiassa on tietenkin noista ajoista tapahtunut valtavaa edistystä ja koneiden sisältämien alkeiskomponenttien määrä on miljoonakertaistunut, mutta komponenttien ryhmittely isommiksi toiminnallisiksi kokonaisuuksiksi noudattaa edelleen niitä linjoja, jotka esitettiin John von Neumannin toimittamassa EDVAC-raportissa vuonna 1947.[1]

Koneenläheiseltä esitysmuodoltaan konekieliohjelmat ovat yksinkertaisesti bittijonoja, yksi kutakin käytettyä muistipaikkaa kohden. Inhimillisen luettavuuden ja ohjelmasuunnittelun helpottamiseksi käskyille on tapana antaa myös symboliset lyhenne- tai "muistikasnimet": esimerkiksi yhteenlaskuoperaatiota voidaan merki-

[1] Tietokoneiden rakenteen monimutkaistuminen yksityiskohtien tasolla heijastuu kuvaavasti niiden konekäskyjen määrässä: kun ensimmäisen toiminnallisen EDVAC-tyyppisen tietokoneen, Cambridgen yliopistossa vuonna 1949 valmistuneen EDSACin 18 konekäskyä sisälsivät jo useimmat nykyistenkin koneiden konekäskyjen perustyypit, oli 1970-luvun alussa melko yleisten HP2100-sarjan minikoneiden käskykannassa jo 80 operaatiota, ja nykymallisen Intel Pentium -prosessorin ohjaamiseen tarvitaan yli 120 konekäskyä.

Symbolinen konekieli:

```
      DEF   ALKU=100H
      DEF   LUKU1=200H
      DEF   LUKU2=202H
      DEF   SUMMA=204H

ALKU  LD    AC, LUKU1
      LD    IA, LUKU2
      ADD   AC, IA
      STO   AC, SUMMA
      HLT
```

Binäärikoodi (osoitteet ja käskyt):

```
100000000: 11000110          /* LD    AC       */
100000001: 00000010          /*       200H     */
100000010: 00000000          /*                */
100000011: 11000000          /* LD    IA       */
100000100: 00000010          /*       202H     */
100000101: 00000010          /*                */
100000110: 00101100          /* ADD   AC, IA   */
100000111: 11010110          /* STO   AC       */
100001000: 00000010          /*       204H     */
100001001: 00000100          /*                */
100001010: 00000100          /* HLT            */
```

Kuva 8.1: *Kahden luvun yhteenlaskun konekielitoteutus.*

tä ADD, muistisanan hakua rekisteriin LD (engl. "load register"), rekisterin sisällön tallennusta muistiin STO (engl. "store register") jne. Samoin voidaan koneen muistipaikoista ohjelmoinnin helpottamiseksi käyttää symbolisia nimiä ("I", "J", "PVM", "SOTU") numeeristen koneosoitteiden sijaan. Tietokoneella toteutettua apuohjelmaa, joka tuottaa tällaisesta *symbolisella konekielellä* (engl. "assembly language") kuvatusta ohjelmasta vastaavan binäärikoodin sanotaan *symbolisen konekielen kääntäjäksi* (engl. "assembler").

Esimerkiksi kuvassa 8.1 on esitetty luvussa 7.6 määritellyllä konekielellä laadittu käskyjono, joka laskee yhteen kaksi tietokoneen muistissa sijaitsevaa lukua ja tallentaa tuloksen kolmanteen muistipaikkaan. Käskyjono on kuvattu sekä symbolisella konekielellä että vastaavana binäärikoodina. Symbolisella konekielellä kirjoitetun

ohjelmatekstin alussa on joukko käännösohjelman toimintaa ohjaavia määrittelyjä, joilla kiinnitetään tuotettavan binäärikoodin ja sen käsittelemien tietojen sijoituspaikat. Ohjelmakoodi on tässä sijoitettu tietokoneen muistipaikkoihin 256–266 (100_{16}–$10A_{16}$), yhteenlaskettavat 2-tavuiset luvut muistipaikkoihin 512–513 ja 514–515 (200_{16}–201_{16} ja 202_{16}–203_{16}), sekä tallennettava 2-tavuinen summa paikkoihin 516–517 (204_{16}–205_{16}).[2] Symbolisen esityksen ja binäärikoodin vertailun helpottamiseksi on binäärikoodin rivien lopussa merkkiparien "/*" ja "*/" väliin kirjoitetuilla kommenteilla ilmaistu, mitä symbolisen esityksen operaatiota kukin binäärikoodin kohta vastaa.

Koska konekieliohjelmien muodostaminen on inhimilliselle ohjelmasuunnittelijalle tuskastuttavan pikkutarkkaa, työlästä ja virhealtista työtä, ohjelmointia helpottamaan on jo 1940-luvulta lähtien kehitetty koko joukko erilaisia korkeamman abstraktiotason käsitteisiin perustuvia ohjelmankuvausmenetelmiä, *korkean tason ohjelmointikieliä*. Tässä suunnassa, jota koneen laitteistoarkkitehtuuri ei rajoita, on kehitys ollut huomattavasti monimuotoisempaa kuin konekielen tasolla, ja uusien, entistä paremmin hallittavien korkean tason ohjelmointikielten kehittäminen onkin merkittävässä määrin helpottanut laajojen tietojenkäsittelysovellusten laatimista.[3]

Korkean tason kielet olivat aluksi vain ohjelmien paperisuunnittelua helpottamaan käytettyjä merkintätapoja, joista lopullinen konekoodi tuotettiin käsin, mutta jo 1950-luvun alussa toteutettiin ensimmäiset *automaattiset kääntäjät*: tietokoneohjelmat, jotka lukevat syöttölaitteelta korkean tason ohjelmakuvauksen ja sen pohjalta automaattisesti tuottavat vastaavan, suorituskelpoisen konekoodin. Merkittävä läpimurto oli John Backuksen johtaman IBM:n tutkimusryhmän vuonna 1957 julkistama FORTRAN (engl. "FORmula TRANslator") -kieli ja sen kääntäjä. FORTRAN oli käsitteelliseltä tasoltaan silloisessa katsannossa huomattavan abstrakti ja helppokäyttöinen kieli, ja sen automaattinen kääntäjä pystyi tuottamaan konekoodia, joka oli tehokkuudeltaan aivan kilpailukykyistä huolellisesti käsin laaditun koodin kanssa. Tämä ensimmäinen "moderni" korkean tason ohjelmointikieli saavutti valtavan suosion, ja sen kehittyneemmät versiot ovat edelleen keskeisiä työkaluja luonnontieteiden ja tekniikan tietojenkäsittelysovelluksia laadittaessa.

[2] Toisin kuin tässä esimerkissä, yleensä ei symbolista konekieltä käyttävän ohjelmoijan tarvitse huolehtia ohjelmalleen varattavien muistialueiden yksityiskohtaisesta muistiinsijoittelusta, vaan sen voi jättää käännösohjelman tehtäväksi.

[3] Suurten ohjelmistojen laatiminen on silti edelleen hyvin vaikeaa, ja jo nelisenkymmentä vuotta on puhuttu "ohjelmistotuotannon kriisistä", kun tietokoneiden ohjelmoinnin tehokkuus ei parane läheskään samalla nopeudella kuin laitteiden kapasitetti. Voidaan myös sanoa, että monista innovaatioista huolimatta nykyiset ohjelmointikielet silti monessa perustavassa suhteessa muistuttavat 1950-luvun lopulla esiteltyjä kantamuotojaan.

8.2 Korkean tason ohjelmointi

Tietokoneiden ohjelmointia tarkasteltaessa on syytä erottaa toisistaan *algoritmin* ja sen toteuttavan *ohjelman* käsitteet. *Algoritmilla* tarkoitetaan yleisesti täsmällistä kuvausta jonkin (laskenta)tehtävän suoritustavasta. Kuvauksen esitystavalle ei sinänsä ole asetettu rajoituksia, mutta sen täytyy olla niin yksityiskohtainen että menetelmä on periaatteessa täysin mekaanisesti seurattavissa. Kun algoritmi tietokonetoteutusta varten kuvataan jollakin täsmällisesti määritellyllä ohjelmointikielellä, saadaan tuloksena vastaava *(tietokone)ohjelma*.

Tietokoneohjelmoinnissa käytettävien algoritmien perusrakenteet ovat toimenpiteiden suorittaminen *peräkkäin*, vaihtoehtoisten toimenpiteiden kesken tehtävä *valinta* jonkin ehdon perusteella, sekä jonkin toimenpideryhmän suorituksen *toisto* niin kauan kuin jokin ehto pysyy voimassa.

Tarkastellaan esimerkkinä tunnettua *Eukleideen algoritmia* n. vuodelta 300 eKr kahden ei-negatiivisen kokonaisluvun suurimman yhteisen tekijän (engl. "greatest common divisor", gcd) määrittämiseksi:

Olkoot luvut m ja n.

1. Jos $n = 0$, niin tulos on s.y.t. $= m$.
2. Jos $n > 0$, niin:
   - 2.1 Olkoon r jakolaskun m/n jakojäännös.
   - 2.2 Aseta $m = n$, $n = r$, ja palaa kohtaan 1.

Algoritmiset perusrakenteet erottuvat tässä selvästi: kohdassa (1) *valitaan* ehdon "$n = 0$" perusteella saadaanko tulos suoraan, vai tarvitaanko kohdassa (2) kuvattu *toisto*. Toistoa vaativassa tapauksessa (2) suoritetaan *peräkkäin* toimenpiteitä (2.1) ja (2.2) niin kauan kuin ehto "$n > 0$" on tosi.

Algoritmissa tulee esiin myös useimmissa nykyisissä korkean tason ohjelmointikielissä keskeinen muuttuva-arvoisen parametrin eli *muuttujan* idea. Huomataan nimittäin, että muuttujanimien m, n ja r tarkoittamat kokonaisluvut, tai algoritmisuunnittelun käsittein muuttujien m, n ja r *arvot*, muuntuvat laskennan edetessä niin, että kun aluksi muuttujilla m ja n on alkuarvot $\{m = m_0,\ n = n_0\}$, niin laskennan päättyessä niiden loppuarvot toteuttavat ehdon $\{m = \mathrm{syt}(m_0, n_0),\ n = 0\}$.

Algoritmin formaalin rakenteen korostamiseksi se voidaan esittää kuvan 8.2 tapaisella *pseudokoodilla*. Pseudokoodiesityksessä ei ole mitään tarkasti määrättyjä ku-

```
input m, n
if n = 0 then output m else
    while n > 0 do
        r ← m mod n
        m ← n
        n ← r
    end do
    output m.
end if
```

Kuva 8.2: *Eukleideen algoritmin pseudokoodiesitys.*

```
input m, n
while n > 0 do
    r ← m mod n
    m ← n
    n ← r
end do
output m.
```

Kuva 8.3: *Eukleideen algoritmi virtaviivaistettuna.*

vaustapasääntöjä, mutta algoritmin rakentuminen toimenpiteiden peräkkäisyyden, valinnan ja toiston pohjalta tulee tulla selkeästi esiin. Kuvan 8.2 esityksessä on valittu englanninkieliset koodisanat **if - then - else** ja **while - do** kuvaamaan toimenpiteiden valintaa ja toistoa, mutta yhtä hyvin olisi voitu käyttää niiden suomenkielisiä vastineita **jos - niin - muuten** ja **niin kauan kuin - toista**. Muuttujan arvoa muuttavalle *sijoitusoperaatiolle* on käytetty vanhasta Algol-ohjelmointikielestä (1958) periytyvää nuolimerkintää "$m \leftarrow n$" (luetaan "m saa arvon n").

Pseudokoodiesityksessä olisi sallittua myös algoritmin joidenkin osien kuvaaminen alustavasti proosatekstinä myöhemmin tapahtuvaa kuvauksen tarkentamista varten (esimerkiksi "valitse syöteluvuista pienin", "järjestä syöteluvut nousevaan suuruusjärjestykseen" tms.), mutta tämänkertainen esimerkki on niin yksinkertainen, ettei tällaiseen kuvauksen *asteittaiseen tarkentamiseen* ole tarvetta.

Kuvan 8.2 algoritmiesityksestä huomataan, että mikäli syötearvot m ja n ovat oletuksen mukaisesti ei-negatiivisia kokonaislukuja, on itse asiassa algoritmin alussa tehtävä muuttujan n nollatestaus erillisenä tarpeeton, koska se sisältyy implisiittisesti myös seuraavaan toistorakenteeseen. Näin algoritmi saadaan kuvan 8.3 mukaiseen hieman virtaviivaisempaan muotoon.

```
(1)   main()
(2)   {  int m, n, r;
(3)      printf("Enter values m and n: ");
(4)      while (scanf("%u %u", &m, &n) != EOF)
(5)      {  while (n > 0)
(6)         {  r = m % n;
(7)            m = n;
(8)            n = r;
(9)         };
(10)        printf("gcd = %u\n", m);
(11)        printf("Enter values m and n: ");
(12)    }
(13)  }
```

Kuva 8.4: *Eukleideen algoritmin toteutus C-ohjelmana.*

Kuvan 8.3 algoritmin mukainen Eukleideen algoritmin toteutus nykyisin paljon käytetyllä C-ohjelmointikielellä on esitetty kuvassa 8.4. Rivinumerointi on lisätty kuvaan 8.4 vain tekstiviittauksia varten: se ei ole C-kielen piirre.

Ohjelma lienee annetun algoritmin pohjalta pääpiirteissään ymmärrettävä joitakin C-kielen detaljeja lukuunottamatta. Rivillä (1) kerrotaan, että kyseessä on sellaisenaan suoritettava ohjelmakokonaisuus, ns. pääohjelma (tämä on C-kielen vaatima ominaisuus). Rivillä (2) sanotaan, että ohjelma käsittelee kolmea kokonaislukuarvoista (engl. "integer") muuttujaa m, n ja r. Laitetasolla ajatellen muuttujat vastaavat tietokoneen muistipaikkoja, tai yleisemmin usean muistipaikan muodostamia muistialueita, joten rivillä (2) suoritettu muuttujien *esittely* ohjaa C-kääntäjää varaamaan tarvittavat muistialueet kolmen kokonaislukuarvoisen muuttujan tallentamista varten.

Rivillä (3) ohjelma kehotetaan kirjoittamaan tulostuslaitteelle (esim. näyttöruudulle) käyttäjää varten pyyntö antaa alkuarvot muuttujille m ja n. Rivillä (4) käyttäjän antamat arvot luetaan syöttölaitteelta (esim. näppäimistö) ja samalla C-kielen piirteitä käyttäen ohjataan ohjelmaa toistamaan rivejä (5...12) niin kauan kuin käyttäjä antaa kokeiltavaksi uusia syötelukuja.

Varsinaisen Eukleideen algoritmin toteutus annetuilla syöteluvuilla m ja n sisältyy riveihin (5 ... 9). Tämä ohjelmanosa noudattelee tarkoin kuvan 8.3 algoritmirakennetta. C-kielen erikoisuuksia ovat muuttujien m ja n arvojen jakojäännökselle käytetty merkintä "m % n" ja sijoitusoperaatiolle käytetty merkintä "$m = n$". Varsinkin

jälkimmäinen notaatio on hieman harhaanjohtava, koska kyse ei ole muuttujien m ja n arvoja koskevasta yhtälöstä muuten kuin siinä mielessä että sijoituksen suorittamisen jälkeen mainittu yhtälö on voimassa. (Myös jakojäännökselle olisi voinut toivoa C-kielen suunnittelijoiden valinneen jonkin osuvamman merkinnän: asiaa tuntematon voisi luulla kaavan "m % n" tarkoittavan "m prosenttia n:stä".)

Rivien (5 ... 9) sisältämän "while-silmukan", so. Eukleideen algoritmin suorituksen päätyttyä tilanteeseen $n = 0$ kehotetaan rivillä (10) ohjelmaa kirjoittamaan lopputuloksena saatu arvo m käyttäjän näkyville, ja rivillä (11) käyttäjää pyydetään antamaan muuttujille m ja n uudet lähtöarvot. Tästä kohdasta ohjelman suoritus palaa jälleen riviltä (4) alkavan while-silmukan alkuun. Kuten muistetaan, tätä silmukkaa toistetaan niin kauan, kunnes käyttäjä jollakin lopetusmerkillä ilmaisee ettei aio enää antaa uusia syötelukuja. Tällöin ohjelman suoritus päättyy.

8.3 Korkean tason ohjelmien suorittaminen

Korkean tason ohjelmointikielellä kirjoitetun ohjelman suorittamiseen koneella on kaksi tapaa. Ohjelma voidaan joko *kääntää* itsenäiseksi konekoodiksi erityisellä käännösohjelmalla (engl. "compiler") ja suorittaa näin syntynyt koodi, tai ohjelma voidaan *tulkita* korkean tason tulkkiohjelmalla (engl. "interpreter"), joka saa syötteenään korkean tason ohjelman ja jäljittelee sen tarkoitettua toimintaa rivi riviltä tuottamatta koskaan vastaavaa konekoodia.

Näistä suoritustavoista konekoodiksi kääntäminen on tavallisin menettely, koska itsenäisen konekoodin suorittaminen on kertalukua nopeampaa kuin ohjelman toiminnan jäljittely toisen ohjelman toimintojen kautta — tämä siis edellyttäen että käytetty käännösohjelma tuottaa laadukasta konekoodia. Hyvien kääntäjien tekeminen onkin melko vaativa tehtävä, ja tämän takia tulkitsevaa toteutusta käytetään usein silloin, kun toteutettava ohjelmointikieli on vasta prototyyppiasteella tai jollakin lailla niin tavallisuudesta poikkeava että kääntäjän tekeminen on aivan erityisen vaikeaa. Lisäksi tulkitseva toteutus teke mahdolliseksi ohjelman suorituksen keskeyttämisen ja ohjelman muuttamisen kesken suorituksen, minkä takia siitä on apua myös ohjelmien kehitys- ja testausympäristöissä. Nykyisin käytetyistä ohjelmointikielistä tyypillisesti tulkattuja ovat BASIC, Java, LISP ja Prolog, joskin näille kaikille on kehitetty myös kääntäjiä.

Kuvassa 8.5 on esitetty tapahtumien kulku, kun kuvan 8.4 esimerkkiohjelma, joka on käsittelyä varten tallennettu Jyväskylän yliopiston atk-keskuksen "tukki"-tietokoneen tiedostojärjestelmän tiedostoon nimeltä `gcd.c`, käännetään konekielelle

```
tukki>  gcc -o gcd.o gcd.c
tukki>  gcd.o

Enter values m and n: 12 16
gcd = 4
Enter values m and n: 18 15
gcd = 3
Enter values m and n: 123 456
gcd = 3
Enter values m and n: 1234567 1234568
gcd = 1
Enter values m and n: ^D

tukki>
```

Kuva 8.5: *Eukleides-ohjelman käännös ja suoritus*

samassa tietokoneessa käytettävissä olevalla C-kääntäjällä (nimeltään `gcc`) ja syntynyt konekoodi (tiedostossa `gcd.o`) suoritetaan.

8.4 Ohjelmointikielten kääntäminen

Käsitellessään syötteenä saamaansa korkean tason ohjelmaa käännösohjelma mm. asettaa ohjelman muuttujat vastaamaan tiettyjä koneen muistialueita ("tilanvaraus"), huolehtii että tietyllä ohjelman suoritushetkellä aktiiviset muuttujat ovat käytettävissä koneen rekistereissä ("rekisterien allokointi"), sekä toteuttaa ohjelman ohjausrakenteen (ehto- ja toistorakenteet jne.) konekooditasolla.

Tarkastellaan yksinkertaisena esimerkkinä kuvassa 8.6 esitettyä C-ohjelmanpätkää, joka laskee syötteenä annetusta ei-negatiivisesta kokonaisluvusta n lähtien tuloksen 2^n, kahdentamalla laskutoimituksen apuna käytettävän muuttujan m arvon ykkösestä alkaen n kertaa. Kahdennuskertojen lukumäärälaskurina toimii muuttuja i, joka saa aluksi arvon n, ja jota siitä lähtien pienennetään ykkösellä jokaisen m:n kahdennuksen yhteydessä, niin kauan kuin $i \neq 0$. (C-merkintä "`while (i != 0)`".)

Ohjelmaa vastaava konekoodi luvussa 7.6 määritellyllä konekielellä on esitetty kuvassa 8.7. Käännöksessä on korkean tason muuttujaa n (2 tavua) asetettu vastaamaan koneen muistipaikat 200_{16}–201_{16}, muuttujaa m muistipaikat 202_{16}–203_{16} ja

```
scanf("%u", &n);
m = 1; i = n;
while (i != 0)
{  m = 2 * m;
   i = i - 1;
};
printf("%u\n", m);
```

Kuva 8.6: *Eksponenttiohjelma C-kielellä.*

muuttujaa i muistipaikat 204_{16}–205_{16}. Muistipaikkoihin 206_{16}–207_{16} ja 208_{16}–209_{16} kääntäjä on tallentanut valmiiksi ohjelman tarvitsemat lukuvakiot 1 ja 2. Ohjelmakoodin kääntäjä on sijoittanut alkamaan muistiosoitteesta 100_{16}.

Koska aiemmassa arkkitehtuurin esittelyssämme ei ole otettu kantaa siihen, miten tietokoneen syöttö- ja tulostustoiminnot hoidetaan, oletetaan kuvan 8.7 koodissa yksinkertaisesti, että ennen ohjelman suorituksen alkamista on syöteluvun n arvo tallennettu muistipaikkoihin 200_{16}–201_{16}, ja ohjelman suorituksen päättyessä tulosarvo $m = 2^n$ löytyy muistipaikoista 202_{16}–203_{16}.

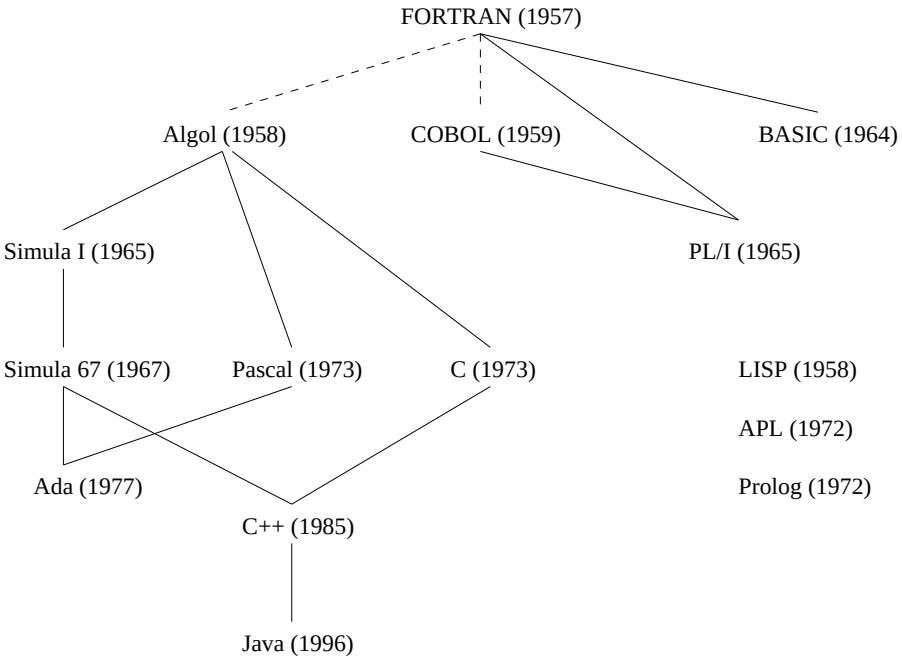
8.5 Korkean tason ohjelmointikieliä

Kirjallisuudessa on 1950-luvulta lähtien ehdotettu useita satoja korkean tason ohjelmointikieliä, mutta vain muutama niistä on vakiintunut käyttöön. Ankara karsinta tarjolle tulleiden vaihtoehtojen kesken johtuu siitä, että uuden ohjelmointikielen vakiintumista ei säätele yksinomaan kielen laatu, vaan myös suurimittaiset institutionaaliset tekijät: mitä organisaatioita kielellä on tukenaan, ketkä kehittävät sille kääntäjiä ja sovelluskantaa, kuka sitä ylläpitää, miten suuri ja tärkeä on muilla kielillä laadittu sovelluskanta, jota myös täytyy ylläpitää jne.

Kuvan 8.8 kaaviossa on hahmoteltu muutaman vuosien mittaan merkittävään asemaan nousseen ohjelmointikielen käsitteellistä "perimysjärjestystä". Tarkastellaan joitakin näistä kielistä esimerkinomaisesti hieman lähemmin.

```
      DEF   START=100H
      DEF   VARN=200H
      DEF   VARM=202H
      DEF   VARI=204H
      DEF   CON1=206H
      DEF   CON2=208H

START LD    AC, CON1     /* m = 1;               */
      STO   AC, VARM
      LD    AC, VARN     /* i = n;               */
      STO   AC, VARI
LOOP  JAZ       END      /* while (i != 0)       */
      LD    AC, CON2     /* {  m = 2 * m;        */
      LD    IA, VARM
      MUL   AC, IA
      STO   AC, VARM
      LD    AC, VARI      /*    i = i - 1;}       */
      LD    IA, CON1
      SUB   AC, IA
      STO   AC, VARI
      JMP       LOOP
END   HLT

VARN  CON 0              /* muutt. n */
VARM  CON 0              /* muutt. m */
VARI  CON 0              /* muutt. i */
CON1  CON 1              /* vakio  1 */
CON2  CON 2              /* vakio  2 */
```

Kuva 8.7: *Eksponenttiohjelman konekoodi.*



Kuva 8.8: *Merkittäviä korkean tason ohjelmointikieliä*

```
      program GCD
      integer m, n, r
10    print *, 'Please give values for m and n'
      read *, m, n
20    if (n .eq. 0) go to 30
      r = mod(m,n)
      m = n
      n = r
      go to 20
30    print *, 'gcd = ', m
      go to 10
      end
```

Kuva 8.9: *Eukleideen algoritmi FORTRAN77-kielellä.*

```
10 print "Please give values for m and n"
20 input m, n
30 if n = 0 then 80
40 let r = m mod n
50 let m = n
60 let n = r
70 goto 30
80 print "gcd = ", m
90 goto 10
100 end
```

Kuva 8.10: *Eukleideen algoritmi BASIC-kielellä.*

8.5.1 FORTRAN

FORTRANin ensimmäisen version julkistuksesta (1957) on jo yli neljäkymmentä vuotta, mutta se on edelleen luonnontieteiden ja tieteellisen laskennan sovellusten valtakieli. Kieltä on maltillisesti päivitetty vuosina 1977 ja 1990; kuvassa 8.9 on esitetty Eukleideen algoritmi vuoden 1977 standardin mukaisella FORTRAN-kielellä.

8.5.2 BASIC

Helppokäyttöinen BASIC-kieli (engl. "Beginner's All-purpose Symbolic Instruction Code") suunniteltiin alunperin (John Kemeny & Thomas Kurtz 1964) ohjelmoinnin oppimista helpottamaan, ja tavallaan se on onnistunut tässä tarkoituksessa liiankin hyvin: kielen rakenteet eivät tue järjestelmällistä ohjelmasuunnittelua, mutta koska sen oppiminen on niin helppoa ja sillä ohjelmointi pienessä mittakaavassa niin vaivatonta, kieli on levinnyt laajaan käyttöön varsinkin harrasteohjelmoijien keskuudessa. Jonkin verran kehittyneempi versio kielestä on pohjana paljon käytetyssä Visual Basic -ohjelmointiympäristössä. Käyttötarkoituksensa takia BASIC-toteutukset ovat usein tulkkipohjaisia, vaikka kielen kääntäminen konekoodiksi ei ole lainkaan vaikeaa. Kuvassa 8.10 on esitetty Eukleideen algoritmi perus-BASICillä.

8.5.3 LISP

LISP (engl. "LISt Processing language) on John McCarthyn vuonna 1958 määrittelemä omalaatuinen, rekursiivisiin (itseviittaaviin) funktiomäärittelyihin perustuva

Ohjelma:

```
(defun gcd (m n)
(cond
  ((equal n 0) m)
  (t (gcd n (mod m n)))))
```

Suoritus:

```
(gcd 12 16)
4
(gcd 123 456)
3
```

Kuva 8.11: *Eukleideen algoritmi LISP-kielellä*

ohjelmointikieli. Puhtaassa muodossaan kielessä ei ole edes muuttujia ja sijoitusoperaatioita, vaan kaikki ohjelman tilannetieto sisältyy sen tekemien sisäkkäisten funktiokutsujen argumentteihin.

Outoudestaan huolimatta kieli on suhteellisen helppo oppia ja oikein käytettynä se on tehokas työkalu monimutkaistenkin ongelmien ratkaisemiseen, varsinkin silloin kun ratkaisulle ei aseteta korkeita tehokkuusvaatimuksia. Kieli on erityisen suosittu ns. tekoälysovellusten ohjelmoinnissa (ks. luku 13). LISP-toteutukset ovat tyypillisesti tulkkipohjaisia, joskin kääntäjiäkin on kehitetty.

Eukleideen algoritmin LISP-toteutuksen ymmärtämiseksi kirjoitetaan algoritmi ensin hieman toiseen muotoon. Merkitään ei-negatiivisten lukujen m ja n suurinta yhteistä tekijää $\gcd(m, n)$; tällöin voidaan algoritmin mukaan laskea:

1. Jos $n = 0$, niin $\gcd(m, n) = m$.
2. Jos $n \neq 0$, niin $\gcd(m, n) = \gcd(n, m \bmod n)$.

Tämä laskukaava voidaan tulkita myös kahden kokonaislukuargumentin funktion $\gcd(\cdot, \cdot)$ määrittteleväksi *palautus-* t. *rekursiokaavaksi*, jonka mukaan kaikilla $m, n \geq 0$ on:

$$\gcd(m, n) = \begin{cases} m, & \text{jos } n = 0, \\ \gcd(n, m \bmod n), & \text{muuten.} \end{cases}$$

Kuvassa 8.11 on esitetty suoraan e.m. palautuskaavaan pohjautuva Eukleideen algoritmin LISP-toteutus.

Ohjelma on melko helppolukuinen, kun tietää että tavallista matemaattista funktiomerkintää $f(x, y)$ vastaa LISPissä merkintä "`(f x y)`". Kuvan 8.11 ohjelma määrittelee ("`defun`") funktion gcd, jonka arvo argumenteilla m ja n määritetään käymällä läpi tietty ehtolista ("`cond`"). Ensimmäinen ehto lausuu, että jos $n = 0$ ("`(equal n 0)`"), niin funktion arvo on m, ja toinen ehto, että muissa tapauksissa ("`t`") funktion arvo on sama kuin $\gcd(n, m \bmod n)$ ("`(gcd n (mod m n))`").

8.5.4 Prolog

Samaan tapaan kuin LISP perustuu matemaattisiin funktiomäärittelyihin, perustuu Prolog-kieli (engl. "Programming in logic", Richard Kowalski & Alain Colmerauer 1972) loogisten predikaattien määrittelyihin ja toteutuvuuskyselyihin. Esimerkiksi Eukleideen algoritmia vastaten voitaisiin määritellä predikaatti

$$\gcd(M, N, D) \quad \equiv \quad \text{"lukujen } M \text{ ja } N \text{ s.y.t. on } D\text{"}$$

totuusehdoilla tai "aksioomilla"

$$\begin{array}{ll} \forall M: & \gcd(M, 0, M), \\ \forall M, N > 0, D: & \gcd(N, M \bmod N, D) \Rightarrow \gcd(M, N, D). \end{array}$$ [4]

Kun nyt luvut M ja N on annettu, voidaan edellisten aksioomien avulla päättelemällä etsiä arvo D, jolla ehto $\gcd(M, N, D)$ on voimassa.

Prolog-tulkkiohjelman keskeinen komponentti on "päättelykone", joka etsii annetut loogiset ehdot toteuttavia muuttujanarvoja. Kuvassa 8.12 on ensin esitetty, miten e.m. gcd-predikaatti määriteltäisiin Prolog-kielessä: kuvan Prolog-ohjelman ensimmäinen rivi määrittelee säännön "ehto $\gcd(M, 0, M)$ on tosi kaikilla arvoilla M", ja toinen rivi määrittelee säännön "ehto $\gcd(M, N, D)$ on tosi, jos $N > 0$ ja arvolla $R = M \bmod N$ ehto $\gcd(N, R, D)$ on tosi". Tämän jälkeen on esitetty, miten Prolog-tulkin päättelykone löytää D-arvot, jotka toteuttavat ehdot $\gcd(12, 16, D)$ ja $\gcd(123, 456, D)$.

8.6 Ohjelmien oikeellisuus

Ohjelmointi korkeankin tason ohjelmointikielellä on tarkkuutta vaativaa ja virhealtista työtä, ja laaditut ohjelmat ovat ensi yrittämällä useammin virheellisiä kuin vir-

[4] Toinen rivi luetaan: jos on totta, että $\gcd(N, M \bmod N, D)$, niin on totta myös että $\gcd(M, N, D)$.

Ohjelma:

```
gcd(M,0,M).
gcd(M,N,D) :- N > 0, R is M mod N, gcd(N,R,D).
```

Suoritus:

```
?- gcd(12,16,D).
D = 4;
no
?- gcd(123,456,D).
D = 3;
no
```

Kuva 8.12: *Eukleideen algoritmi Prolog-kielellä.*

heettömiä. Tavallisesti virheitä etsitään ohjelmista *testaamalla* niitä erilaisilla syöteaineistoilla. Jos ohjelma tällöin toimii väärin, siinä tietenkin on virhe, mutta kun yleensä on mahdollista testata vain pieni osa kaikista potentiaalisista syöteaineistoista, ei testien onnistuminen vielä takaa ohjelman virheettömyyttä.

Jotta voitaisiin olla aivan varmoja siitä, että ohjelma todella toimii oikein kaikissa mahdollisissa tilanteissa, ohjelma pitäisi *todistaa* oikeaksi. Tämä on periaatteessa mahdollista (tietyin rajoituksin), mutta käytännössä hyvin hankalaa: tietokoneohjelmat ovat isoja ja mutkikkaita kokonaisuuksia, ja niiden formaalit oikeellisuustodistukset vielä kertaluokkaa mutkikkaampia. Oikeellisuustodistusten merkitys on kuitenkin lisääntymässä "hankalissa" tai "kriittisissä" ympäristöissä toimivien ohjelmien, tai ainakin niiden keskeisimpien komponenttien kohdalla. Tällaisia ovat esimerkiksi jotkin tietoliikenne- tai reaaliaikajärjestelmien osat, jotka eivät välttämättä ole ohjelmateksteinä kovin pitkiä, mutta joiden toimintaympäristössä esimerkiksi syötteiden määrä tai järjestys voi vaihdella, ja lukuisia häiriötilanteita voi esiintyä.

Pienenä esimerkkinä ohjelmien oikeellisuustarkasteluista todistetaan kuvan 8.6 eksponenttiohjelman oikeellisuus. Kuvassa 8.13 on tarkasteltava algoritmi uudelleen kirjoitettuna C-kielen erityispiirteistä puhdistettuna pseudokoodiesityksenä.

Algoritmin oikeellisuuden todistamista varten tulee todistaa kaksi väitettä:

(i) Algoritmin suoritus päättyy kaikilla syötearvoilla n, jotka ovat ei-negatiivisia kokonaislukuja.

(ii) Jos syötearvo n on ei-negatiivinen kokonaisluku, niin algoritmin palauttama

input n
$m \leftarrow 1$, $i \leftarrow n$
while $i \neq 0$ **do**
    $m \leftarrow 2 \cdot m$
    $i \leftarrow i - 1$
end do
output m.

Kuva 8.13: *Eksponenttialgoritmi.*

tulosarvo on $m = 2^n$.

Väite (i) on selvästi voimassa, koska algoritmin **while**-silmukassa esiintyvän "silmukkamuuttujan" i arvo on kokonaisluku, jonka alkuarvo on $n \geq 0$ ja jota pienennetään jokaisella silmukan suorituskerralla. Tämä on mahdollista vain äärellisen monta kertaa ennen kuin muuttujan i arvoksi tulee 0.

Väitteen (ii) todistus perustuu siihen, että vaikka algoritmin käsittelemien muuttujien m ja i arvot jokaisella **while**-silmukan suorituskerralla muuttuvat, niiden suhteesta säilyy voimassa seuraava ns. *silmukkainvariantti*:

> Muuttujien m ja i arvot silmukan kunkin suorituskerran alussa toteuttavat ehdon $I : \{m \cdot 2^i = 2^n\}$.

Oletetaan nimittäin, että muuttujien m ja i arvot toteuttavat ehdon I silmukan jonkin suorituskerran alussa. Oletetaan lisäksi, että $i \neq 0$, niin että silmukan runko tulee suoritettavaksi. Silmukan rungossa muuttujille sijoitetaan uudet arvot $m' = 2m$ ja $i' = i - 1$. Mutta tällöin myös nämä uudet arvot toteuttavat ehdon I, onhan nimittäin

$$m' \cdot 2^{i'} \quad = \quad (2m) \cdot 2^{i-1} \quad = \quad m \cdot 2^i \quad = \quad 2^n.$$

Siten jokainen silmukan suorituskerta säilyttää invariantin I.

Tarkastellaan sitten kuvan 8.13 ohjelman toimintaa annetulla syötteellä $n \geq 0$ kokonaisuutena. Huomataan, että ennen **while**-silmukan ensimmäistä suoritusta muuttujien m ja i arvot toteuttavat ehdon $P_0 : \{m = 1, i = n\}$, ja tavoitteena on osoittaa, että silmukan viimeisen suorituksen jälkeen muuttujan m arvo toteuttaa ehdon $P_1 : \{m = 2^n\}$. Nyt voidaan todeta, että alkuehto P_0 takaa, että invariantti I on voimassa ennen silmukan suoritusta:

$$m = 1,\ i = n \quad \Rightarrow \quad m \cdot 2^i = 2^n.$$

Edellisen tarkastelun nojalla invariantti I säilyy tämän jälkeen voimassa jokaisella silmukan suorituskerralla. Toisaalta I yhdessä silmukan lopetusehdon $\{i = 0\}$ kanssa takaa halutun lopputuloksen:

$$m \cdot 2^i = 2^n,\ i = 0 \quad \Rightarrow \quad m = 2^n.$$

Siten algoritmi toimii halutulla tavalla kaikilla ei-negatiivisilla kokonaislukusyötteillä n.

Luku 9

Käyttöjärjestelmät ja laitteistot

9.1 Käyttöjärjestelmä

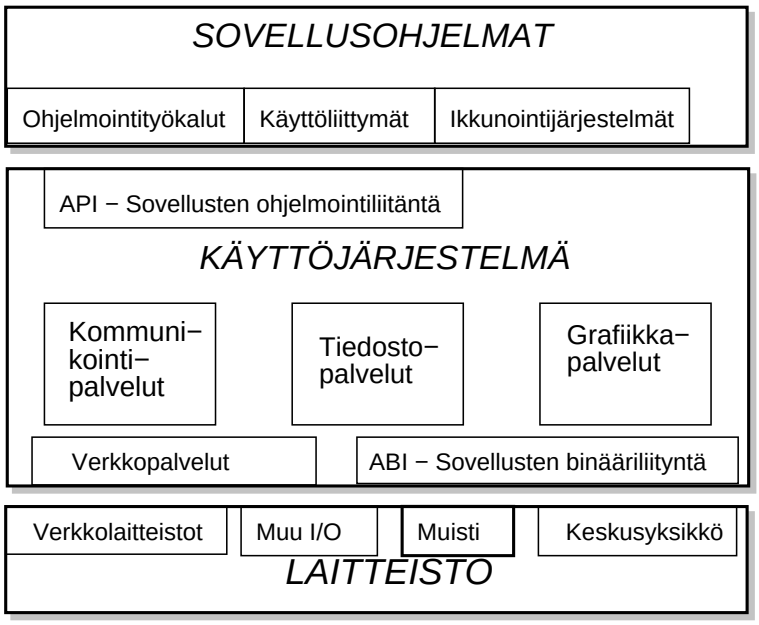
Käyttöjärjestelmä on tietokoneen peruspalveluja tarjoava varusohjelmisto, joka muodostaa koneen käyttöä helpottavan "puskurikerroksen" sovellusohjelmoijan tai loppukäyttäjän ja varsinaisen laitteiston väliin. Käyttöjärjestelmä mm.:

- yksinkertaistaa ja yhtenäistää laitteiston osien (muistijärjestelmä, oheislaitteet jne.) käyttöä sovellusohjelmissa;
- jakaa laitteistoresursseja (prosessoriaika, työtila jne.) samanaikaisesti suoritettaville töille;
- tarjoaa yleispalvelut sovellusohjelmien ja datan hallintaan (tiedostojärjestelmä, ohjelmien käynnistäminen jne.).

Kehittyneet käyttöjärjestelmät ovat olleet edellytys tietotekniikan sovellusten monimuotoistumiselle, sillä ilman käyttöjärjestelmää:

- tietokoneella voisi suorittaa vain yksi käyttäjä yhtä ohjelmaa kerrallaan;
- jokaisen ohjelman pitäisi huolehtia itse kaikista tietokoneen oheislaitteiden ohjauksen yksityiskohdista;
- tietokoneresurssien yhteiskäyttö (levytilan jakaminen, tietoliikenne jne.) olisi jokseenkin mahdotonta.

Käyttöjärjestelmän ja sen päälle rakennettujen ohjelmistotyökalujen rooli vaihtelee, mutta suunnilleen samat, kuvassa 9.1 esitetyt komponentit löytyvät muodossa tai toisessa useimmista käyttöjärjestelmällä varustetuista tietokoneista.

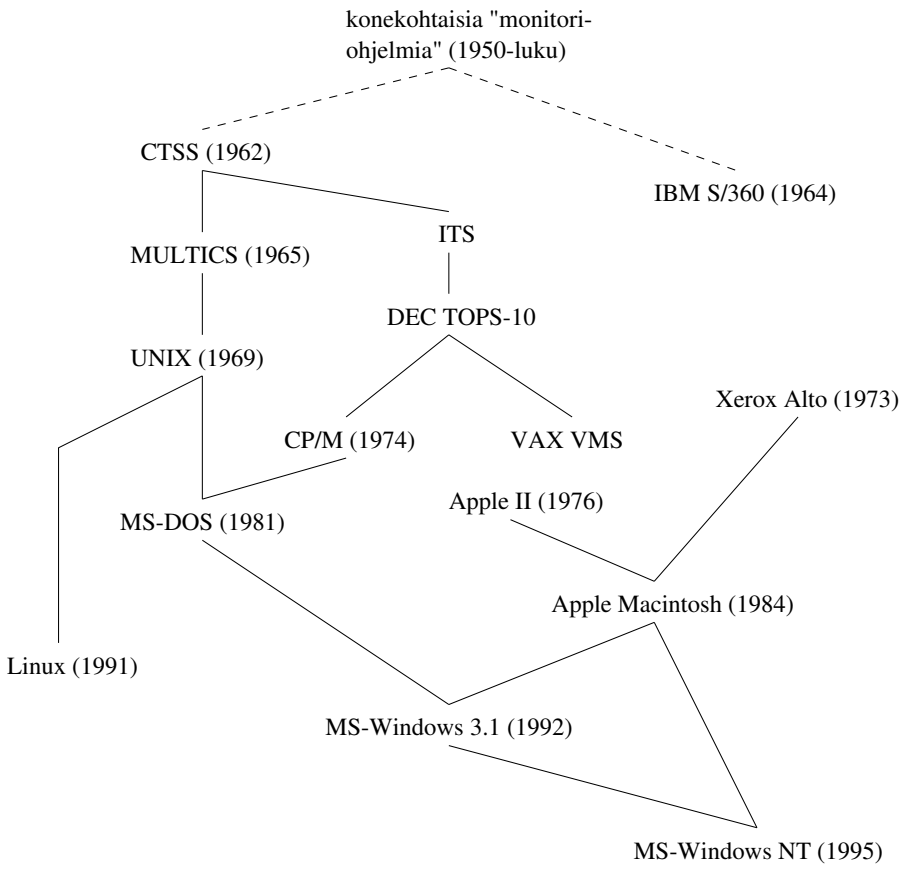


Kuva 9.1: *Tyypillisen käyttöjärjestelmän rakenne.*

9.2 Käyttöjärjestelmien historiaa

Nykyiset tietokoneiden käyttöjärjestelmät ovat tulos monivaiheisesta teknisestä kehityksestä, jossa monet keskeiset ideat järjestelmän rakenteesta ja toteutuksesta ovat peräisin 1960-luvulta ja sen käyttöliittymästä 1970-luvulta (kuva 9.2).

Tietokoneiden käyttöä helpottamaan laadittiin jo 1950-luvulla konekohtaisia "monitoriohjelmia", joiden tehtävänä oli aluksi helpottaa kunkin yksittäin suoritettavan ohjelman lataamista tietokoneen muistiin ja auttaa tietokoneen oheislaitteiden (reikäkortinlukija, kirjoitin jne.) käytössä; sittemmin myös järjestellä eri käyttäjien peräkkäin suoritettavia töitä niin, ettei työn vaihtaminen normaalitilanteissa enää vaatinut inhimillisen operaattorin puuttumista laitteiston asetuksiin. Tällaista töiden



Kuva 9.2: *Merkittäviä käyttöjärjestelmiä.*

suorittamista tietokoneella peräkkäin yksi kerrallaan sanotaan laitteiston *eräkäytöksi*, ja se oli tietokoneiden vallitseva käyttömuoto 1970-luvun loppupuolelle asti.[1]

1960-luvulle tultaessa tietokoneet olivat jo tehostuneet sen verran keskimääräisen sovellustehtävän vaativuuteen ja syöte- ja tulostuslaitteiden nopeuteen nähden, että tuli mahdolliseksi tarjota tilaisuus koneen käyttöön pääteyhteyden välityksellä *samanaikaisesti* usealle käyttäjälle. Tällaisessa *osituskäyttöjärjestelmässä* koneen keskusyksikkö palvelee kutakin käyttäjää vuorotellen lyhyen (esim. 1 ms) *aikaviipaleen* ajan kerrallaan. Käyttäjille syntyy tällöin parhaassa tapauksessa se vaikutelma, että he ovat kukin koneen ainoa asiakas, koska vuorovaikutteisessa käytössä keskusyksikön laskenta-ajan tarve on vähäinen verrattuna siihen aikaan, minkä vaatii tietojen syöttäminen ja tulostaminen sekä käyttäjän ajatustyö päätteen ääressä.

Osituskäyttöjärjestelmässä tietokoneen käytön kokonaistehokkuus paranee huomattavasti, kun käyttäjät saavat vastaukset laskentatehtäviinsä heti, ja voivat tarvittaessa muuttaa tehtävänasetuksia saman tien, tarvitsematta odottaa vuoroaan eräajojonossa. Toisaalta osituskäyttöä tukevan käyttöjärjestelmän laatiminen on vaikea tehtävä, johon liittyen tehtiin 1960-luvulla paljon uraauurtavaa tutkimustyötä. Eturivissä tässä oli Massachusettsin teknillisen korkeakoulun MIT:n tutkimusryhmä, jonka toteuttama CTSS (engl. "Compatible Time Sharing System", 1962) oli ensimmäinen yleiskäyttöinen osituskäyttöjärjestelmä. Nopeasti koottua CTSS-järjestelmää kunnianhimoisempi hanke oli sen seuraaja MULTICS (engl. "MULTiplexed Information and Computer System", 1965), jonka monet innovaatiot, kuten hierarkkinen tiedostojärjestelmä ja ns. virtuaalimuisti (ks. kappale 9.7), ovat keskeisiä nykyisissäkin käyttöjärjestelmissä.

CTSS-järjestelmän perillisinä syntyi muitakin osituskäyttöjärjestelmiä: Digital Equipment -yhtiön pientietokoneille toteutettu ITS (engl. "Incompatible Time Sharing system") ja sen seuraaja TOPS-10, sekä Bell-yhtiön tutkimuslaitoksessa toteutettu Unix (1969). Unix on monien kilpailuvaiheiden jälkeen noussut valta-asemaan lähes kaikkien muiden kuin PC-mikrotietokoneiden käyttöjärjestelmänä, ja on tällä hetkellä nopeasti yleistymässä PC-ympäristöissäkin suomalaisen Linus Torvaldsin aloitteesta syntyneen Linux-toteutuksen muodossa.

Tietotekniikan alkuvuosikymmenien jättiläinen IBM ei ollut kiinnostunut osituskäyttöjärjestelmistä, vaan yhtiössä toteutettiin sen omille laitteistoille massiivinen

[1]Eräajojärjestelmää sovelletaan edelleen esimerkiksi vaativia laskentatehtäviä supertietokoneympäristössä suoritettaessa: vaikkapa sääennustuksen laskemiseen on tarkoituksenmukaisempaa varata hetkellisesti koko käytettävissä oleva tietokonekapasiteetti ja vapauttaa se sitten seuraavalle työlle kuin "kilpailuttaa" ennusteen laskemista muiden samanaikaisesti tekeillä olevien töiden kanssa.

System/360-eräkäyttöjärjestelmä (1964), jolla (myöhempine versioineen) oli kaupallis-hallinnollisissa tietojenkäsittelysovelluksissa tärkeä asema 1970-luvun lopulle saakka.

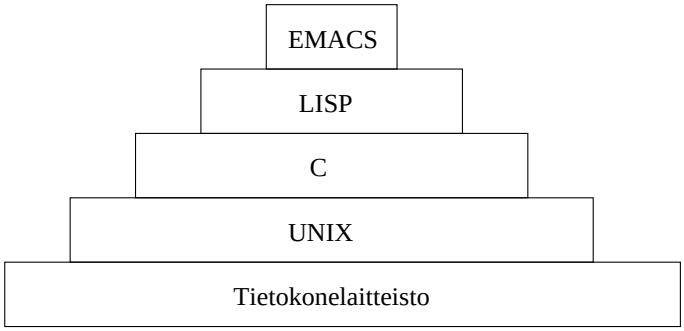
Mikrotietokoneiden kehittyminen ja yleistyminen 1980-luvun alkupuolella asetti käyttöjärjestelmille uudenlaisia haasteita. Perinteisen osituskäytön merkitys väheni, kun käyttäjät saivat omia tietokoneita, mutta toisaalta osituskäyttöjärjestelmiä varten kehitetyt tekniikat tulivat nyt palvelemaan kutakin yksittäistä käyttäjää, jolla voi olla samanaikaisesti useita töitä käynnissä omassa koneessaan (puhutaan ns. *moniajojärjestelmistä*). Yhden yhteisen keskuskoneen päätekäytön on vähitellen kokonaan korvaamassa hajautetun mikrotietokoneiden verkon keskinäinen tietoliikenne, jonka tarjoamien mahdollisuuksien hyödyntäminen asettaa taas käyttöjärjestelmien kehitykselle uusia haasteita.

Tärkeä uusi vaatimus, jonka yhden käyttäjän mikrotietokoneiden yleistyminen käyttöjärjestelmille asetti, oli koneiden käyttömukavuuden parantaminen. Ensimmäiset mikrotietokoneiden käyttöjärjestelmät — ITS/TOPS-järjestelmän kaltainen varhaisten mikromerkkien CP/M, sen ja Unixin piirteitä yhdistelevä IBM-tyyppisten mikrojen MS-DOS, sekä Apple-yhtiön ensimmäisten konemallien käyttöjärjestelmä — tarjosivat käyttäjälle vain tekstipohjaisen liittymän alkeelliseen yksiajoympäristöön. Mikrotietokoneiden syntyessä oli kylläkin jo nykyisen kaltainen edistynyt ikkunointiin ja osoitusmekanismiin perustuva käyttöliittymä toteutettu Xerox-yhtiön kokeellisessa Alto-tietokoneessa (1973), mutta yhtiö ei ollut kiinnostunut tämän tuotteen markkinoinnista. Xeroxilta omaksutun käyttöliittymäidean kaupallisen arvon onnistui realisoimaan vasta Apple-yhtiö edistyksellisessä Macintosh-tietokoneessaan (1984), ja Windows 95/NT -järjestelmien myötä tämä moniajoa hyödyntävä, ikkunointiin ja osoitusmekanismiin perustuva käyttöliittymätyyppi on vakiintunut myös IBM-tyyppisiin mikrotietokoneisiin.

9.3 Virtuaalikone

Nykyaikaisia tietokoneohjelmia ei laadita suoraan peruslaitteistolle, vaan käyttöjärjestelmäympäristön luomalle korkeamman tason "virtuaalikoneelle". Kukin käyttöjärjestelmäympäristö luo oman tyyppisensä virtuaalikoneen, ja sama virtuaalikone (esim. Unix-ympäristö) voi olla toteutettuna erilaisilla laitteistoilla.

Yleisemminkin ohjelmistotyön voi ajatella olevan uusien virtuaalikoneiden rakentamista olemassaolevien pohjalle. Esimerkiksi kun Unix-järjestelmän käyttäjä kirjoittaa tekstiä yleisesti käytetyllä EMACS-editorilla (teksturilla), hän käyttää tietoko-



Kuva 9.3: *Eräs virtuaalikonehierarkia.*

nettaan viisiportaisen virtuaalikonehierarkian kautta (kuva 9.3). EMACS-editori on nimittäin ohjelmoitu LISP-kielellä, jonka tulkki puolestaan on toteutettu alunperin C:llä.[2] C-ohjelmat toimivat Unix-käyttöjärjestelmän luomassa virtuaaliympäristössä, joka sovelluksen näkökulmasta näyttää samalta alla olevan tietokonelaitteiston teknisistä yksityiskohdista riippumatta. Jos nyt EMACS-editori, tai yleisemmin koko Unix-ympäristössä toteutettujen ohjelmistojen joukko, halutaan siirtää uudelle laitealustalle, riittää periaatteessa laatia uusi toteutus "vain" alimman tason Unix-virtuaalikoneelle, minkä jälkeen hierarkian ylemmät tasot voidaan siirtää muuttamattomina. Vastaavasti EMACS-editori on periaatteessa Unix-käyttöjärjestelmästä riippumaton, ja on pienin muutoksin siirrettävissä mihin tahansa muuhunkin systeemiympäristöön, jossa on toteutettuna LISP-tulkki.

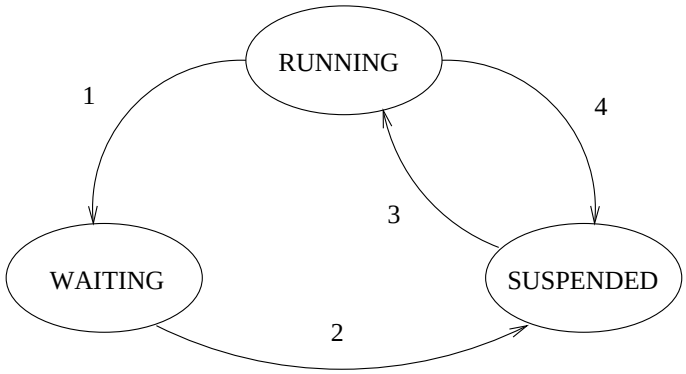
9.4 Prosessit

Prosessin käsite on keskeinen nykyisissä moniajokäyttöjärjestelmissä, joissa tietokone ainakin näennäisesti suorittaa monta tehtävää samanaikaisesti. Abstraktisti prosessilla tarkoitetaan yhden tehtävän, so. tietokoneohjelman suorituksen muodostamaa kokonaisuutta. Konkreettisesti prosessin esityksen tietokoneessa muodostavat suoritettavan ohjelman koodi ja data (syötetiedot ja ohjelman sisäiset tietorakenteet), sekä joukko prosessia koskevaa kirjanpitotietoa, kuten ohjelman suorituskohdan osoitin, ohjelman sijainti tietokoneen muistissa jne.

[2]Itse asiassa myös EMACS itse sisältää osanaan täysimittaisen LISP-tulkin, jonka avulla käyttäjä voi halutessaan laajentaa editorin toimintoja rajattomasti.

Tietokoneessa on yleensä samaan aikaan suoritettavana useita prosesseja, jopa niin että sama ohjelmakoodi voi olla samanaikaisesti käynnissä useana kopiona eri syötetiedoilla. Jos koneessa on vain yksi prosessori, voi prosesseista vain yksi kerrallaan olla todella *aktiivinen*, so. prosessorin käsittelyssä (engl. "running"); muut ovat tällä aikaa joko *valmiustilassa*, mutta pysäytettyinä (engl. "suspended", valmiina suoritukseen, kun saavat prosessorin haltuunsa) tai *odotustilassa* (engl. "waiting", odottamassa esimerkiksi syöttö- tai tulostusoperaation valmistumista).

Käyttöjärjestelmän suoritus on myös yksi prosessi muiden joukossa. Se luo ja tuhoaa sovellusprosesseja, käynnistää ja pysäyttää niiden suorituksen, sekä jakaa niille systeemin resursseja (suoritusaikaa, muistitilaa, oheislaitteiden käyttövuoroja). Myös sovellusprosessit voivat luoda avukseen uusia "lapsiprosesseja".



Kuva 9.4: *Moniajojärjestelmän prosessin tilanvaihtokaavio.*

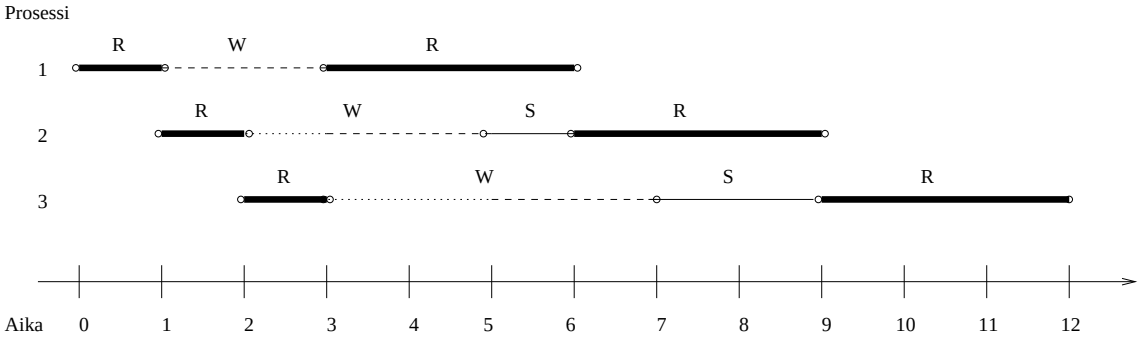
Kuva 9.4 esittää moniajojärjestelmässä suoritettavana olevan prosessin eri tilojen välisiä siirtymämahdollisuuksia, jotka ovat:

1. Aktiivinen prosessi siirtyy odotustilaan jonkin ulkoisen syyn takia (esim. oheislaitteelta tai toiselta prosessilta tarvittavan datan odotus); käyttöjärjestelmän *vuorottajaohjelma* (engl. "scheduler") valitsee sen tilalle uuden aktiivisen prosessin valmiustilassa olevien prosessien joukosta.

2. Odotuksen syy poistuu (esim. tiedonsiirto oheislaitteelta valmistuu) ja prosessi siirtyy valmiustilaan.

3. Valmiustilassa oleva prosessi saa vuorottajalta suoritusvuoron.

4. Aktiivinen prosessi siirtyy valmiustilaan, kun sille annettu aikaviipale loppuu. Vuorottaja valitsee uuden aktiivisen prosessin valmiustilassa olevien joukosta.



Kuva 9.5: *Kolmen prosessin toimintojen limittyminen moniajojärjestelmässä*

Tarkastellaan prosessien tilanvaihtoa vielä esimerkin valossa. Olkoon yksiprosessorisen tietokoneen suoritettavana kolme prosessia, jotka kukin ensin suoríttavat jotain laskutehtävää 1 ms ajan, sitten lukevat yhteiseltä syöttölaitteelta 2 ms ajan, ja lopuksi laskevat vielä 3 ms ajan. Yksinkertaisuuden vuoksi oletetaan, että prosesseille myönnettävän aikaviipaleen pituus on vähintään 3 ms, niin ettei niiden tarvitse keskeyttää laskentaansa aikaviipaleen loppumisen takia, ja ettei prosessien tilanvaihto vaadi lainkaan ylimääräistä aikaa. Jos järjestelmässä ei ole moniajopiirrettä, prosessien suorittamiseen kuluu aikaa yhteensä 3×6 ms $= 18$ ms. Moniajoympäristössä prosessien tiedonsiirtoa ja laskentaa voidaan limittää niin, että niiden suoritus vaatii vain 12 ms (kuva 9.5).

9.5 Keskeytykset

Moniajojärjestelmässä tarvitaan jokin mekanismi, jolla prosessori saadaan pois aktiivisena olevalta prosessilta takaisin käyttöjärjestelmän haltuun. Tämä mekanismi ovat ns. *keskeytykset* (engl. “interrupts”). Useimmat prosessien tilasiirtymät liittyvät nimenomaan keskeytyksiin, joilla jokin oheislaite, ajastinkello tai prosessi itse ilmoittaa vaativansa toimenpiteitä. Keskeytysten käsittely tietokoneessa etenee seuraavaan tapaan:

1. Laite tai ohjelma tekee keskeytyspyynnön muuttamalla jonkin keskusyksikössä sijaitsevan lipukerekisteriin (engl. “flag”) arvoa.

2. Prosessori tutkii jokaisen suorittamansa konekäskyn jälkeen lipukkeet ja huomaa keskeytyksen.

3. Uudet keskeytykset estetään käsittelyn ajaksi.

4. Prosessori tallentaa aktiivisena olleen prosessin kirjanpitotiedot (prosessin *ympäristön*, engl. "context") erityiselle *keskeytyskäsittelijän muistialueelle*, jotta keskeytyneen prosessin suoritusta voidaan jatkaa myöhemmin.

5. Prosessori suorittaa keskeytyskäsittelyrutiinin, joka tunnistaa keskeytyssignaalin lähteen ja käynnistää vastaavan palvelun.

6. Keskeytyskäsittelyn päätteeksi prosessori palauttaa keskeytyneen prosessin ympäristön ja jatkaa sen suoritusta — tai mahdollisesti siirtää keskeytyneen prosessin valmiustilaan ja vaihtaa aktiivista prosessia. (Näin käy esimerkiksi jos keskeytyksen syy oli ajastinkellon antama merkki aktiivisen prosessin aikaviipaleen loppumisesta.)

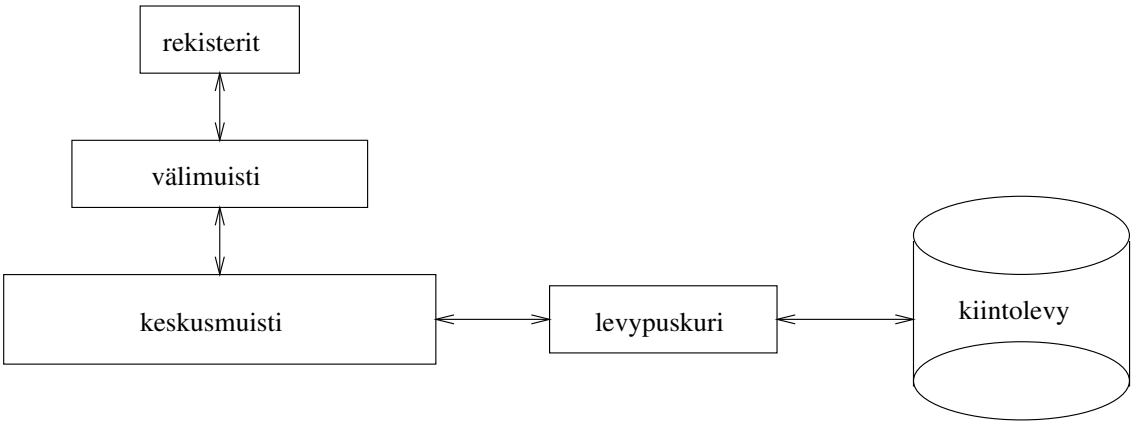
7. Keskeytykset sallitaan taas.

9.6 Muistinhallinta

Moniajojärjestelmässä ei ohjelmaa kirjoitettaessa tai käännettäessä vielä kiinnitetä sen lopullista sijaintia tietokoneen muistissa, vaan vasta käyttöjärjestelmä varaa ohjelmalle sen tarvitseman muistialueen kun ohjelman suorittava prosessi luodaan. Käyttöjärjestelmä voi myös siirtää prosesseja niiden suorituksen aikana keskusmuistissa paikasta toiseen, ja mahdollisesti jopa keskus- ja tukimuistin välillä.

Muistinhallinnan yksityiskohtien piilottaminen sovellusohjelmilta käyttöjärjestelmän palvelukutsujen taakse tekee mahdolliseksi senkin, että ohjelman käyttämä "looginen" osoiteavaruus voi olla suurempi kuin koneessa olevan fyysisen keskusmuistin määrä. Tällöin osa ohjelman käsittelemistä tiedoista sijaitsee keskusmuistissa, osa jollakin hitaammalla mutta tilavammalla tukimuistilaitteella. (Kustannussyistä tietokoneen muistilaitteet muodostavat kuvan 9.6 esittämän kaltaisen hierarkian nopeista mutta kalliista, ja siksi pienistä, muistiyksiköistä hitaisiin mutta halpoihin ja suuriin.[3]) Sovellusohjelmoijan ei kuitenkaan tarvitse kiinnittää mitään huomiota

[3]Esimerkiksi nykyisissä (2002) mikrotietokoneissa on keskusmuistin koko tyypillisesti 128–512 megatavua, ja tukimuisteina käytettyjen kiintolevyjen koot vaihtelevat välillä 4–32 gigatavua. Levymuistit ovat siis tyypillisesti noin sata kertaa tilavampia kuin keskusmuistit.



Kuva 9.6: *Tietokoneen muistilaitteiden hierarkia.*

tietojen sijoitteluun eri muistilaitteille, vaan käyttöjärjestelmä luo ympäristön, jossa sovellusohjelmien kannalta koko muistijärjestelmä on yksi suuri yhtenäinen alue.

9.7 Virtuaalimuisti

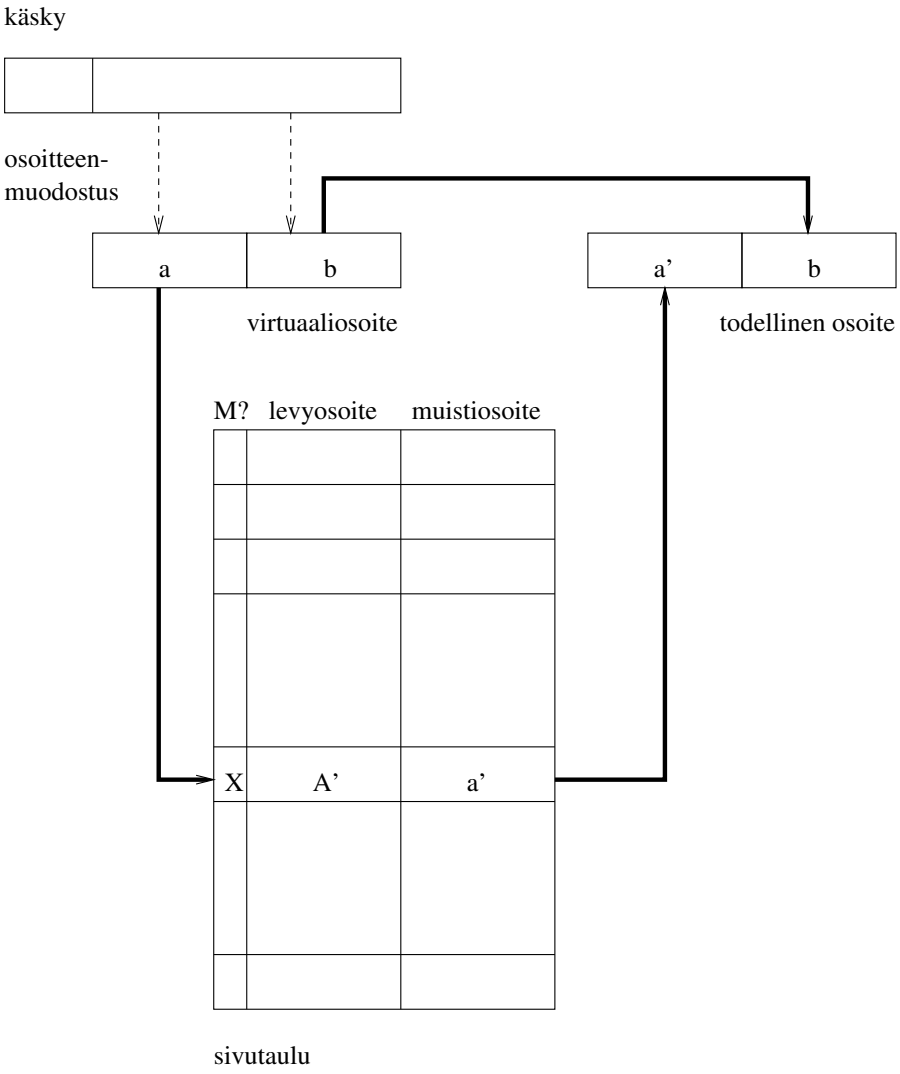
Muistinhallinnan ongelmat ratkaistaan moniajojärjestelmissä ns. *virtuaalimuistitekniikalla*: erottamalla ohjelman käyttämien muistiosoitteiden muodostama "osoiteavaruus" ja tietokoneen fyysisten muistipaikkojen muodostama "muistiavaruus" täysin toisistaan. Loogiset osoitteet ja fyysiset muistipaikat yhdistetään toisiinsa prosessikohtaisilla *sivutauluilla* (engl. "page tables", kuva 9.7).

Sivutaulujen idea on seuraava:

1. Osoiteavaruus jaetaan 2^r tavun kokoisiin *sivuihin* ($r = 9 \ldots 13$), jotka voidaan sijoittaa kukin eri paikkaan tietokoneen fyysisessä muistissa (tyhjiä sivuja vastaavia fyysisiä muistisivuja ei tarvitse edes luoda).

2. Ohjelman viitatessa osoitteeseen $m = a \cdot 2^r + b$ muistinhallintajärjestelmä hakee ensin sivutaulusta loogista sivua a vastaavan fyysisen sivun alkuosoitteen a'; b on tällöin halutun muistipaikan suhteellinen muistiosoite ao. sivulla, so. loogista osoitetta m vastaava fyysinen osoite on $m' = a' \cdot 2^r + b$.

3. Sivutaulussa on kenttä, joka ilmaisee onko haettu sivu valmiiksi keskusmuistis-



Kuva 9.7: *Osoitteenmuodostus virtuaalimuistijärjestelmässä.*

Viite	*Sivu 1*	*Sivu 2*	*Sivu 3*
0051	**0000–0999**	?	?
1076	0000–0999	**1000–1999**	?
0052	*0000–0999*	1000–1999	?
3974	0000–0999	1000–1999	**3000–3999**
2342	0000–0999	**2000–2999**	3000–3999
0053	*0000–0999*	2000–2999	3000–3999
1511	0000–0999	2000–2999	**1000–1999**
3975	0000–0999	**3000–3999**	1000–1999
0054	*0000–0999*	3000–3999	1000–1999
1782	0000–0999	3000–3999	*1000–1999*
3976	0000–0999	*3000–3999*	1000–1999

Kuva 9.8: *LRU-periaatteen mukainen sivunvaihtojono.*

sa, vai joudutaanko se noutamaan tukimuistista. Jälkimmäisessä tapauksessa viittaus sivutauluun aiheuttaa *sivunvaihtokeskeytyksen* (engl. "page fault interrupt"), jonka käsittelyn yhteydessä tehdään tarvittavat muistirakenteiden päivitykset.

4. Uuden sivun tuonti keskusmuistiin merkitsee yleensä jonkin vanhan sivun palauttamista keskusmuistista tukimuistiin. Palautettava sivu voidaan valita eri tavoin.

Yksi virtuaalimuistin sivunvaihtoperiaate on palauttaa tukimuistiin se sivu, jonka edellisestä käytöstä on kulunut mahdollisimman kauan. Tämän ns. LRU-periaatteen (engl. "Least Recently Used") havainnollistamiseksi tarkastellaan seuraavaa esimerkkiä. Oletetaan, että yhden keskusmuistisivun koko on 1000 tavua, ja järjestelmä on antanut eräälle suoritettavana olevalle prosessille käyttöön kolmen keskusmuistisivun kiintiön. Kun prosessin osoiteviittausten jono on

$$0051, 1076, 0052, 3974, 2342, 0053, 1511, 3975, 0054, 1782, 3976,$$

vaihtelee sen keskusmuistisivujen sisältö kuvan 9.8 esittämällä tavalla. Jonon suorittaminen aiheuttaa kuusi sivunvaihtokeskeytystä.

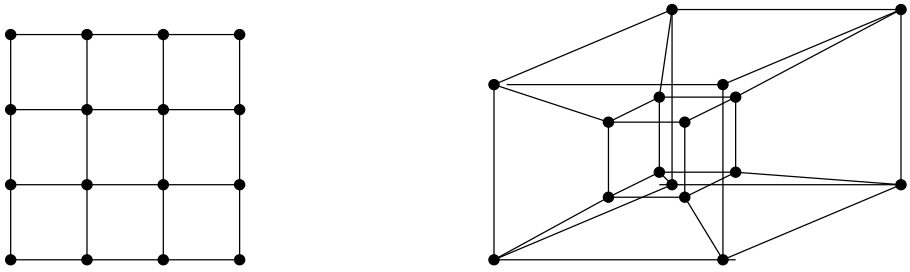
9.8 Rinnakkaiskoneet

Loppuhuomautuksena mainittakoon, että käyttöjärjestelmiä joudutaan kehittämään samalla, kun uusia tietokonearkkitehtuureja tulee markkinoille. Eräs mielenkiintoi-

nen sovelluskenttä ovat nk. rinnakkaiskoneet.

Tietokoneiden laskentatehoa on jo 1960-luvulta lähtien pyritty lisäämään kytkemällä useita prosessoreita toimimaan synkronisesti ("tasatahtisesti") rinnakkain. Tällaisissa rinnakkaiskoneissa prosessorit on kytketty toisiinsa ja mahdolliseen yhteiseen keskusmuistiin nopeilla koneensisäisillä tiedonsiirtoväylillä.

Nykyisin on jo melko tavallista, että tietokone sisältää esimerkiksi kaksi prosessoria, mutta ns. *supertietokoneissa* niitä voi olla huomattavasti enemmänkin: esimerkiksi Jyväskylän yliopiston tietotekniikan laitoksen SUN UE4000 "minisuperkone" sisältää 8 prosessoria ja valtakunnallisen Tieteellisen laskennan palveluyksikön CSC:n (engl. "Centre for Scientific Computing") IBM pSeries 690-kone 512 prosessoria. Onpa valmistettu jopa yli 60000 prosessoriakin sisältäviä koneita (Thinking Machines Corporation'in Connection Machine 2 — tosin tämän koneen prosessorit ovat erittäin yksinkertaista tyyppiä, joten määrää ei voi suoraan verrata "tavanomaisiin" superkoneisiin).



(a) Hila (n=4) (b) Hyperkuutio (r=4)

Kuva 9.9: *Rinnakkaiskoneiden kytkentätopologioita.*

Rinnakkaiskoneen prosessorit voidaan kytkeä yhteen eri tavoin: kaksi suosittua kytkentäjärjestystä ("topologiaa") ovat *hila*, jossa $N = n^2$ prosessoria sijoitetaan yhteysväylien muodostaman $n \times n$-ristikon solmukohtiin, ja *hyperkuutio*, jossa $N = 2^r$ prosessoria sijoitetaan loogisesti r-ulotteisen hyperkuution kulmiin (kuva 9.9). Hilatopologian etuna on yksinkertainen toteutus, mutta haittana se, että kommunikointietäisyys prosessorista toiseen on pisimmillään luokkaa $\sqrt{N}$ askelta. Hyperkuutiotopologiassa kommunikointietäisyys on enintään $\log_2 N$ askelta, mutta kommunikointiverkon rakenne riippuu prosessorien määrästä ja sen toteuttaminen on hankalaa.

Rinnakkaiskoneille ei ole vakiintunut yksiprosessorikoneiden von Neumann -arkkitehtuurin tapaista standardimallia, vaan niitä on lukuisia eri tyyppejä. Yleisellä tasolla rinnakkaisarkkitehtuurit on tapana jakaa prosessorien itsenäisyyden mukaan *tehtävärinnakkaisiin* MIMD-koneisiin (engl. "Multiple Instruction streams, Multiple Data streams") ja *datarinnakkaisiin* SIMD-koneisiin (engl. "Single Instruction stream, Multiple Data streams"). MIMD-tyyppiset koneet ovat yleiskäyttöisiä rinnakkaiskoneita, joiden prosessorit suorittavat kukin ohjelmaansa täysin itsenäisesti, mutta vuorovaikutuksessa toisten prosessorien kanssa (esim. edellä mainitut Sun UE4000 ja IBM pSeries 690). SIMD-tyyppiset koneet ovat erikoiskoneita, joissa kaikki prosessorit suorittavat synkronisesti samaa ohjelmakoodia, mutta eri data-alkioille. SIMD-tyyppiset koneet ovat MIMD-koneita edullisempia, ja ne ovat käyttökelpoisia, jos käsiteltävä data on säännönmukaista, vaikkapa jonkin suureen mittausarvoja säännöllisen mittaushilan pisteistä. Edellä mainitussa TMC CM/2 -tietokoneessa on sekä MIMD- että SIMD-prosessointia tukevia piirteitä.

Rinnakkaiskoneiden etuna on laskennan nopeutuminen, mutta niiden yleistymistä on hidastanut joukko käytännöllisiä tekijöitä. Ensinnäkin on rinnakkaisuutta tehokkaasti hyväksikäyttävien ohjelmien tekeminen osoittautunut yllättävän vaikeaksi, ja toisekseen rinnakkaiskoneet ovat tyypillisesti kalliita erikoislaitteita, jotka vanhenevat nopeasti vakiomallisten koneiden nopean kehityksen rinnalla.[4]

Mielenkiintoinen, lähiverkkojen nopeutuessa toteuttamiskelpoiseksi tullut idea on tavallisen työasemaverkon käyttö rinnakkaislaskentaan (ns. *klusterilaskenta*). Etuna tässä mallissa on, että rinnakkaislaskentaympäristö voidaan rakentaa edullisista standardilaitteista, jotka mahdollisesti hankittaisiin joka tapauksessa muuta käyttöä varten. Ongelmina toisaalta ovat, että prosessoreiden kommunikointi tavanomaisen tietoliikenneverkon kautta on huomattavasti hitaampaa kuin yhden koneen sisäisen väylän avulla, ja verkon kautta kommunikoivien koneiden asynkronisen ("epätahtisen") toiminnan hallinta on vaikeaa. Lähivuosina nähdään, johtaako tämä kehitys suurteholaskennan ja laskentaresurssien yhteiskäytön vallankumoukseen, vai estävätkö laskentamalliin liittyvät vaikeudet sen leviämisen laajempaan käyttöön.

[4] Tietokoneiden laskentateho on 1970-luvun alusta lähtien kaksinkertaistunut säännöllisesti noin puolentoista vuoden välein — ilmiö, joka tunnetaan ns. *Mooren lakina* — joten nyt (1999) kaupasta ostettava mikrotietokone on yhtä tehokas kuin kymmenen prosessorin rinnakkaiskone vuodelta 1994, ja voittaa tehossa sataprosessorisen supertietokoneen vuosimallia 1989.

Luku 10

Tietoliikenne

Tietoliikenteen kehittyminen on ollut yksi 1990-luvun merkittävimmistä teknologisista edistysaskeleista. Kehitystä on tapahtunut samanaikaisesti kolmella rintamalla:

- Puhelinteknologiassa verkkojen digitalisoitumisen ja langattoman tiedonsiirron myötä.
- Kaupallisessa tiedonsiirrossa, kuten TV- ja satelliittiyhteyksissä.
- Tietokoneiden tiedonsiirrossa, joka on kehittynyt keskustietokoneen pääteyhteyksistä nykyaikaisiin informaatioverkkoihin.

Perinteisesti laajinta tietoliikenteen käyttö on ollut puheviestimien puolella, mutta viimeaikoina datapohjainen tiedonsiirto on vallannut alaa tietotekniikan yleisen kehityksen myötä. Pidetään jopa mahdollisena, että puheluista tulee verkkojen myötä ilmaisia ja vain datayhteydestä maksetaan.

10.1 Tietoverkkojen kehitys

Aina 1980-luvun puoliväliin tietokoneiden välinen tiedonsiirto suoritettiin kahden koneen välisenä (pisteeestä-pisteeseen) yhteytenä. Pidemmillä etäisyyksillä käytettiin puhelinverkkoa, johon data koodattiin modeemien avulla. Varsinainen tietoverkon kehittyminen, eli ajatus koneiden välisestä "ulkoisesta" väylästä nykytilaan lähti liikkeelle USA:n puolustusministeriön tarpeesta turvata tiedonsiirto kiriisitilanteissa. Seuraavassa kehityshistoria tiivistettynä.

Vuodet 1957-1979. Neuvostoliitto laukaisee Sputnik-satelliitin, minkä seurauksena USA kiinnostuu tiedonsiirron turvaamisesta. Kehitystyön tuloksena syntyy ARPA-verkko (Internetin edeltäjä), joka annetaan myöhemmin tutkijoiden käyttöön. Tästä alkaa kehitys kohti Internet verkkoa. ARPA-verkon pohjalle kehitetään palveluja, mm. sähköposti ja TCP/IP-kommunikointiprotokolla.

Vuodet 1980-90. Uusia verkkoratkaisuja syntyy, kuten BITNET-verkkoprotokolla, ja Ranskassa tulee markkinoille puhelinverkossa toimiva videotext-järjestelmä: Minitel, joka on tavallaan WWW:n edeltäjä. Nimipalvelu kehitetään, mahdollistaen automaattisen reitityksen. Ensimmäinen rekisteröity DNS-nimi on "Symbolics.com". USA:ssa avataan kansallinen NSFNET (56Kbps nopeudella) verkko lähinnä yliopistojen käyttöön. Network News Transfer Protocol (NNTP) kehitetään TCP/IP:n päälle ja Usenet "news" postiryhmät syntyvät. Vuonna 1988 opiskelijan kehittämä Internet mato-ohjelma sotkee 10 % verkon koneista ja Jarkko Oikarinen Oulun yliopistosta kehittää Internet Relay Chat (IRC) ohjelman verkkokeskusteluihin. Myös Suomi rekisteröi maakohtaisen tunnuksen Finland (FI) yhtäaikaa Ranskan, Islannin, Norjan ja Ruotsin kanssa.

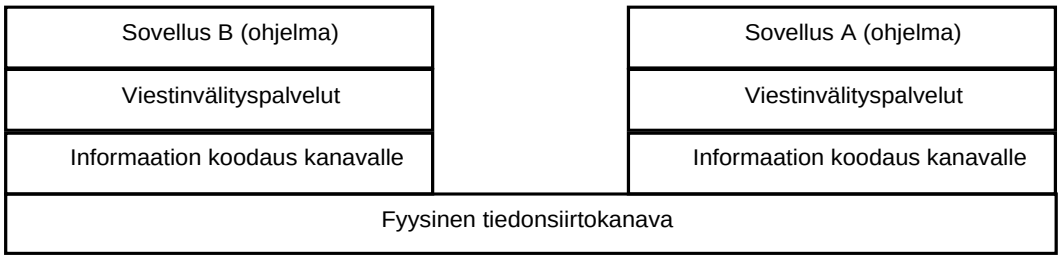
Vuodet 1990-1995. Alkuperäinen ARPANET lakkautetaan. Gopher-järjestelmä kehitetään verkon globaaliksi tiedostopalveluksi, ja kilpaileva World Wide Web (WWW) julkistetaan CERN:ssä. Vuonna 1993 Mosaic-selainohjelma "räjäyttää pankin" ja WWW:n käyttömäärä kasvaa yli 3000 kertaiseksi vuodessa. Tämän seurauksena Internet huomataan ensimmäistä kertaa julkisuudessa ja yritysmaailmassa. Verkottuminen alkaa voimistua, yritykset liittyvät verkkoon.

Vuodet 1995-2000. Kaupallinen kilpailu synnyttää nk. selainsodan (Netscape vs. MicroSoft). Uusia teknologioita kehitetään: virtuaaliympäristöt (VRML), ryhmätyökalut, verkkotietokone (metatietokone) ja internetkännykkä. Samalla standardointi ja kaupallinen kehitys kiihtyy.

10.2 Tietoverkon periaate

Tietoverkko, kuten muutkin modernit tietotekniikan osa-alueet, on useammasta eritasoisesta palvelusta koostuva järjestelmä. Karkeasti katsoen kaksi ohjelmaa voi tietoverkon kautta kommunikoida kuvan 10.1 kaltaisella tavalla, missä tieto fyysisesti kulkee johtoa (tai ilmaa) pitkin, ja tietoverkko on fyysisten yhteyksien lisäksi yhteisesti sovittu tapa informaation siirtoon.

Tietoliikenteessä standardoitujen ratkaisujen merkitys on suurempi kuin muilla tietotekniikan osa-alueilla. Tunnetuin tietoliikenteen yleinen standardi, joka pyrkii kat-



Kuva 10.1: *Tietoverkon perusperiaate.*

tamaan kaikenlaisen tiedonsiirron perusperiaatteet, tunnetaan nk. OSI-mallin nimellä. Tämä malli on kansainvälisten standardointijärjestöjen suunnittelema sopimus tiedonsiirroksi avoimiin ympäristöihin, mutta se kehitettiin liian myöhään, sillä monet "de facto" standardit kuten TCP/IP, ISDN, ISBN eivät suoraan noudata sitä. Tästä huolimatta OSI-malli tarjoaa hyvän pohjan tietoverkkoratkaisujen ymmärtämiselle. Malli koostuu seitsemästä kerroksesta.

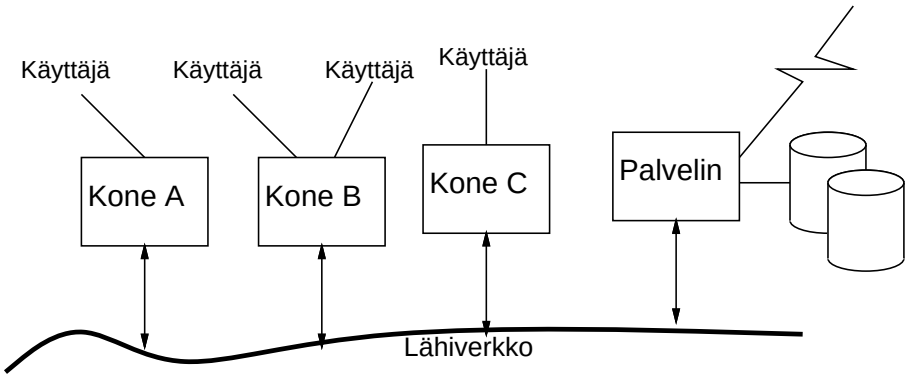
Kerros	Merkitys	Vastaa suunnilleen
1	Fyysinen taso	Kaapeliliitäntä
2	Data yhteys	Open/close
3	Verkko taso	Alempi-Internet
4	Välitystaso	TCP -protokolla
5	Yhteystaso	FTP/Telnet
6	Esitystaso	HTTP-palvelut
7	Sovellustaso	Email-ohjelmat

Kukin kerros tarjoaa joukon palveluja, joita ylemmällä tasolla toteutettavat ohjelmat voivat hyödyntää. Siten esimerkiksi sovellusohjelman tekijän ei tarvitse tietää miten alemman tason tietoliikenne toteutetaan, sillä alempien OSI-kerroksien ohjelmat tekevät tämän hänen puolestaan.

10.3 Lähiverkot ja laajaverkot

Tietoverkkojen myötä perinteiset yhden keskuskoneen ja siihen kytkettyjen päätelaitteiden muodostamat tietojenkäsittelyjärjestelmät ovat jo korvautuneet hajautetuilla järjestelmillä, joissa joukko suhteellisen itsenäisiä *työasematietokoneita* on

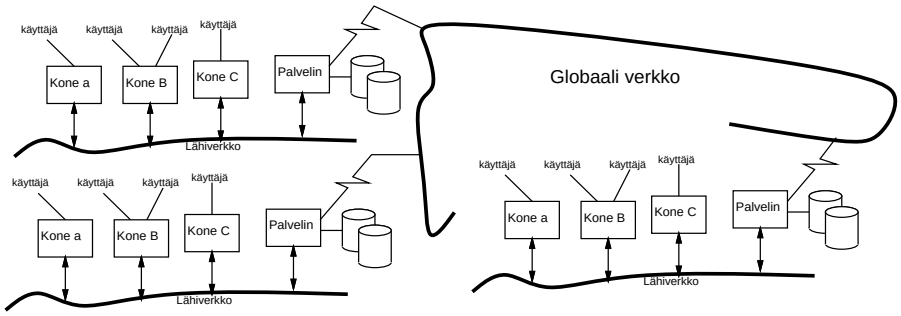
tietoliikenneyhteyksin kytketty yhteen *tietokoneverkoksi*. Tällöin samaan organisaatioon kuuluvat, tiiviisti kytketyt koneet muodostavat *lähiverkon* (LAN = local area network, esimerkiksi Jyväskylän yliopiston mikrotietokoneverkot). Tämänkaltaisen verkon rakennetta on havainnollistettu kuvassa 10.2.



Kuva 10.2: *Lähiverkon periaate: palvelinkone ja itsenäisiä käyttäjiä.*

Lähiverkot ovat tyypillisesti toimistoverkkoja, jossa yksi koneista on palvelin (server), jonka tarjoamia resursseja (kirjoittimet, levyt, Internet-yhteys, käyttäjätunnusten hallinta) muut koneet voivat hyödyntää. Niissä on yleensä käyttäjien omien työasemakoneiden lisäksi yhteiskäyttöisiä *palvelinkoneita*, jotka tarjoavat verkon kaikille koneille yhteisiä palveluita: huolehtivat esimerkiksi keskitetystä tiedostojen tallennuksesta (tiedostopalvelin) tai verkon tietoliikenteestä (tietoliikennepalvelin), sisältävät yhteisesti käytettyjä ohjelmia (ohjelmistopalvelin), ohjaavat verkon yhteisiä syöte- ja tulostuslaitteita (oheislaitepalvelin) jne. Yksi fyysinen palvelinkone voi tietenkin huolehtia useista palveluista.

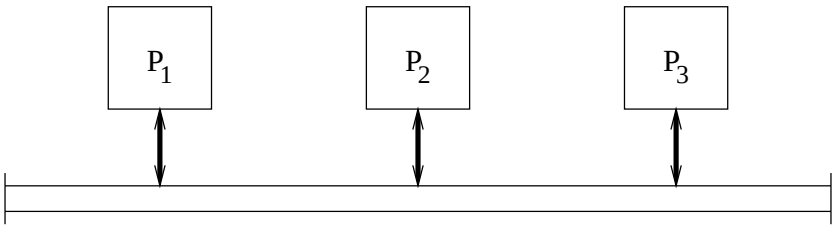
Joukko yhteen kuuluvia lähiverkkoja muodostaa kuvan 10.3 kaltaisen *laajaverkon* (WAN = wide area network), mistä käy esimerkkinä Suomen maakohtainen korkeakouluverkko FUNET. Lähes kaikista laajaverkoista on nykyään liityntä yhteiseen kansainväliseen Internet-verkkoon. Verkot voidaan lisäksi tehdä konsernikohtaisesti, jolloin koneiden väliset fyysiset etäisyydet ovat mahdollisesti erittäin suuria, mutta näkyvät käyttäjilleen lähiverkon kaltaisina.



Kuva 10.3: *Laajaverkon periaate.*

10.4 Kytkentäratkaisut

Koska tietoverkoilla on samanaikaisesti useita lähettäjiä ja vastaanottajia, on jollakin tavalla huolehdittava, että viestit eivät mene sekaisin. Siis yhteisen resurssin käytöstä on sovittava määräämällä verkon kytkentöjensyntymismekanismi. Tavallisimmat kytkentäratkaisut ovat *kilpavaraus-* toiselta nimeltään *Ethernet-väylä* (kuva 10.4) ja *valtuusrengas* (engl. "Token Ring", kuva 10.5).

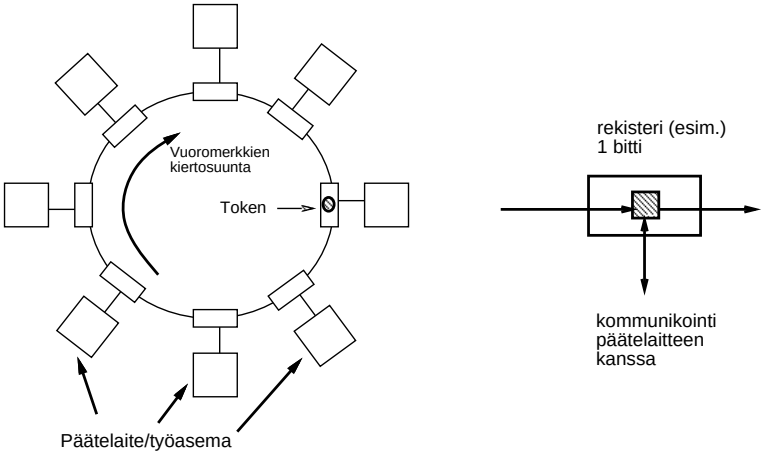


Kuva 10.4: *Kilpavarausväylä.*

Kilpavarausverkossa on kaikki verkon koneet kytketty samaan tiedonsiirtoväylään, jolle ne saavat kukin milloin tahansa lähettää sanomia.[1] Sanomat ovat fyysisesti kaikkien väylällä sijaitsevien koneiden luettavissa, mutta ainoastaan sen koneen tietoliikenneohjelmisto, jolle sanoma on tarkoitettu, ottaa sen jatkokäsittelyyn. Jos

[1] Tarkkaan ottaen väylä on kuormituksen vähentämiseksi yleensä jaettu noin kymmenen toisiaan lähellä sijaitsevan koneen *segmentteihin*, jotka on kytketty yhteen segmenttien välistä tietoliikennettä säätelevillä *silloilla* tai *reitittimillä*.

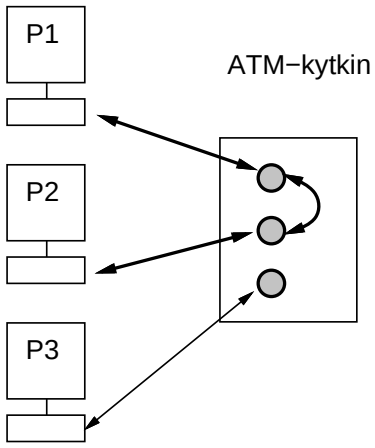
useampi kone yrittää kirjoittaa väylälle yhtä aikaa, tapahtuu *yhteentörmäys*, kirjoitusyritys peruutetaan ja kukin kone odottaa satunnaisesti valitun lyhyen ajan ennen kuin yrittää lähettää sanomansa uudelleen.



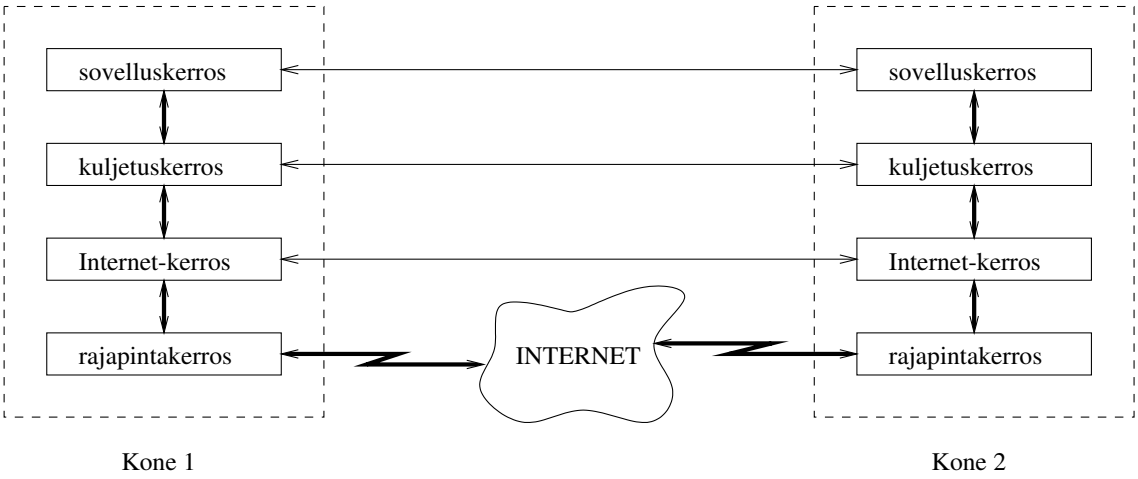
Kuva 10.5: *Valtuusrengas.*

Valtuusrengas yhdistää koneet renkaaksi, jonka käyttö on määrätty siten, että kaikki viestit kiertävät rengasta samaan suuntaan. Yhteentörmäyksen välttämiseksi kone voi lisätä renkaaseen viestin vain silloin, kun kohdalle saapuu vapaan viestin *vuoromerkki* (engl. "token"). Vuoromerkin käyttö on tehty helpoksi jakamalla verkko rekistereillä yhtämoneen osaan kuin verkossa on koneita. Alkuperäisessä IBM:n kehittämässä verkossa (IEEE standard 802.5) rekisterien koko on yksi bitti, jolloin viestin ensimmäinen bitti kertoo onko kysessä vuoromerkki vai muu viesti. Vuoromerkin tapauksessa rekisterin kohdalla oleva kone voi muuttaa bitin arvon osoittamaan viestiä ja laittaa tämän jälkeen oman viestinsä verkkoon. Koneen, joka lisää renkaaseen viestin, on myös huolehdittava sen poistamisesta ja uuden vuoromerkin sijoittamisesta renkaaseen, kun viesti on kiertänyt renkaan läpi. Menetelmä on monimutkaisempi, mutta jonkin verran tehokkaampi kuin kilpavarausväylä. Nykyään verkosta on olemassa myös variaatioita, joissa rekisterien koko on suurempi, jolloin verkkoa voi kertää useampi viesti samanaikaisesti.

Väyläratkaisuja huomattavasti nopeampi, mutta myös kalliimpi menettely on asentaa verkkoon nopea keskuskytkin, joka muodostaa kulloinkin kommunikoivien koneiden välille suoran yhteyden. Tällaiset piirikytkentäiset *ATM-verkot* (kuva 10.6) ovat parhaillaan yleistymässä verkkojen tiedonsiirtovaatimusten kasvamisen ja kytkinten kehittymisen myötä.



Kuva 10.6: *ATM-verkko.*



Kuva 10.7: *Vuorovaikuttavat TCP/IP-protokollapinot.*

Lähiverkoista laajaverkkoihin siirryttäessä tiedonsiirron ongelmat monimutkaistuvat, kun datapaketit joudutaan reitittämään lähiverkosta toiseen useiden, mahdollisesti epäluotettavien välitysverkkojen kautta. Internet-verkossa noudatetaan ns. TCP/IP-protokollaa (engl. "Transmission Control Protocol/Internet Protocol"), joka määrittelee nelitasoisen vuorovaikuttavien yhteyskäytäntöjen hierarkian (kuva 10.7). Mallissa on sovellettu luvun 9.3 virtuaalikoneideaa tietoliikenteeseen. Alimman tason *rajapintakerroksen* ohjelmat huolehtivat fyysisen tiedonsiirron tehtävistä ja sen päälle rakentuvan *Internet-kerroksen* ohjelmat sanomien reitittämisestä useiden aliverkkojen kautta. Internet-kerroksen palveluja käyttävä *kuljetuskerros* varmistaa lähetettyjen pakettien perillemenon kuittausmenettelyä käyttäen. Ylimpänä oleva *sovelluskerros* sisältää sitten muille ohjelmille tarkoitetut tiedonsiirtopalvelut, kuten tiedostojensiirto-, sähköposti- ja etäkäyttöprotokollat.

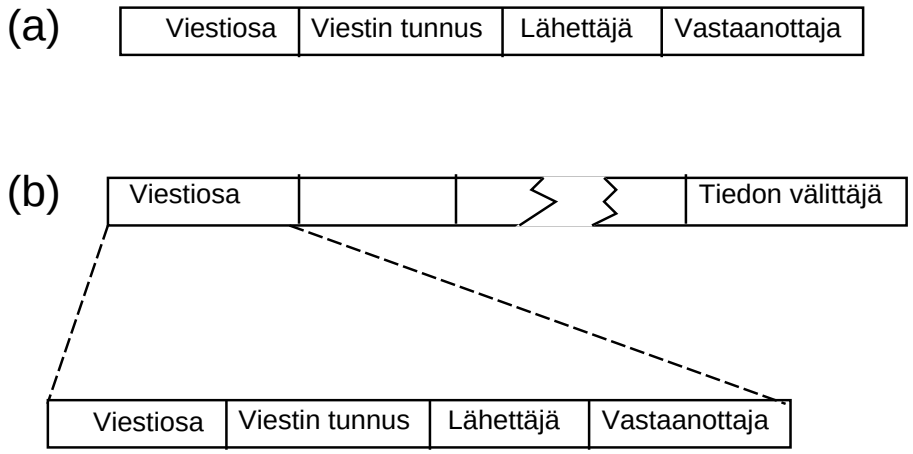
10.5 Viestien välitys tietoverkossa

Pohjimmiltaan tiedonsiirto on useamman rinnakkaisen laitteen kommunikointia jaetun resurssin, väylän, kanssa. Samat perusperiaatteet, joita hyödynnetään tietokoneen sisäisessä toiminnassa ovat käytössä myös laajemminkin verkkotasolla. Tässä yhteydessä käsittelemme aihetta kuitenkin varsin pintapuolisesti.

Fyysisellä tasolla tiedonsiirto voi olla joko rinnakkaista tai sarjamuotoista, eli signaalit voidaan kuljettaa yhdessä tai useammassa johtimessa. Radioverkoissa johdinta vastaavat taajuuskanavat tai signaalin voimakkuustasot, riippuen siitä millä tavalla signaali koodataan (moduloidaan). Tietotekniikan näkökulmasta tämä on vain toteutustekninen kysymys.

Lähetettävä informaatio pakataan viesteihin, *paketteihin*, joiden rakenne riippuu käytettävästä teknologiasta. Hieman yleistäen pakkaus tapahtuu lisäämällä lähetettävän informaation alkuun tunnisteita viestin luonteesta (tyypistä), lähettäjästä ja vastaanottajasta. Kukin OSI-mallin kerros lisää viestiin omia tunnisteitaan. Yksinkertaisen paketin rakenne on esitetty kuvassa 10.8a, missä ensimmäisenä on vastaanottajan tunnus ja lopussa itse lähetettävä informaatio. Ajatus on siis sama kuin kirjekuoressa, jonka päällä on osoite ja sisällä viesti.

Viestin kulkiessa erityyppisten välityspalveluiden läpi, nämä lisäävät viestiin omat kehyksensä, mikä vastaan suunnilleen samaa ajatusta kuin kirjekuoren sijoittaminen toisen kirjekuoren sisään. Esimerkiksi kuvan 10.8a alkuperäinen viesti voidaan pakata kuvan 10.8b tavalla laajemman kehyksen (isomman kirjekuoren) sisään. Tätä ajatusta hyödyntävät useat erityyppiset tiedonvälityspalvelut, joiden ei tarvitse tul-



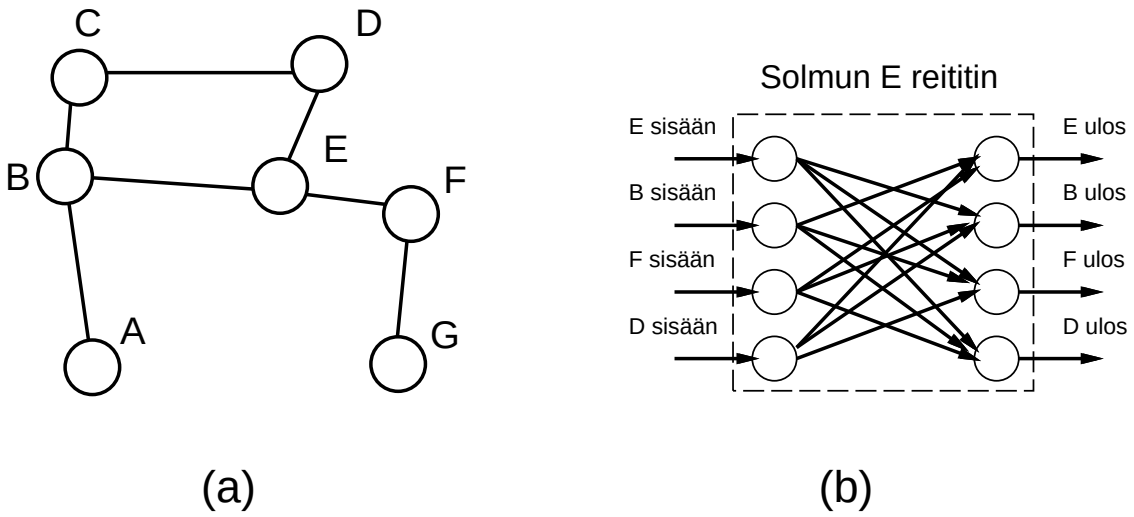
Kuva 10.8: *Yksinkertaisen viestipaketin rakenne (a) ja viestin sijoittaminen kehyksen (engl. Frame) sisään (b).*

kita koko viestiä, vaan ainoastaan päällimmäisen kehyksen osoite ja tunnustiedot. Kaikki muu on palveluiden näkökulmasta osa lähetettävää viestiä (dataa).

Kun viesti on pakattu asianmukaisella tavalla, voidaan se ikäänkuin postittaa, jolloin tietoverkon reitityspalvelut osaavat kuljettaa sen eteenpäin. Olettakaamme, että olemme lähettämässä viestiä kuvan 10.9a mukaisessa verkossa. Yksinkertaisimmillaan kukin verkon solmu tietää mitä seuraavaa linkkiä pitkin tiettyyn kohteeseen menevä viesti pitää lähettää. Solmut toimivat siten puhelinkeskuksena, yhdistäen hetkellisesti kaksi naapuriaan.

Tällöin viesti, joka on lähetetty solmusta A solmuun G on mahdollista reitittää usealla tavalla. Reitinvalinta on optimointitehtävä, joka riippuu paitsi reitin lyhyydestä, niin myös sen kuormitusasteesta. Yksinkertaisimmillaan kaikki reititystavat määrätään ennakkoon, jolloin viestit kulkevat aina samaa reittiä solmujen välillä. Monimutkaisemmassa toimintatavassa välitettävä informaatio voi kulkea useana pakettina eri reittejä pitkin lähettäjältä vastaanottajalle.

Periaatteessa, ja usein myös käytännössä, viestin reititys tapahtuu hajautetusti tietoverkon solmukoneissa toimivien itsenäisten reitittimien toimesta. Nämä ovat siis ikäänkuin paikallisia postikeskuksia. Solmukoneen (postikeskuksen) tarvitsee tietää ainoastaan mille naapurille osoitteen viesti tulee lähettää. Saadessaan viestin naapuri toimii samalla tavalla. Lopulta viestin tulisi päätyä solmuun, jonka osoite on sama kuin viestin osoite.



Kuva 10.9: *Reititys tietoverkossa. Kukin solmu tietää mitä linkkiä pitkin viestin on kuljettava, jotta se lopulta päätyisi oikeaan osoitteeseen. Solmut siirtelevät viestejä toisilleen, kunnes viesti saapuu määränpäähänsä.*

Yhdessä solmut muodostavat reititystaulukon, joka esimerkkimme verkolle voisi olla kuvan 10.10 mukainen.

Esimerkiksi solmun A lähettämä viesti solmulle G kulkisi reittiä:
$A \rightarrow B$, $B \rightarrow E$, $E \rightarrow F$, $F \rightarrow G$.

Normaaleille käyttäjille reititystaulujen käyttäminen on kuitenkin hankalaa, joten osoitteiden määräämiseen tarvitaan käyttäjäystävällisempi ratkaisu. Nimi- ja reitityspalveluiden tarkoitus on vapauttaa käyttäjä reittien määräämisestä käsin. Tämä onnistuu, kun verkkoon tehdään yleinen ja kattava nimipalvelu, jossa kullakin koneella on nimi. Nimet tallennetaan "globaaliin" nimien tietopankkiin, joka on ositettu suurempiin ja pienempiin kokonaisuuksiin, esim. `jane.mit.jyu.fi`:

jane - Tietokone
mit - Tietotekniikiikan laitos
jyu - Jyväskylän yliopisto
fi - Suomi

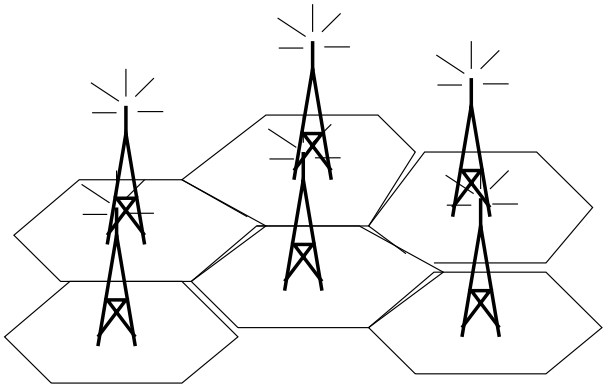
Nyt verkossa oleva kone (esim. "lenna.luth.se") voi selvittää lyhyimmän polun verkkoon ".fi" pyytämällä reittiä nimipalveluita hoitavalta tietokoneelta.

	Lopullinen vastaanottaja						
Solmu	*A*	*B*	*C*	*D*	*E*	*F*	*G*
A	-	B	B	B	B	B	B
B	A	-	C	C	E	E	E
C	B	B	-	D	D	D	D
D	C	C	C	-	E	E	E
E	B	B	B	D	-	F	F
F	E	E	E	E	E	-	G
G	F	F	F	F	F	F	-

Kuva 10.10: *Reititystaulukon periaate. Kukin rivi on yhden solmun (tietoverkon koneen) reititysohje, joka kertoo mille naapurille viesti on lähetettävä, jotta se lopulta päätyisi vastaanottajalle.*

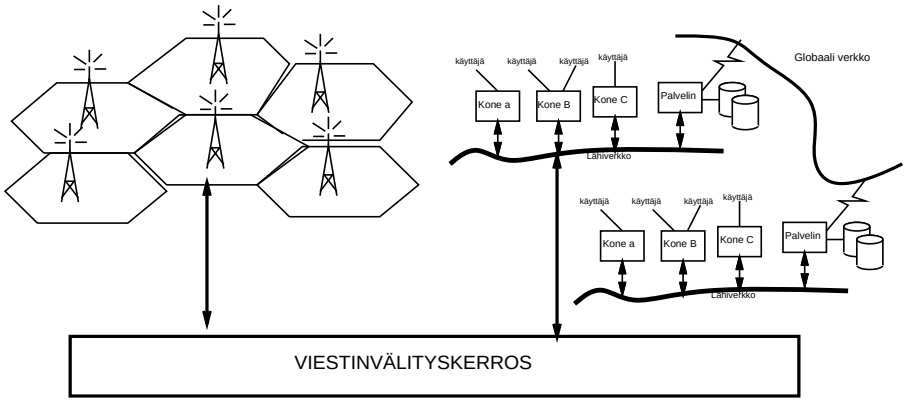
10.6 Langaton tiedonsiirto

Langaton tiedonsiirto on tullut suuren yleisön tietoisuuteen kännyköiden ja GSM-järjestelmän myötä. Näissä on käytössä selli-tyyppinen tekniikka, eli *soluverkko*, joissa käytetään suhteellisen pienitehoisia tiedonsiirtojärjestelmiä alueellisesti jaoteltuina. Näin saadaan paremmin käyttöön jaetut resurssit (taajuudet). Kaksi pääteknologiaa ovat TDMA (Time Division Multiplexing) ja CDMA (Carrier Sense Multiplexing) protokollat. Nämä edustavat (tietyssä mielessä) sarjamuotoista ja rinnakkaista tiedonsiirtoa.



Kuva 10.11: *Soluverkon periaate.*

Solverkon periaate (kuva 10.11) on jakaa radiotaajuusverkko maantieteellisesti pieniin osiin käyttämällä heikkotehoisia lähettimiä. Näin samoja taajuuksia (tiedonsiirtokanavia) voidaan käyttää uudelleen tietyn välimatkan päässä, ja järjestelmä pystyy suoriutumaan suuremmasta määrästä tiedonvälitystä. Tietyssä mielessä yhden lähettimen ja siihen yhteydessä olevien laitteiden (GSM-puhelimet) järjestelmä vastaa lähiverkkoa ja soluista muodostettu kokonaisuus laajaverkkoa.



Kuva 10.12: *Soluverkon yhdistäminen osaksi globaalia tietoverkkoa.*

Nykyään langattomassa tiedonsiirrossa on etupäässä kyse puheen siirrosta tai erikoissovelluksista (satelliitit, kannettavat tietokoneet). Tulevaisuudessa myös kannettavien tietokoneiden ja henkilökohtaisten kommunikaattorien myötä voidaan olettaa data-tyyppisen tiedonsiirron merkityksen kasvavan, mistä esimakua tuo WAP-teknologia. Jo nyt on nähtävissä langattomien ja Internet-tyyppisten tietoverkkojen yhdentyminen (kuva 10.12). Tällä hetkellä kommunikointi verkkojen välillä on sovelluskohtaista, mutta tulevaisuudessa lienee odotettavissa laajempia standardeja, jotka mahdollistavat läpinäkyvät palvelut kaikkien verkkoihin kytkettyjen järjestelmien välille.

Langattomuus on siirtymässä myös koteihin. Esimerkiksi kelpaa vaikkapa Bluetooth, joka on johtavien laitevalmistajien kehittelemä standardi langattomalle tiedonsiirrolle ja sitä tukeville mikropiireille. Tavoitteena on halpa (10-100 mk) piiriperhe, jota voi käyttää kodinkoneissa ja kulutuselektroniikassa. Useat valmistajat, kuten Eriksson, Nokia ja Intel ovat kehittäneet visioita verkotetusta kodista, jossa tietokoneet, kellot, radiot ym. ovat verkotettuja ja kommunikoivat toistensa kanssa. Ensimmäisenä markkinoille ehtinevät langattomat modeemit ja taskutieturit. Nämä laitteet toimivat hyvin pienellä teholla, 10-100 metrin säteellä.

Luku 11

Tietokannat

11.1 Tiedonhallinta

Suurten tietomäärien hallinta on tietotekniikan tärkeimpiä sovelluksia. Tietokoneet soveltuvat erinomaisesti suurten, järjestyneiden datajoukkojen käsittelyyn, ja nykyisin kaikki vähänkin tärkeämmät rekisterit on jo tietokoneistettu (hallinnon henkilö- ja asiarekisterit, kauppojen asiakas- ja tuoterekisterit, pankkien tilitiedot, lentoyhtiöiden paikanvaraus, yliopiston opiskelijarekisteri jne.).

Organisaatioympäristössä tiedonhallintasovellusten suunnitteluun vaikuttavat monet muutkin kuin puhtaasti tietotekniset seikat, sillä organisaation tiedonhallinta ja sen muu toiminta ovat kiinteässä yhteydessä toisiinsa. Automatisoitua tiedonhallintaa suunniteltaessa joudutaan selvittämään ja ottamaan kantaa siihen, minkälaisia rekistereitä organisaatiossa on (tai tulisi olla), mitä tietoa tarvitaan, mistä se saadaan, kenen käyttöön se tulee, ja lukuisia muita tämänkaltaisia seikkoja. Käytännön atk-tehtävissä vaikeinta onkin usein tarvittavien tietojen määrittely ja jäsentäminen — ohjelmointi yms. tietotekninen työ on tämän jälkeen pelkkää rutiinia. Tietojenkäsittelyä organisaatioympäristössä, tietotekniikan tasoa "ylemmästä" näkökulmasta tarkastelee *tietojärjestelmätiede* (engl. "information systems science").

Tekniseltä kannalta tiedonhallinnan keskeinen työkalu on *tiedonhallintaohjelmisto*. Perinteisesti nämä ovat olleet teksti- ja numerotiedon tallentamiseen ja käsittelyyn suunniteltuja, keskitettyjä *tietokantajärjestelmiä*, mutta tietotekniikan monimuotoistuminen asettaa tiedonhallinnallekin jatkuvasti uusia haasteita: miten voidaan esimerkiksi ääni- ja kuvatiedon hallinta järjestää tehokkaasti, tai miten pitäisi rakentaa tiedonhallintajärjestelmä, jonka sisältämät tiedot sijaitsevat hajautettuina

organisaation sisäisen tietoverkon solmukoneissa (ns. intranet-ympäristö), tai jopa eri puolille laajaa ja epäluotettavaa Internet-verkkoa sijoiteltuina?

11.2 Tietomallit

Ohjelmoijan näkökulmasta tietokanta on työkalu, joka tarjoaa joukon palveluita, joita ainakaan laajemmassa mielessä ei kannata toteuttaa omalla ohjelmoinnilla. Toisaalta tietokannan käyttö liittää sovelluksen osaksi monimutkaista järjestelmää, jonka niksien hallinta voi olla vaikeampaa, kuin oman pienen tietojenhallintajärjestelmän ohjelmointi. Se milloin sovellus todella tarvitsee tietokantaohjelmiston käyttöä onkin valintakysymys. Samalla tavalla on vaikeaa päättää mitkä tiedot tulee tallentaa tietokantaan ja mitkä tiedot taas ei. Nyrkkisääntönä voi pitää ajatusta, että pidemmäksi aikaa tallennettava tieto on järkevää tallentaa, kun taas ohjelman väliaikaisten tietorakenteiden tallentaminen raskaaseen tietokantaesitykseen ei ole järkevää.

Tietokantapohjaisia järjestelmiä kehitetään lähinnä suurten tietomassojen hallintaan. Tietokannan avulla pyritään takaamaan tallennettavien tietojen yhteensopivuus tilanteissa, jossa on mahdollisesti useita tiedon tuottajia, tuotantotapoja tai käyttäjiä. Perusajatuksena on käyttää kaiken tiedon tallentamiseen yhtenäistä *tietomallia*, joka on täsmällinen formalismi sille, miten reaalimaailman oliot, tapahtumat ja niiden suhteet esitetään tietokoneessa. Tietomallien päävaihtoehdot ovat:

Hierarkkinen tietomalli, jossa tiedot tallennetaan puurakenteeksi. Esimerkiksi korjaamon tietokanta voi sisältää tiedot autoista. Jokainen erityyppinen auto on tietokannan solmu, jonka alta löytyvät tiedot tämän autotyypin osista, joiden alta löytyvät tiedot osien valmistajista jne. Nykyään hierarkkisia tietomalleja käytetään harvoin, sillä niiden joustavuus tiedon jäsentelyyn on heikko.

Verkkorakenteinen tietomalli, jossa jokainen tietoalkio voidaan linkittää mihin tahansa toiseen tietoalkioon. Malli antaa siis varsin vapaat kädet suunnittelijalle, mutta samalla ajatus tietojen yhtenäisyydestä kärsii. Verkkomallia onkin vaikea ylläpitää, mistä käy osoituksena Internetin WWW-sivusto. Käyttäjiä kiusaavat epäkelvot linkit syntyvät, kun WWW-sivu (tietoalkio) poistetaan ottamatta huomioon, että sivuun on viittauksia muista sivuista.

Relaatiomalli, on tällä hetkellä käytetyin tietomalli. E. F. Coddin vuonna 1970 esittelemässä mallissa tiedot esitetään yksinkertaisesti taulukoina, ns. *relaatio-*

tauluina, joissa kukin ryhmä yhteen olioon tai tapahtumaan liittyviä ominaisuuksia muodostaa taulukon yhden rivin. Käsitteellisestä selkeydestään huolimatta relaatiomalli ei ole suinkaan helppo toteuttaa tietokoneella tehokkaasti (relaatiotaulut ovat usein hyvin suuria, ja niitä *ei* toteuteta ohjelmointikielten taulukkorakenteilla). Pitkälle 1980-luvulle asti vallitsikin laajalti käsitys, että relaatiomalli on kyllä teoreettisesti tyylikäs, mutta käytännössä toteuttamiskelvoton idea, ja toteutuksissa suosittiin käsitteellisesti mutkikkaampia mutta toteutukseltaan yksinkertaisempia tietomalleja, ns. verkkomallia ja hierarkista tietomallia. Nykyisin relaatiomalli on kaikkien yleisesti käytettyjen tietokantajärjestelmien perustana (Access, dBase, Ingres, Oracle, Paradox jne.)

Oliomallin ajatuksena on tallentaa samaan tietokantaan sekä tieto että sen käsittelymenetelmät. Tällöin raakatiedosta, esimerkiksi digitaalisista kuvista, voidaan tuottaa jo tietokannan puolella korkeamman tason tietoa. Esimerkiksi haettaessa kuvia autoista, voisi kukin tieto-olio tutkia oman raakatietonsa ja palauttaa kysyjälle vain ne kuvat, jotka ovat yhteensopivia hakukriteerin kanssa. Tällä hetkellä oliotietokannat ovat vasta suunnitteluasteella. Markkinoilla on tosin muutamia oliotietokantoja, joissa oliomallia ei kuitenkaan ole toteutettu loppuun asti.

Koska relaatiomalli on tällä hetkellä käytetyin tietomalli, keskitytään seuraavissa luvuissa sen tarkasteluun.

11.3 Relaatiotaulut

Tarkastellaan esimerkkinä kuvan 11.1 relaatiotaulua, joka sisältää joidenkin Jyväskylän yliopiston tietotekniikan laitoksen henkilökuntaan kuuluvien tietoja. Kuvan esittämän relaatiotaulun *nimi* on JYTIE, ja se sisältää 12 *riviä* t. *monikkoa*, joista kukin kuvaa yhtä tietotekniikan laitoksen työntekijää. Kukin taulukon kuudesta sarakkeesta vastaa yhtä henkilöstä tallennettua ominaisuutta eli *attribuuttia*. Tässä tapauksessa taulun attribuutit ovat henkilön *sukunimi*, *etunimi*, hänen edustamansa tietotekniikan *linja* (ohjelmistotekniikka/ tietoliikenne/ tieteellinen laskenta), henkilön *virka* (taulukon henkilöiden tapauksessa lehtori/ laboratorioinsinööri/ professori/ assistentti/ yliassistentti), henkilön *huoneen* numero sekä (keksitty) nelinumeroinen henkilökuntatunnus *hetu*.

Relaatiotaulun *avain* on jokin sellainen minimaalinen attribuuttijoukko, joka yksilöi taulun rivit. Taulun JYTIE ilmeinen avainattribuutti on henkilökuntatunnus

sukunimi	*etunimi*	*linja*	*virka*	*huone*	*hetu*
Ernvall	Jarmo	ohjelmistot	leht	MaD304	8276
Hämäläinen	Pentti	ohjelmistot	leht	MaD309	1354
Hämäläinen	Timo	tietoliik	labins	AS330	2359
Joutsensalo	Jyrki	tietoliik	prof	MaE222	5430
Koikkalainen	Pasi	ohjelmistot	prof	MaE333	2243
Kärkkäinen	Kari	tietlask	ass	MaE325	9572
Kärkkäinen	Tommi	ohjelmistot	prof	MaE321	2772
Lappalainen	Vesa	ohjelmistot	leht	MaD305	3458
Männikkö	Timo	tietlask	yliass	MaE328	8127
Neittaanmäki	Pekka	tietlask	prof	MaE317	9331
Santanen	Jukka	ohjelmistot	leht	MaE320	1287
Tiihonen	Timo	tietlask	prof	MaE324	9234

Kuva 11.1: *Relaatiotaulu JYTIE: tietotekniikan laitoksen henkilökuntaa.*

hetu. Henkilön sukunimi yksinään ei riitä avaimeksi, sillä taulussa on kaksikin riviparia (henkilöparia), joissa tämän attribuutin arvo on sama ("Hämäläinen", "Kärkkäinen"). Henkilön sukunimi ja etunimi yhdessä muodostaisivat avainparin taulussa JYTIE, mutta isommissa henkilötauluissa tämä ei riittäisi. (Esimerkiksi Jyväskylän puhelinluettelo vuodelta 2000 listaa neljä Pentti Hämäläistä ja viisi Timo Hämäläistä.) Taulussa JYTIE ei myöskään attribuuttikolmikko (*sukunimi, etunimi, virka*) ole määritelmän mukainen minimaalinen avain, sillä siinä on attribuutti *virka* ylimääräinen.

11.4 Relaatioalgebra ja SQL-kyselykieli

Relaatiotaulu, jossa on k attribuuttia, on matemaattiselta kannalta k-paikkainen *relaatio*, missä taulun rivit listaavat relaatioon kuuluvat k-monikot. Tämän vastaavuuden mukaisesti voidaan relaatiotauluja yhdistellä ja niistä poimia tietoja tavanomaisin joukko-opin operaatioin (relaatioiden yhdiste, leikkaus, karteesinen tulo jne.) Relaatiotaulujen manipuloinnissa käytetyille joukko-opin operaatioille, tai kuten tässä yhteydessä sanotaan "relaatioalgebralle" on tietty standardoitu esitystapa: relaatiotietokantojen *kyselykieli* SQL (engl. "Structured Query Language", ANSI/ISO-standardi 1992).

Rivien valinta

Yksi SQL-kyselykielen perusoperaatio on tietyn ehdon E täyttävien rivien *valinta* annetusta relaatiosta (taulusta) R. Matemaattisesti kyse on relaatio-operaatiosta σ_E, joka on määritelty seuraavasti:

$\sigma_E(R)$ = uusi relaatio (taulu) R', joka sisältää ne R:n monikot, joiden kohdalla ehto E on voimassa.

SQL-kielen standardinmukainen muoto tälle operaatiolle on:

```
SELECT * FROM R WHERE E
```

Esimerkiksi jos kuvan 11.1 relaatioon JYTIE sovellettaisiin rivivalintaehtoa $E \equiv$ (*etunimi* = "Timo"), saataisiin tuloksena relaatio σ_E(JYTIE) =

```
Hämäläinen  Timo  tietoliik  labins  AS330   2359
Männikkö    Timo  tietlask   yliass  MaE328  8127
Tiihonen    Timo  tietlask   prof    MaE324  9234
```

SQL-kieltä käyttävässä relaatiotietokantajärjestelmässä tämä kysely siis kirjoitettaisiin:

```
SELECT * FROM JYTIE WHERE etunimi ='Timo'
```

Sarakkeiden valinta

Toinen perusoperaatio on tietyn sarakejoukon S valinta relaatiosta R. Matemaattisesti kyse on relaation R *projektiosta* attribuuttijoukolle S:

$\pi_S(R)$ = uusi relaatio (taulu) R', joka sisältää vain joukkoon S sisältyvät R:n attribuutit (sarakkeet).

SQL-kielinen merkintä tälle operaatiolle on:

```
SELECT S FROM R
```

Esimerkiksi jos relaatiossa JYTIE valitaan $S \equiv$ (*sukunimi*, *huone*), niin saadaan tulos $\pi_S(JYTIE) =$

```
Ernvall       MaD304
Hämäläinen    MaD309
Hämäläinen    AS330
Joutsensalo   MaE222
Koikkalainen  MaE333
Kärkkäinen    MaE325
Kärkkäinen    MaE321
Lappalainen   MaD305
Männikkö      MaE328
Neittaanmäki  MaE317
Santanen      MaE320
Tiihonen      MaE324
```

Valinta- ja projektio-operaatioita voi myös yhdistellä, niin että vaikkapa SQL-kysely

`SELECT` sukunimi, linja, huone `FROM` JYTIE `WHERE` virka $=$′ prof′

tuottaa tuloksenaan taulun

```
Joutsensalo   tietoliik    MaE222
Koikkalainen  ohjelmistot  MaE333
Kärkkäinen    ohjelmistot  MaE321
Neittaanmäki  tietlask     MaE317
Tiihonen      tietlask     MaE324
```

Matemaattisesti tarkastellen tässä on kyse yhdistetystä relaatio-operaatiosta

$$\pi_S(\sigma_E(R)),$$

missä $E \equiv$ (*virka* = "prof") ja $S \equiv$ (*sukunimi, linja, huone*).

Relaatiotaulujen yhdistäminen

Yleensä relaatiotietokantaan kuuluu useita relaatioita (tauluja), jotka vastaavat kuvattavan järjestelmän eri osakokonaisuuksia. Taulujen osittaminen on tärkeää myös, jotta samaa tietoa vältettäisiin tallentamasta moneen paikkaan: yhtäpitäväksi tarkoitetun tiedon ylläpitäminen monessa paikassa johtaa monenlaisiin ongelmiin, kuten tietää jokainen jolla on vaikkapa kaksi kalenteria. Relaatiotaulujen osittamista

sukunimi	*etunimi*	*linja*	*virka*	*huone*	*hetu*
Astala	Kari	yleinen	prof	MaD359	6596
Geiss	Stefan	stokast	prof	MaD344	7894
Järvenpää	Esa	yleinen	ass	MaD369	5621
Järvenpää	Maarit	yleinen	tutk	MaD370	1265
Kahanpää	Lauri	opettaja	prof	MaD372	5887
Kuusalo	Tapani	yleinen	prof	MaD358	3474
Lehtonen	Ari	tkteoria	leht	MaD374	2354
Orponen	Pekka	tkteoria	prof	MaD307	5904

Kuva 11.2: *Relaatiotaulu JYMAT: matematiikan laitoksen henkilökuntaa.*

tietyt riippumattomuus- ja minimaalisuusehdot täyttävällä tavalla tarkastelee relaatiotaulujen *normalisointiteoria*.

Oletetaan, että Jyväskylän yliopiston organisaatiota kuvaavaan tietokantaamme kuuluisi vielä kuvan 11.2 mukainen taulu, joka sisältää tietoja yliopiston matematiikan laitoksen henkilökunnasta.

Tauluja, joilla on samat attribuutit, voidaan yhdistää tavanomaisilla joukko-operaatioilla: esimerkiksi tietotekniikan ja matematiikan laitosten tiedot saadaan samaan tauluun SQL-operaatiolla

```
(SELECT * FROM JYTIE) UNION (SELECT * FROM JYMAT)
```

Relaatiotaulujen liitos

Relaatiotietokannan taulujen tärkein yhdistämistapa on erirakenteisten taulujen liittäminen yhteen attribuuttivertailujen määräämällä tavalla. Matemaattiselta kannalta kyse on operaatiosta, jossa ensin muodostetaan lähtötaulujen R ja S karteesinen tulo

$$R_1 \times R_2 \quad = \quad \text{uusi relaatio, jonka rivit ovat kaikki mahdolliset } R_1\text{:n ja } R_2\text{:n riviparit,}$$

ja tästä sitten valitaan monikot, jotka toteuttavat halutun ehdon E:

$$\sigma_E(R_1 \times R_2) \quad = \quad \text{edellisestä ne rivi(pari)t, jotka täyttävät ehdon } E.$$

Usein halutaan vielä valitut rivit projisoida tietylle attribuuttijoukolle S:

$$\pi_S(\sigma_E(R_1 \times R_2)) \quad = \quad \text{valituista riveistä/ripareista ainoastaan joukkoon } S \text{ kuuluvat sarakkeet.}$$

kirjano	*tekijä*	*otsikko*	*lainaaja*
233	Kreyszig	Adv Eng Math	5430
481	Randell	Origins of Computers	2243
517	Kernighan	C Programming	3458
532	Ullman	Database Systems	5904
634	Feller	Probability Theory	2243
937	Goldschlager	Intro to Comp Science	5904
968	Welsh	Running Linux	3458

Kuva 11.3: *Relaatiotaulu JYKIRJAT: kirjaston kokoelmatietoja.*

SQL-kielessä vastaava liitosoperaatio kirjoitetaan:

SELECT S FROM R_1, R_2 WHERE E.

Täydennetään esimerkiksi Jyväskylän yliopiston organisaatiotietokantaamme kuvan 11.3 mukaisella taululla, joka sisältää tietoja yliopiston kirjaston kokoelmista. Nyt saadaan vaikkapa tietotekniikan laitoksella lainassa olevien kirjojen nimeketiedot ja lainaajat selville SQL-kyselyllä

```
SELECT JYKIRJAT.tekij, JYKIRJAT.otsikko, JYTIE.sukunimi
FROM JYKIRJAT, JYTIE
WHERE JYKIRJAT.lainaaja = JYTIE.hetu
```

joka tuottaa tuloksenaan relaatiotaulun

```
Kreyszig   Adv Eng Math          Joutsensalo
Randell    Origins of Computers  Koikkalainen
Kernighan  C Programming         Lappalainen
Feller     Probability Theory    Koikkalainen
Welsh      Running Linux         Lappalainen
```

Jos tietyn niminen attribuutti esiintyy vain yhdessä liitokseen osallistuvassa relaatiossa, sen lähtörelaation osoittava etuliite voidaan jättää pois. Siten edellinen kysely voitaisiin kirjoittaa myös yksinkertaisemmin:

```
SELECT tekij, otsikko, sukunimi
FROM JYKIRJAT, JYTIE
WHERE lainaaja = hetu.
```

Joko tietotekniikan tai matematiikan laitoksella lainassa olevien kirjojen tiedot saadaan selville kyselyllä

```
(SELECT tekij, otsikko, sukunimi
FROM JYKIRJAT, JYTIE
WHERE lainaaja = hetu)
UNION
(SELECT tekij, otsikko, sukunimi
FROM JYKIRJAT, JYMAT
WHERE lainaaja = hetu)
```

joka tuottaa tuloksen

```
Kreyszig      Adv Eng Math           Joutsensalo
Randell       Origins of Computers   Koikkalainen
Kernighan     C Programming          Lappalainen
Feller        Probability Theory     Koikkalainen
Welsh         Running Linux          Lappalainen
Ullman        Database Systems       Orponen
Goldschlager  Intro to Comp Science  Orponen
```

SQL yleisesti

Edellä kuvattujen operaatioiden lisäksi SQL-kyselyissä on vielä mahdollista ryhmitellä rivejä attribuuttien arvojen mukaan, laskea attribuuttien ryhmittäisiä maksimi- tai keskiarvoja, rajata vastaustauluun mukaan otettavia riviryhmiä, järjestää vastaustaulu jonkin attribuutin arvojen mukaan nousevaan tai laskevaan järjestykseen jne. Kyselyn yleinen muoto on:

```
SELECT [distinct] attribuutit FROM taulut
[WHERE ehto]
[GROUP BY attribuutit]  /* Ryhmittely                          */
[HAVING ryhmäehto]      /* Mukaan otettavien ryhmien määritys  */
[ORDER by attribuutit]  /* Lopputuloksen järjestäminen         */
```

missä hakasulkuihin merkityt osat voivat puuttua.

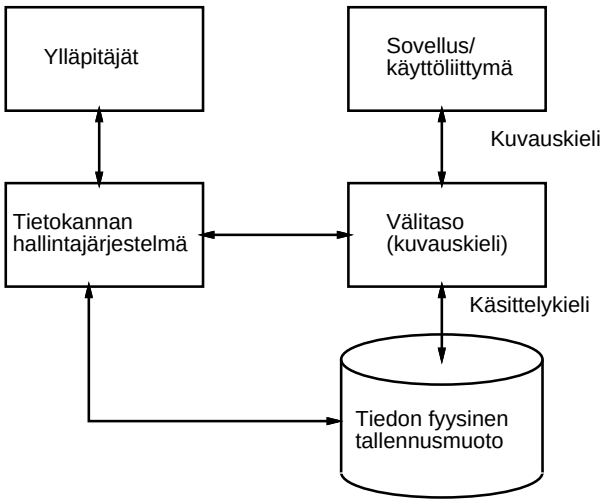
Käytännön tietokantasovelluksissa järjestelmän käyttäjä ei yleensä pääse tekemään suoraan mielivaltaisia SQL-kyselyjä tietokantaan, vaan SQL:ää käytetään vain jär-

jestelmän "välikielenä". Käyttäjä saattaa tällöin olla yhteydessä tietokantaan esimerkiksi jonkin graafisen käyttöliittymän kautta (esim. tilitietojen kysely pankkipalveluohjelmalta tai verkkoselaimessa täytettävä koneislomake), joka käyttömukavuuden lisäämisen ohella rajoittaa käyttäjälle sallittujen kyselyiden joukkoa. Kyselyt saattaa myös tuottaa jokin ohjelma oman toimintansa pohjaksi (esim. massapostituksen tarratulostusohjelma).

Kyselytoimintojen lisäksi SQL-standardi määrittelee myös joukon operaatioita relaatiotaulujen käsittelyyn — taulujen luontiin, attribuuttien määrittelyyn, taulujen päivitykseen jne. — mutta näihin ei puututa tässä.

11.5 Tietokantojen suunnittelusta

Tietokantojen ammattimaisessa suunnittelussa on käyttöön vakiintunut nk. kolmitasoinen tietokanta-arkkitehtuuri (kuva 11.4), jossa tiedon fyysinen tallennus ja sen käsittely sovellustasolla on erotettu nk. välitason (käsitetason) avulla.



Kuva 11.4: *Kolmitasoinen tietokanta-arkkitehtuuri, jossa tietokannan käyttö ja fyysinen tallennusmuoto on erotettu toisistaan.*

Kolmitasoisen tietokanta-arkkitehtuurin taustalla ovat käytännön toteutukseen ja tietokantojen ylläpitoon liittyvät seikat. Useimmissa tapauksissa tietokantaan joudutaan tekemään rakenteisia muutoksia sen käytön aikana, jolloin fyysistä tiedon-

tallennusmuotoa joudutaan vaihtamaan. Jos fyysistä ja käyttäjän näkemää käsitteellistä tasoa ei ole erotettu toisistaan, tulee muutostöistä kalliita, kenties jopa mahdottomia toteuttaa.

Kolmitasoisen arkkitehtuurin ansioista muutokset voidaan tehdä siten, että varsinainen sovellusohjelma pysyy muuttumattomana. Luonnollisesti saman fyysisen tietokannan päälle on nyt helppo tehdä myös uusia sovelluksia, sillä sovellusten käsitteet voidaan kirjoittaa jokaiselle sovellukselle yksilöllisesti.

Edellämainitun lisäksi tietokantojen ylläpitoon liittyy myös tietokannan hallintajärjestelmä, joka tarjoaa kehittäjille ja tukihenkilökunnalle joukon työkaluja tietokannan rakenteen kehittämiseksi ja ylläpitämiseksi. Ylläpitotehtävät kattavat toimintoja tietojen varmuuskopioinnista uusien sovellusten kehittämiseen ja tukemiseen. Laajoissa järjestelmissä tietokannan hallintaan kuuluvat ylläpidon lisäksi toimenpiteet järjestelmän suorituskyvyn ylläpitämiseksi sekä parantamiseksi tietojen ja tietohakujen määrän kasvaessa.

Luku 12

Algoritmit ja laskennan vaativuus

12.1 Algoritmit ja ohjelmat

Palautetaan mieliin, että *algoritmilla* tarkoitetaan yleisesti täsmällistä kuvausta jonkin (laskenta)tehtävän suoritustavasta. Kuvauksen esitystavalle ei sinänsä ole asetettu rajoituksia, mutta sen täytyy olla niin yksityiskohtainen, että menetelmä on periaatteessa täysin mekaanisesti seurattavissa. Tietokonetoteutusta varten algoritmi on kuvattava jollakin ohjelmointikielellä (Basic, Fortran, C/C++, Pascal, ko. koneen (symbolinen) konekieli, ...) Ohjelmointikielen valinnalla ei ole periaatteellista merkitystä, sillä kaikki tavanomaiset ohjelmointikielet ovat teoreettisesti yhtä ilmaisuvoimaisia — eroa on vain niiden helppokäyttöisyydessä ja hallittavuudessa, kun laaditaan yksittäistä algoritmia laajempia ohjelmistokokonaisuuksia. Algoritmin toteutus valitulla ohjelmointikielellä on *(tietokone)ohjelma*.

12.2 Esimerkki: Valintalajittelu

Tarkastellaan esimerkkinä ns. *valintalajittelualgoritmia* (engl. "selection sort") annetun n kokonaisluvun lukujonon järjestämiseksi suuruusjärjestykseen. Menetelmän idea on yksinkertaisesti etsiä kulloinkin järjestämättä olevien lukujen joukosta seuraavaksi pienin ja sijoittaa se kohdalleen, kunnes kaikki luvut on saatu järjestykseen.

Olkoot järjestettävät luvut talletettu taulukon A alkioiksi $A[1], \ldots, A[n]$. Siis jos järjestettävänä on vaikkapa lukujono $(3, 1, 4, 2)$, on algoritmin toiminnan alussa $A[1] = 3$, $A[2] = 1$, $A[3] = 4$ ja $A[4] = 2$. Yksityiskohtaisesti kuvattuna mene-

telmä on seuraava:

1. Aseta $i \leftarrow 1$.
2. Etsi taulukon alkioista $A[i], \ldots, A[n]$ pienin. Olkoon tämän indeksi $i_{\min}$.
3. Vaihda taulukon alkioiden $A[i]$ ja $A[i_{\min}]$ sisällöt keskenään.
4. Aseta $i \leftarrow i + 1$; jos $i < n$, palaa kohtaan 2.
5. Kun $i = n$, taulukon alkiot ovat suuruusjärjestyksessä.

Esimerkkijonon $(3, 1, 4, 2)$ kohdalla algoritmin suoritus etenee seuraavasti:

0. Aluksi $A = [3, 1, 4, 2]$, $n = 4$.
1. Asetetaan $i \leftarrow 1$.
2. Alkioista $A[1 \ldots 4]$ pienin on $A[2] = 1$; siis $i_{\min} \leftarrow 2$.
3. Vaihdetaan alkiot $A[1]$ ja $A[2]$ keskenään; taulukon uusi sisältö on $A = [1, 3, 4, 2]$.
4. Asetetaan $i \leftarrow 2$; koska $2 < 4$, palataan algoritmin kohtaan 2.
2. Alkioista $A[2 \ldots 4]$ pienin on $A[4] = 2$; siis $i_{\min} \leftarrow 4$.
3. Vaihdetaan alkiot $A[2]$ ja $A[4]$ keskenään; taulukon uusi sisältö on $A = [1, 2, 4, 3]$.
4. Asetetaan $i \leftarrow 3$; koska $3 < 4$, palataan algoritmin kohtaan 2.
2. Alkioista $A[3 \ldots 4]$ pienin on $A[4] = 3$; siis $i_{\min} \leftarrow 4$.
3. Vaihdetaan alkiot $A[3]$ ja $A[4]$ keskenään; taulukon uusi sisältö on $A = [1, 2, 3, 4]$.
4. Asetetaan $i \leftarrow 4$; nyt $i = n$ ja algoritmin suoritus päättyy.

Tietokonetoteutusta varten valintalajittelualgoritmi voidaan kuvata C/C++-ohjelmointikielellä kuvassa 12.1 esitettyyn tapaan.

Ohjelman kokeilemista varten rakennetaan sille vielä kuvassa 12.2 esitetty C/C++-kielinen testiympäristö, jossa järjestettävien lukujen määrää n kasvaa luvusta 10000 lukuun 100000. Ohjelma tuottaa taulukkoon A n satunnaisesti valittua kokonaislukua, järjestää ne valintalajittelualgoritmia käyttäen ja ilmoittaa paljonko järjestämiseen kului aikaa.

```
void selectsort(int n, int a[])
{
  int i, j, imin, amin;

  for (i=0; i<n-1; i++)
    {
    imin = i;
    amin = a[i];
    for (j=i+1; j<n; j++)
      {
      if (a[j] < amin)
        {
        imin = j;
        amin = a[j];
        }
      }
    a[imin] = a[i];
    a[i] = amin;
    }
}
```

Kuva 12.1: *Valintalajittelualgoritmin toteutus C/C++-ohjelmana.*

12.3 Valintalajitteluohjelman suoritusaika

Kun kuvan 12.2 testiohjelma (tiedostossa `selectsort.c`) käännetään konekielelle Linux käyttöjärjestelmän tietokonessa ja syntynyt konekoodi (tiedostossa `selectsort`) suoritetaan samassa koneessa saadaan kuvassa 12.3 esitetty tulos.

Kuvassa 12.4 on esitetty testiajon mukainen valintalajitteluohjelman suoritusajan riippuvuus lajiteltavien alkioiden määrästä. Kaavion x-akselilla on alkioiden määrä tuhansissa ja y-akselilla ohjelman suoritusaika sekunteina.

Kaavion perusteella näyttäisi ohjelman suoritusaika kasvavan kuten lajiteltavien alkioiden määrän neliö. Tällaisessa tapauksessa sanotaan, että suoritusaika on *kertaluokkaa* n^2, tai lyhyesti $O(n^2)$ (engl. "order of"), missä n on lajiteltavien lukujen alkioiden määrä.

```
#include <iostream.h>
#include <iomanip.h>
#include <time.h>
#include <stdlib.h>

#define N 100000

int main(void)
{
  int a[N], n, i;
  double c0, c1;

  cout << setw(6) << "n" << setw(6) << "time" << endl;
  for (n=10000; n<=N; n+=10000)
    {
    for (i=0; i<n; i++)
      a[i] = rand();
    c0 = double(clock());
    selectsort(n, a);
    c1 = double(clock());
    cout << setw(6) << n << setw(6)
         << 100*(c1 - c0)/CLOCKS_PER_SEC << endl;
    }

  return 0;
}
```

Kuva 12.2: *Valintalajitteluohjelman testausympäristö.*

```
> g++ -o selectsort selectsort.c
> ./selectsort
     n  time
 10000    71
 20000   282
 30000   634
 40000  1200
 50000  2049
 60000  3474
 70000  4763
 80000  6404
 90000  8204
100000 11305
```

Kuva 12.3: *Valintalajitteluohjelman testisuoritus.*

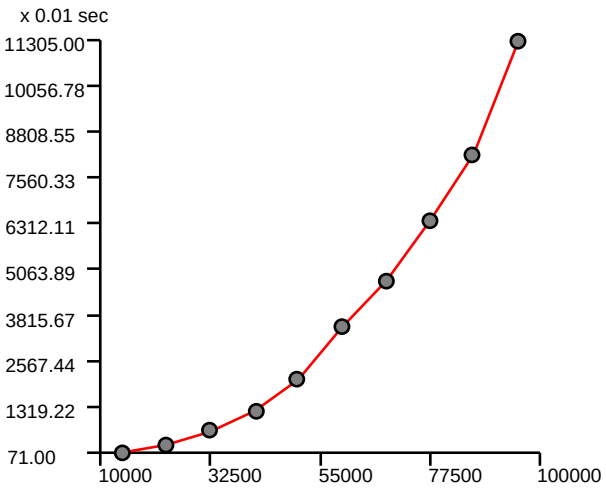
12.4 Valintalajittelualgoritmin analyysi

Valintalajitteluohjelman neliöllinen suoritusaika olisi itse asiassa voitu ennustaa myös suoraan käytettyä algoritmia analysoimalla. Kirjoitetaan menetelmä ensin hieman aiempaa täsmällisemmässä pseudokoodimuodossa:

(1) toista i:n arvoilla $i = 1, \ldots, n-1$:
(2) $\quad i_{\min} \leftarrow i$
(3) $\quad$ toista j:n arvoilla $j = i+1, \ldots, n$:
(4) $\quad\quad$ jos $A[j] < A[i_{\min}]$, niin $i_{\min} \leftarrow j$
(5) $\quad$ vaihda alkiot $A[i]$ ja $A[i_{\min}]$ keskenään.

Jos nyt oletetaan, että kunkin algoritmin suorittaman alkeisoperaation (sijoitusoperaation, alkiovertailun jne.) suoritusaika on 1 aikayksikkö, niin kullakin i:n arvolla ($i = 1, \ldots, n-1$) on:

- rivin 2 suoritusaika 1 yksikkö,
- rivien 3-4 suoritusaika $n - i$ yksikköä,
- rivin 5 suoritusaika 1 yksikkö



Kuva 12.4: *Valintalajitteluohjelman suoritusaika.*

Kaikkiaan algoritmin suoritusaika n alkion syötetaulukolla saadaan tästä summaamalla yli kaikkien i:n arvojen:

$$\begin{aligned} \text{time}(n) &= \sum_{i=1}^{n-1}((n-i)+2) &&= \sum_{j=1}^{n-1} j + 2(n-1) \\ &= \frac{1}{2}n(n-1) + 2n - 2 &&= \frac{1}{2}n^2 + \frac{3}{2}n - 2 \\ &= O(n^2). \end{aligned}$$

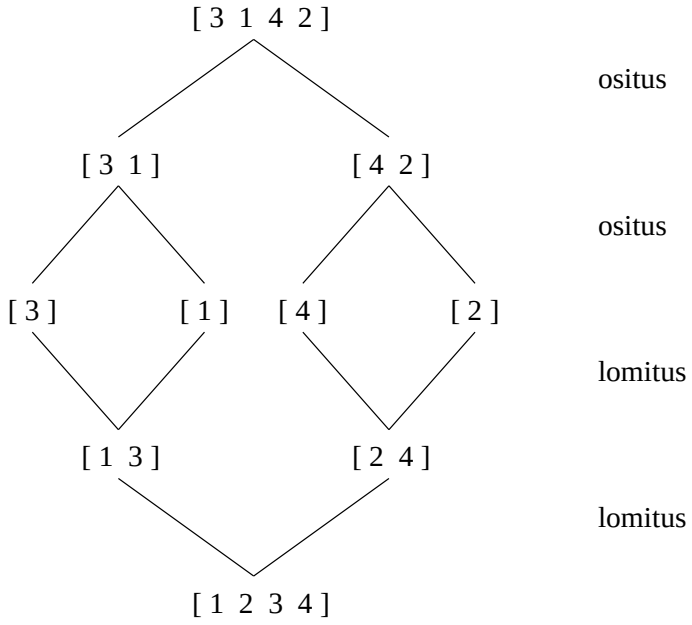
Todellisuudessa tietenkin alkeisoperaatioiden suoritusajat poikkeavat toisistaan, mutta jos ollaan kiinnostuneita vain suoritusajan kasvunopeudesta, so. kertaluokasta, tällä ei ole merkitystä.

12.5 Lomituslajittelualgoritmi

Toinen menetelmä annetun lukujonon järjestämiseen on seuraava rekursiivinen (osittava) *lomituslajittelualgoritmi*:

> Olkoot lajiteltavat luvut tallenettu taulukon A alkioiksi $A[1 \ldots n]$. Oletetaan yksinkertaisuuden vuoksi, että luku n on muotoa 2^k jollakin $k = 0, 1, 2, \ldots$

1. Jos $n = 1$, taulukossa A on vain yksi alkio, eikä mitään tarvitse tehdä.
2. Jos $n \geq 2$, jaa taulukko A kahteen yhtäsuureen puolikastaulukkoon, $A' = A[1 \ldots n/2]$ ja $A'' = A[(n/2 + 1) \ldots n]$, ja käytä tässä kuvattua menetelmää ensin kummankin puolikkaan lajitteluun erikseen.
3. Kun puolikastaulukot A' ja A'' on lajiteltu, *lomita* niiden alkiot yhteen taulukkoon A poimimalla alkioita kummastakin vuorollaan suuruusjärjestyksessä.



Kuva 12.5: *Lomituslajittelualgoritmin toiminta.*

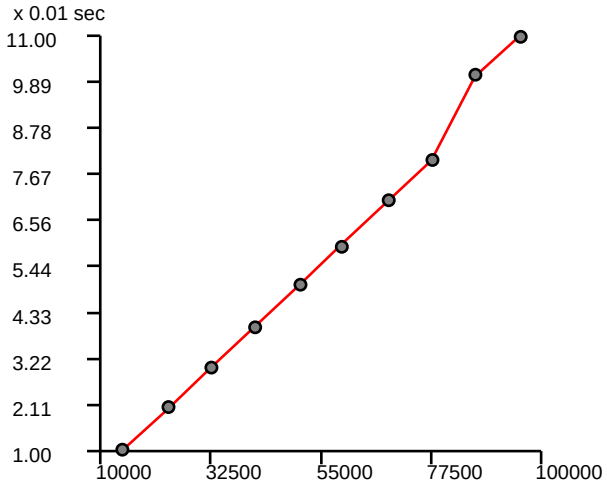
Esimerkiksi taulukon $A = [3, 1, 4, 2]$ järjestäminen tällä menetelmällä sujuu kuvan 12.5 osoittamaan tapaan.

Algoritmin kuvassa 12.6 esitetty toteutus C/C++-ohjelmana on jälleen kohtuullisen suoraviivainen. (Kuvan koodista on jätetty pois ne osat, jotka ovat samat kuin kuvien 12.1 ja 12.2 valintalajitteluohjelmassa.)

Testattaessa osoittautuu, että lomituslajittelu on huomattavasti tehokkaampi järjestämismenetelmä kuin valintalajittelu. Kuvassa 12.7 on esitetty edellisen C/C++-toteutuksen suoritusaikakaavio, missä x-akseli ilmaisee syötelukujen määrän ja y-akseli ohjelman suoritusajan sekunteina. (Huomaa että kuvan 12.7 mittakaava on

```
void merge(int m, int n, int a[], int atmp[])
{  int i, j, k;

  i = 0; j = m; k = 0;
  while (i < m && j < n) {
    if (a[i] <= a[j]) {
      atmp[k] = a[i];
      i++;
    } else {
      atmp[k] = a[j];
      j++;
    }
    k++;
  }
  while (i < m) {
    atmp[k] = a[i];
    i++;
    k++;
  }
  while (j < n) {
    atmp[k] = a[j];
    j++;
    k++;
  }
  for (i=0; i<n; i++)
    a[i] = atmp[i];
}

void mergesort(int n, int a[], int atmp[])
{  int m;

  if (n == 1) return;
  m = n/2;
  mergesort(m, &a[0], atmp);
  mergesort(n-m, &a[m], atmp);
  merge(m, n, a, atmp);
}

int main(void)
{  int a[N],atmp[N], n, i;
  ...
   mergesort(n, a, atmp);
   ...
}
```

Kuva 12.6: *Lomituslajittelualgoritmin C/C++-toteutus.*



Kuva 12.7: *Lomituslajitteluohjelman suoritusaika.*

toinen kuin kuvan 12.4.) Kaavion perusteella näyttäisi ohjelman suoritusaika nyt kasvavan vain lineaarisesti lajiteltavien lukujen määrän suhteen. Onkohan näin?

12.6 Lomituslajittelualgoritmin analyysi

Merkitään lomituslajittelualgoritmin n alkion taulukon $A[1 \ldots n]$ lajitteluun tarvitsemaa aikaa $T(n)$:llä. Oletetaan yksinkertaisuuden vuoksi, että n on jokin kakkosen potenssi ja että kunkin alkeisoperaation kesto on 1 aikayksikkö.

Helposti nähdään, että taulukon $A[1 \ldots n]$ puolittamiseen ja lajiteltujen puolikastaulukoiden yhdistämiseen kuluva aika on kertaluokkaa $O(n)$ yksikköä. Ajatellaan yksinkertaisuuden vuoksi, että tämä aika on tasan n yksikköä. (Vakiotekijän poistamisella tästä kohden ei ole vaikutusta menetelmän kokonaissuoritusajan kertaluokkaan.)

Koko algoritmin suoritusaikaa kuvaa tällöin palautuskaava t. *rekursioyhtälö*

$$\begin{aligned} T(1) &= 1, \\ T(n) &= 2T(n/2) + n, \qquad \text{kun } n = 2^k, k \geq 1. \end{aligned}$$

Voidaan osoittaa (tai tarkastaa induktiolla), että tämän rekursioyhtälön ratkaisu

suljetussa muodossa on

$$T(n) = n \log_2 n + n.$$

Lomituslajittelun aikavaativuus ei siis ole aivan lineaarinen, vaan kertaluokkaa $O(n \log_2 n)$. Kuten testiajoista nähtiin, tämä on kuitenkin huomattavasti parempi saavutus kuin valintalajittelun $O(n^2)$-suoritusaika.

12.7 Lyhimpien etäisyyksien laskeminen

Toisentyyppisenä esimerkkinä tarkastellaan seuraavaa laskentatehtävää: On annettu etäisyystaulukko $D[1 \ldots n, 1 \ldots n]$, joka kuvaa n kaupungin pareittaisia etäisyyksiä kilometreinä. Taulukkoon on kuitenkin talletettu tiedot vain suorista tieyhteyksistä kaupunkien välillä; etäisempien kaupunkiparien kohdalla taulukossa on tyhjää. Tehtävänä on täydentää taulukon puuttuvat etäisyydet kaupunkien välisten lyhimpien reittien mukaan.

Esimerkiksi syötetaulukko $D[1 \ldots 9, 1 \ldots 9]$ voisi sisältää kuvan 12.8 mukaiset, eräiden Suomen kaupunkien etäisyystiedot. Taulukko pitäisi tällöin täydentää kuvan 12.9 esittämään muotoon.

	Jkl	*Vaa*	*Jsu*	*Tre*	*Hln*	*Lti*	*Lrt*	*Tku*	*Hki*
Jkl	0	283	244	151		170	220		
Vaa	283	0		244				332	
Jsu	244		0				236		
Tre	151	244		0	75			153	
Hln				75	0	80		144	98
Lti	170				80	0	146		110
Lrt	220		236			146	0		
Tku		332		153	144			0	165
Hki					98	110		165	0

Kuva 12.8: *Eräiden Suomen kaupunkien naapurietäisyydet.*

Lyhimpien etäisyyksien määritystehtävä ratkeaa tyylikkäästi seuraavalla ns. *Floydin algoritmilla*:

1. Tallennetaan taulukon D tyhjiin paikkoihin jokin hyvin suuri luku $d_{\max}$ ("ääretön", esim. kaikkien D:n epätyhjien alkioiden summa).

	Jkl	*Vaa*	*Jsu*	*Tre*	*Hln*	*Lti*	*Lrt*	*Tku*	*Hki*
Jkl	0	283	244	151	226	170	220	304	280
Vaa	283	0	527	244	319	399	503	332	417
Jsu	244	527	0	395	462	382	236	548	492
Tre	151	244	395	0	75	155	301	153	173
Hln	226	319	462	75	0	80	226	144	98
Lti	170	399	382	155	80	0	146	224	110
Lrt	220	503	236	301	226	146	0	370	256
Tku	304	332	548	153	144	224	370	0	165
Hki	280	417	492	173	98	110	256	165	0

Kuva 12.9: *Lyhimmät etäisyydet eräiden Suomen kaupunkien välillä.*

2. Suoritetaan seuraavat kolme sisäkkäistä silmukkaa:

```
(1)  toista k:n arvoilla k = 1...n:
(2)      toista i:n arvoilla i = 1...n:
(3)          toista j:n arvoilla j = 1...n:
                 jos D[i,k] + D[k,j] < D[i,j],
                 niin aseta D[i,j] ← D[i,k] + D[k,j].
```

Algoritmin idea lyhyesti kuvattuna on, että kullakin silmukkamuuttujan k arvolla $1 \ldots n$ määritetään lyhimmät sellaiset kaikkien kaupunkiparien i, j väliset reitit, jotka kulkevat enintään kaupunkien $1 \ldots k$ kautta. Kun k saavuttaa arvon n, ovat lasketut reitit silloin absoluuttisestikin lyhimmät. Algoritmin aikavaativuus on selvästi luokkaa $O(n^3)$: kolme sisäkkäistä silmukkaa, kukin silmukkamuuttujan arvoilla $1 \ldots n$. Koska syötetaulukossa D on $N = n^2$ alkiota, vaativuus on siis luokkaa $O(N^{3/2})$ syötealkioiden määrän suhteen. Algoritmin C/C++-toteutus on esitetty kuvassa 12.10.

12.8 Kauppamatkustajan ongelma

Tarkastellaan sitten toisenlaista etäisyyksien laskentatehtävää, ns. *kauppamatkustajan ongelmaa* (engl. TSP = "travelling salesman problem"). Tässä ongelmassa annetaan syötteenä n kaupungin (täydellinen) etäisyystaulukko $D[1 \ldots n, 1 \ldots n]$. Tehtävänä on löytää lyhin sellainen kaupunkikierros, joka kulkee täsmälleen kerran kunkin

```
void floyd(int n, double d[NMAX][NMAX])
{
  int i, j, k;

  for (k=0; k<n; k++)
    {
    for (i=0; i<n; i++)
      {
      for (j=0; j<n; j++)
        {
        if (d[i][k]+d[k][j] < d[i][j])
          d[i][j] = d[i][k]+d[k][j];
        }
      }
    }
}
```

Kuva 12.10: *Floydin algoritmin C/C++-toteutus.*

kaupungin kautta — matemaattisesti sanoen siis sellainen lukujen $1 \ldots n$ permutaatio $\pi(1), \ldots, \pi(n)$, joka minimoi kustannuksen

$$c(\pi) \quad = \quad \sum_{i=1}^{n-1} D[\pi(i), \pi(i+1)] \quad + \quad D[\pi(n), \pi(1)].$$

Leikillisestä nimestään ja asettelustaan ("kauppamatkustajan" kaupunkikierroksen minimointi) huolimatta tämä ongelma esiintyy mitä moninaisimmissa käytännön sovelluksissa: jakeluautojen reittisuunnittelussa, piirilevyjen porausajan minimoinnissa, rinnakkaistietokoneiden optimaalisessa töidenjärjestelyssä, DNA-jonojen luotettavassa sekvenoinnissa jne.

12.9 Kauppamatkustajan ongelman ratkaisuyritelmiä

Vaikka TSP-ongelma päällisin puolin hieman muistuttaa lyhimpien reittien määritysongelmaa, se on laskennallisesti huomattavasti vaikeampi. Triviaaliratkaisu olisi tietenkin kokeilla n kaupungin kartalla kaikki $n!$ mahdollista reittiä ja valita niistä

lyhin. Tämä ei kuitenkaan onnistu käytännössä, sillä jos esimerkiksi $n = 22$, niin $n!$ on jo noin 10^{21}. Jos tietokoneella pystyttäisiin tutkimaan esimerkiksi 1 reitti millisekunnissa, vaatisi kaikkien reittien läpikäynti noin 36 miljardia vuotta. Vertailukohdaksi mainittakoon, että maailmankaikkeuden tähänastinen ikä on nykykäsityksen mukaan vain 10–20 miljardia vuotta.

Ensimmäinen mieleen tuleva parannusidea voisi tällöin olla ostaa nopeampi tietokone: vaikkapa miljoonan prosessorin rinnakkaiskone, jonka kukin prosessori tutkii yhden reitin nanosekunnissa. Tämä tie ei kuitenkaan johda kovin pitkälle, sillä maailmankaikkeuden ikää vastaava 20 miljardia vuotta riittää tällaisellakin koneella vasta 30 kaupungin karttojen käsittelyyn.

Parempi kehitystavoite olisi keksiä tehokkaampi algoritmi, ja tässä suunnassa onkin tehty runsaasti tutkimustyötä. Erilaiset vaihtoehtoisten reittien muodostaman hakuavaruuden karsintamenetelmät ja heuristiikat kuten ns. "simuloitu jäähdytys" ja "geneettiset algoritmit" auttavat muutamien kymmenien, joskus satojenkin kaupunkien tapauksiin asti, mutta eivät pidemmälle. Ongelma vaikuttaa siis todella vaikealta.

12.10 Polynominen ja eksponentiaalinen aika

Perusvaikeus TSP-ongelmassa ja muissa samantapaisissa tehtävissä on tutkittavien ratkaisuvaihtoehtojen (reittien) määrän $n! \approx n^n$ eksponentiaalinen kasvu (ns. "kombinatorinen räjähdys"). Jos nimittäin ratkaisuvaihtoehtojen määrä m kasvaisi vain polynomisesti syötteen koon n suhteen, niin että $m \leq n^k$ jollakin vakiolla k, konetehonkin kasvattaminen auttaisi: 10 kertaa aiempaa tehokkaammalla koneella pystyttäisiin ratkomaan $10^{1/k}$ kertaa aiempaa suurempia ongelman tapauksia. Mutta jos ratkaisuvaihtoehtojen määrä kasvaa eksponentiaalisesti, niin että $m \geq c^n$ jollakin vakiolla $c > 1$, konetehon lisääminen ei auta: tehon 10-kertaistaminen kasvattaa ratkaistavien tapausten kokoa enintään *vakiolla* $\log_c 10$.

Tämän takia "käytännössä ratkeavina" pidetään yleensä vain sellaisia ongelmia, joilla on jokin syötteen koon n suhteen polynomisessa ajassa toimiva ratkaisumenetelmä. Tällaisia ovat, aiempien esimerkkiemme mukaan, esimerkiksi lukujoukon järjestämisongelma (menetelmiä: valintalajittelu $O(n^2)$, lomituslajittelu $O(n \log_2 n)$), lyhimpien etäisyyksien laskeminen (menetelmä: Floydin algoritmi $O(N^{3/2})$). Kaikkien polynomisessa ajassa ratkeavien ongelmien luokalle käytetään merkintää P.

Kauppamatkustajan ongelmalle ei kymmenien vuosien tutkimusponnisteluista huolimatta ole onnistuttu kehittämään kaupunkien määrän n suhteen polynomisessa

ajassa toimivaa ratkaisumenetelmää, joten tällä perusteella epäillään vahvasti että TSP $\notin$ P. Polynomisen ratkaisumenetelmän olemassaoloa ei kuitenkaan ole myöskään osattu todistaa mahdottomaksi. Siten on mahdollista, joskin hyvin epäuskottavaa, että sittenkin olisi TSP $\in$ P, mutta tehokasta algoritmia ongelman ratkaisemiseen ei vain vielä ole löydetty.

Kauppamatkustajan ongelma ei ole tässä suhteessa ainoa laatuaan, vaan se kuuluu tiettyyn suureen "vaikean tuntuisten" ongelmien ekvivalenssiluokkaan, ns. *NP-täydellisiin ongelmiin*, joilla on se yhteinen ominaisuus että ne ovat joko *kaikki helppoja* (luokassa P) tai *kaikki vaikeita* (luokan P ulkopuolella). Tällä hetkellä ei tiedetä, kumpi vaihtoehto on tosi, joskin empiirisen kokemuksen pohjalta NP-täydellisten ongelmien vaikeutta pidetään jokseenkin varmana. Sitovan todistuksen löytäminen jompaan kumpaan suuntaan on kuitenkin kuuluisa ratkaisematon kysymys, ns. "P = NP"-ongelma.

Luku 13

Tekoäly

Tekoäly (engl. "Artificial Intelligence"[1]) on monirönsyinen ja muuttuva tutkimusala, jolle on vaikea antaa aivan täsmällistä määritelmää. Yksi aika osuva on:

> Tekoälytutkimuksen tavoitteena on saada tietokoneet suoriutumaan tehtävistä, joiden ratkaiseminen toistaiseksi vaatii älykkyyttä.[2]

Pragmaattinen "insinöörinäkemys" alasta voisi olla, että sen tavoitteena on yksinkertaisesti kehittää entistä korkeamman tason ohjelmistotekniikkaa. Tämän näkökulman korostamiseksi käytetään tekoälypohjaisesta ohjelmistotekniikasta usein termiä *tietämystekniikka* (engl. "Knowledge Engineering"). Historiallisesti onkin ollut niin, että tekoälyyn kuuluviksi katsotut tutkimusalat ovat muuttuneet tavanomaiseksi tietotekniikaksi sitä mukaa kun ne on saatu hallintaan: ohjelmointikielten kääntäjätekniikka oli tekoälyä 1950- ja 1960-luvuilla, logiikkaohjelmointi (Prolog) ja ns. symbolialgebra 1970-luvulla, hahmontunnistus ja ns. neurolaskenta 1980-luvulla jne. 1990-luvun saavutus voisi olla vaikkapa šakin ja muiden täsmällisesti määriteltyjen pelien siirtäminen tekoälytutkimuksesta tavanomaisen tietotekniikan puolelle.

[1] Nimen keksi tiettävästi LISP-kielen kehittäjä ja osituskäyttöjärjestelmien varhainen puolestapuhuja John McCarthy vuonna 1956 järjestetyssä ns. Dartmouthin kesäkoulussa, josta tekoälytutkimuksen nykymuodossaan katsotaan alkaneen.

[2] Teoksesta E. Rich, K. Knight, "Artificial Intelligence" (1991).

13.1 Tekoäly ja kognitiotiede

Tekoälytutkimuksella on kuitenkin myös filosofis-psykologinen puolensa. Tällä suunnalla ala niveltyy *kognitiotieteeseen*, so. älyllisten prosessien yleiseen tutkimukseen, ja tästä näkökulmasta alan filosofisesta merkityksestä ja tulevaisuudennäkymistä on käyty ajoittain kiivastakin keskustelua.

Kognitiotieteen puolella oli pitkään keskeisessä asemassa ns. *vahvan tekoälyn* tutkimusohjelma tai "paradigma". Tämän ajattelutavan mukaan kaiken älyllisen toiminnan tunnusmerkki on säännönmukainen *symbolien käsittely*, kuten kieliopillinen puhe, looginen ja matemaattinen päättely, pelit jne., ja viime kädessä kaikki älylliset toiminnot voidaan myös kuvata sopivalla kuvauskielellä esitetyn käsitesysteemin symbolimanipulaationa.[3] Tekoälytutkimuksen tavoitteena on löytää älyllisen toiminnan yleiset lainalaisuudet ja toteuttaa ne tietokoneella. Koska ihmisaivot ovat vain yksi, ja tunnetusti vajavainen tämän älyllisen toiminnan säännöstön "toteutusalusta", on tekoälytutkimuksen edistyessä ja tietokoneiden tehostuessa ajan oloon väistämätöntä että koneet ohittavat ihmisen älykkyydessä.

Luottamus vahvan tekoälyn tutkimusohjelmaan alkoi horjua 1980-luvulla mm. filosofien Hubert Dreyfus (teos *What Computers Can't Do* (1972/79)) ja John Searle (artikkeli *Minds, Brains, and Programs* (1980)) kritiikin myötä, ja myös siksi että tekoälytutkimuksen edistyminen yleisen älykkään toiminnan jäljittelyssä oli huomattavasti odotettua hitaampaa. Dreyfus ja Searle kiinnittivät huomiota ensinnäkin siihen, että ihmiset ovat erityisen hyviä juuri sellaisissa "esisymbolisissa" toiminnoissa kuten visuaalisten hahmojen tai ennaltamääräämättömien samankaltaisuuksien tunnistamisessa, joiden ohjelmointi tietokoneelle on osoittautunut ylivertaisen vaikeaksi, ja toiseksi siihen että käsitteiden aito ymmärtäminen ei ole mekaanista, vaan edellyttää "maailmassa toimimista".[4] Siten inhimillisessä mielessä älykästä toimintaa ei ehkä voikaan palauttaa mekaaniseen symbolimanipulaatioon, vaan se näyttää rakentuvan monimutkaisella tavalla ihmisten "esisymbolisten" biologisten ja sosiaalisten valmiuksien varaan — ja jos älykkyys on pohjaltaan biologinen ja sosiaalinen

[3] Selkeimmin tämän "symbolijärjestelmähypoteesin", jonka mukaan järjestelmä on älykäs, jos ja vain jos se pystyy universaaliin symbolimanipulaatioon, muotoilivat Allen Newell ja Herbert Simon artikkelissaan *The Physical Symbol System Hypothesis* vuodelta 1976.

[4] Esimerkiksi "tuolin" käsitettä ei voi tyhjentävästi määritellä kuvailemalla sen rakentuminen alikäsitteistä "jalat", "istuin", "selkänoja" tms., koska jokaiselle tällaiselle kuvailulle voidaan löytää vastaesimerkki kappaleesta, joka rikkoo annettua sääntöä ja jota silti sanottaisiin "tuoliksi". Ainoa käypä määritelmä näyttäisi olevan, että (ihmis)tuoli on mikä tahansa esine, jolla ihminen voi istua — mutta tällainen määritelmä edellyttää kokemukseen perustuvan ymmärryksen siitä, millä edellytyksillä ihminen voi istua jossakin, ja tällaista "ruumiillista" ymmärrystä tietokoneella ei tietenkään voi olla.

ilmiö, sen yleispätevä simulointi tietokoneella on vähintäänkin vaikeaa, ellei peräti mahdotonta.

Tällä hetkellä vallitseva kognitiotieteellinen tekoälytulkinta lienee ns. *heikon tekoälyn* paradigma, jonka mukaan ihmisälykkyys ei ehkä ole sama asia kuin tietokoneen tekoäly, mutta tietokoneohjelmat ovat älyllisen toiminnan hyviä *malleja*, ja niiden mahdollisuuksia ja rajoituksia tutkimalla on mahdollista oppia asioita myös inhimillisestä älykkyydestä.

13.2 Turingin testi

Brittiläinen matemaatikko ja tietojenkäsittelyteoreetikko Alan Turing esitti vuonna 1950 sittemmin "Turingin testinä" tunnetuksi tulleen operationaalisen määritelmän (oik. määritelmä*ehdotuksen*) sille, milloin voisimme sanoa tietokoneen toimintaa "älykkääksi".

Turingin kuvitteellinen testiasetelma on kyselypeli, johon osallistuu kolme pelaajaa: A (vakuuttaja), B (harhauttaja) ja C (kyselijä). Pelaaja C saa esittää A:lle ja B:lle vapaamuotoisia kirjallisia kysymyksiä, mutta ei näe heitä. Pelin versiossa I pelaajat A ja B ovat mies ja nainen, ja C yrittää kyselemällä selvittää kumpi on kumpi. A yrittää vakuuttaa C:n omasta oikeasta sukupuolestaan, kun taas B yrittää saada C:n päätymään virheelliseen lopputulokseen. Pelin versio II on muuten samanlainen, mutta pelaajat A ja B ovat nyt ihminen ja kone. Turingin määritelmän mukaan peliin osallistuvaa konetta voidaan pitää älykkäänä, jos "tyypillinen" kyselijä C erehtyy version II pelissä yhtä usein kuin version I pelissä.

Turingin testi herättää runsaasti mielenkiintoisia kysymyksiä, joista ensimmäinen on tietenkin voivatko tietokoneet koskaan läpäistä (aihepiiriltään rajoittamatonta) Turingin testiä? Inhimillistä keskustelua jäljitteleviä ohjelmia on laadittu vuosien mittaan useita, alkaen Joseph Weizenbaumin leikillisestä Eliza-"psykoterapeutista" (1966), mutta nämä ovat kaikki olleet aihealueiltaan suppeasti rajoitettuja. Tällaisten keskusteluohjelmien kesken pidetään nykyisin vuosittain jopa kilpailu, jossa palkitaan se ohjelma joka saa harhautettua useimmat tuomarit järjestetyssä "rajoitetussa Turingin testissä". Tämän ns. Loebner-kilpailun tulosten perusteella voidaan arvioida, että ainakin vielä ollaan kovin kaukana todellisen Turingin testin läpäisevän ohjelman syntymisestä.

Malliksi on kuvassa 13.1 näyte Eliza-ohjelman kanssa käydystä keskustelusta (käyttäjän kirjoittamat rivit on merkitty "$>$"-merkillä). Kuten nähdään, ohjelman toiminta vaikuttaa aluksi järkevältä, mutta jo muutaman rivin vuorovaikutuksen jälkeen

```
I am the psychotherapist.  Please, describe your problems.
Each time you are finished talking, type RET twice.
> Hello.
How do you do?  What brings you to see me?
> I suspect you are just a computer.
You have your mind on computers, it seems.
> Yes, I am reading a book about them.
Why do you say yes you are reading a book about them?
> Because that's what I am doing.
Is it because that's what you are doing that you came to me?
> You are a computer, aren't you?
You think too much about computers.
> OK, bye bye.
Earlier you said that's what you are doing?
> What?!
Can you elaborate on that?
> No.
Why not?
> BYE!
My secretary will send you a bill.
```

Kuva 13.1: *Keskustelu Eliza-ohjelman kanssa.*

tulee näkyviin, että se itse asiassa vain muuntaa käyttäjän sille antamia syötetekstejä toiseen muotoon, "ymmärtämättä" varsinaisesti lainkaan mitä keskustelu koskee.[5]

Toinen kysymys on, onko Turingin testi hyvä älykkyyden määritelmä? Sen perusteellahan ainakaan koirat ja pikkulapset eivät olisi älykkäitä, ja toisaalta keskusteluälykkään tietokoneen kyvyt eivät välttämättä riittäisi ihmisenkaltaiseen älylliseen toimintaan reaalimaailman muuttuvissa tilanteissa.

13.3 Tekoälyn (tietämystekniikan) tutkimusaloja

Tekoälypohjaisen ohjelmistotekniikan (tietämystekniikan) tutkimusalat voidaan jakaa muutamaan laajaan kokonaisuuteen.

Hahmontunnistuksen (engl. "pattern recognition") tavoitteena on kehittää automaattisia menetelmiä reaalimaailman signaalien jäsentämiseen ja luokitteluun. Reaalimaailman tuottama anturidata ei ole samalla lailla valmiiksi jäsenneltyä ja virheetöntä kuin tyypillisen tietokone-esimerkkiohjelman käsittelemät "käyttäjän antamat syötteet", vaan sisältää vaihtoehtoisia tulkintamahdollisuuksia, virheitä, vääristymiä ja kohinaa. Hahmontunnistuksen osa-alueita ovat mm. *konenäkö* (engl. "computer vision"), joka tutkii visuaalisten signaalien luokittelua, ja *puheentunnistus* (engl. "speech recognition"), joka nimensä mukaisesti keskittyy puheinformaation jäsentämiseen äänisignaaleista.

Luonnollisen kielen käsittelyssä (engl. "natural language processing") tutkitaan kirjoitetun tekstin tulkitsemista. Tässä tapauksessa syötedata on siis sinänsä valmiiksi jäsenneltyä ja "siistiä", mutta tavoitteena on sen *merkityksen* (semantiikan) määrittäminen.

Tietämyksen esittäminen (engl. "knowledge representation") ja *päättely* (engl. "inference") ovat tukialoja, joiden tehtävänä on kehittää tehokkaita tietorakenteita ja algoritmeja tekoälyjärjestelmissä tarvittavan (inhimillisen) tietämyksen esittämiseen ja siitä tehtävään tehokkaaseen (loogiseen tai intuitiiviseen) päättelyyn.

Koneoppimisen (engl. "machine learning") tavoitteena on kehittää tekniikoita, joiden avulla tietokoneohjelma voi parantaa suoritustaan luokittelu- tai ennustustehtävissä esimerkeistä tai kokemuksesta saamansa palautetiedon avulla.

[5] Tässä käytetty Eliza-ohjelman versio sisältyy kappaleessa 9.3 esimerkkinä tarkasteltuun EMACS-editorin, jossa se käynnistyy komennolla "doctor".

13.4 Tietämystekniikan sovelluksia I: Asiantuntijajärjestelmät

Asiantuntijajärjestelmät t. "älykkäät sanakirjat" ovat jonkin erityisalan tietämystä ja päättelyheuristiikkoja sisältäviä asiantuntijan apuohjelmistoja. Asiantuntijajärjestelmien idea syntyi 1970-luvun lopulla, kun yleisen tekoälyn kehittäminen näytti ajautuvan umpikujaan, ja ajateltiin että älykästä toimintaa kannattaisikin pyrkiä ensin ymmärtämään rajatuilla erikoisalueilla. Asiantuntijajärjestelmien idean ollessa tuore ja kehitystyön kiivaimmillaan 1980-luvun alussa suunniteltiin jopa ihmisasiantuntijoiden korvaamista tietokoneistetuilla at-järjestelmillä, mutta nykyisin lienee hyväksytty että tietokonejärjestelmät ovat parhaimmillaan inhimillisen asiantuntijan työn tukena.

Tunnettuja at-järjestelmiä ovat mm. spektrianalyysiohjelmisto DENDRAL, verinfektioanalyysiin laadittu MYCIN, malminetsintäjärjestelmä PROSPECTOR, tietokonejärjestelmien konfigurointiin kehitetty XCON, sekä nykyisin laajassa käytössä olevat symbolialgebraohjelmistot Macsyma, Maple ja Mathematica.

Asiantuntijajärjestelmän toteutus perustuu tavallisesti erityisalan tietämystä sisältävään *tietämyskantaan* (engl. "knowledge base") ja sitä hyödyntävään *päättelymekanismiin* (engl. "inference engine"). Tyypilliset toteutustyökalut ovat Prolog-kielen kaltaisia "sääntöjärjestelmiä" (engl. "rule-based system", "production system"). Puhtaan loogisen päättelyn lisäksi toteutuksessa voidaan käyttää myös monimuotoisempia, esim. tilastollisia tekniikoita.

13.5 Tietämystekniikan sovelluksia II: Luonnollisen kielen käsittely

Luonnollisen kielen käsittely (tietokonelingvistiikka) on joiltakin osiltaan edennyt viime vuosina nopeasti, ja sillä on runsaasti sovelluksia.

Morfologinen ja syntaksianalyysi (sanahahmojen ja lauserakenteiden jäsentäminen) on epätriviaali, mutta nykyisin melko hyvin hallittu ala, jolla mm. Helsingin yliopiston tutkimusryhmä (Kimmo Koskenniemi, Fred Karlsson) on tehnyt uraauurtavaa työtä. Työn tulokset ovat jo nykyisin laajassa käytössä mm. sanakirjaohjelmissa ja mikrotietokoneiden tekstinkäsittelyohjelmien oikeinkirjoituksen tarkastustoiminnoissa.

Semanttinen analyysi (tekstin merkityksen ymmärtäminen) on edelleen erittäin vaikeaa. Syynä tähän on kielen kontekstuaalisuus: tekstin merkitystä ei yleensä voi ymmärtää tekstin sinänsä varassa, vaan tarvitaan laajaa tietämystä niistä asioista, joihin teksti viittaa. Tämä kontekstuaalisuus ilmenee usein jopa yksittäisten ilmaisujen monitulkintaisuutena ("Kuinka voitte?", "Tien toisella puolella kasvoi viisi koivua ja toisella puolella kuusi.", "Ensin käydään saunassa ja sitten syödään vasta.")

Luonnollisen kielen semanttisella analyysilla olisi onnistuessaan paljon sovelluksia: automaattinen kielenkääntäminen, keskustelevat käyttöliittymät, automaattinen tiedonkeruu teksteistä jne. Viimemainittuun sovellukseen liittyy käynnissä oleva mielenkiintoinen CYC-tutkimushanke (Douglas Lenat alk. 1984), jossa on tarkoitus 20 vuoden kuluessa rakentaa kaiken inhimillisen "perustiedon" sisältävä tietämyskanta, jonka varassa tulevat ohjelmat voivat täydentää tietämystään itsenäisesti annettujen tekstien (sanomalehtiartikkeleita, kirjoja) avulla.

Luonnollisen kielen käsittelyn työkaluja ovat tehokkaat tietämyksen esittämisrakenteet ja päättelymenetelmät, sekä lingvistiseen tutkimukseen perustuva kieliopillinen analyysi.

13.6 Tietämystekniikan sovelluksia III: Hahmontunnistus

Hahmontunnistustutkimuksen perinteinen toiminta-ala on ollut numeerisen anturidatan, esimerkiksi ääni- tai kuvasignaalien, käsittely ja luokittelu korkeamman tason rakenteiden tunnistamiseksi. Äänidatasta pyritään jäsentämään äänteitä tai sanoja, ja kuvadatasta esineitä (esim. robotiikassa), tekstuureja (esim. satelliittidatan analyysissa) tai hahmoja (esim. käsinkirjoitetun tekstin luokittelussa).

Viime aikoina hahmontunnistuksen menetelmiä on alettu soveltaa myös symbolimuotoisen datan (esim. tietokannoissa) käsittelyyn: *koneoppimistutkimuksessa* (engl. "machine learning") kehitetään menetelmiä luokittelujen tai toimintojen automaattiseen oppimiseen esimerkeistä, ja *tiedonrikastustutkimuksessa* t. *tiedon louhinnassa* (engl. "data mining", "knowledge discovery in databases") pyritään löytämään suurissa data-aineistoissa (esim. kaupan tai hallinnon rekisterit) ilmeneviä säännönmukaisuuksia automaattisesti, ilman että järjestelmän käyttäjällä on selvää ennakkokäsitystä siitä millaisia säännönmukaisuuksia data voisi sisältää. Sovelluksia näille menetelmille on tarjolla esimerkiksi kaupan asiakasprofiilien määrittämisessä, tai monimutkaisten järjestelmien kuten tehtaiden tai tietoliikenneverkkojen vikakäyt-

täytymisen luokittelussa ja ennustamisessa.

Hahmontunnistuksen perustekniikoita ovat signaalimuunnokset, tilastolliset mallit ja testit, sekä tehokkaat algoritmit suurten tietomäärien käsittelyyn. Uusi, tai oikeastaan uudelleen kelpuutettu menetelmäperhe ovat ns. *neuroverkot*. Nämä ovat yksinkertaistettujen hermosolumallien (alunperin McCullochin–Pittsin neuronimallin vuodelta 1943) motivoima hahmontunnistus- ja koneoppimismenetelmien perhe.

Neuroverkkopohjainen hahmontunnistus oli huomattavan kiinnostuksen kohteena 1950-luvulla ja 1960-luvun alkupuolella, mutta ajautui kriisiin 1960-luvun puolivälissä, kun silloisen perustekniikan, ns. Rosenblattin *perceptron*-neuromallin rajoitukset tulivat ilmeisiksi, eikä mallin oikeaa yleistystä löydetty. Alalla tehtiin 1970-luvun ajan pienellä volyymilla laadukasta, mutta vähän huomattua työtä: mm. suomalaisen Teuvo Kohosen nyt maailmanmaineeseen nousseet perusideat ovat tuolta ajalta. Kiinnostus neuroverkkomalleja kohtaan alkoi taas lisääntyä 1980-luvun alkupuolella, kun kilpailevien symbolimanipulaatioon perustuvien tekniikkojen edistys vuorostaan hidastui, ja vuosikymmenen puolivälissä useat tutkimusryhmät toisistaan riippumatta keksivät Rosenblattin tekniikoita oikealla tavalla yleistävän ns. *vastavirta-algoritmin* (engl. “backpropagation algorithm”) neuroverkkojen syntetisointiin annetusta esimerkkidatasta. Alan tutkimus on nykyisin hyvin laajaa, ja neuroverkkotekniikoita soveltavia hahmontunnistusjärjestelmiä lienee toteutettu jo kymmeniä tuhansia.

Kirjallisuutta

Tietotekniikan alan kirjallisuus on nykyisin jo hyvin laaja. Lisää tietoa tässä monisteessa käsitellyistä perusasioista saa mm. seuraavista teoksista.

Yleisteoksia, johdatuksia

Goldschlager, L. & Lister, A. *Computer Science : A Modern Introduction, 2nd Ed.* Prentice-Hall, 1988.
Dewdney, A. K. *The (New) Turing Omnibus : 66 Excursions in Computer Science.* Computer Science Press, 1993.

Tietotekniikan historia

Williams, M. R. *A History of Computing Technology, 2nd Ed.* IEEE Computer Society, 1997.

Tietokoneen rakenne ja toiminta

Tanenbaum, A. S. *Structured Computer Organization, 4th Ed.* Prentice-Hall, 1999.
Stallings, W. *Computer Organization and Architecture, 5th Ed.* Prentice-Hall 1999.
Hennessy, J. L. & Patterson, D. A. *Computer Architecture, A Quantitative Approach, 2nd ed.* Morgan Kauffmann, 1996.

Ohjelmointikielet ja niiden toteuttaminen

Sethi, R. *Programming Languages: Concepts and Constructs, 2nd Ed.* Addison-Wesley, 1996.
Aho, A. V., Sethi, R. & Ullman, J. D. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

Käyttöjärjestelmät

Bacon, J. *Concurrent Systems : Operating Systems, Database and Distributed Systems: An Integrated Approch, 2nd Ed.* Addison-Wesley, 1998.
Stallings, W. *Operating Systems : Internals and Design Principles, 3rd Ed.* Prentice-Hall, 1998.

Tietoliikenne

Tanenbaum, A.S. *Computer Networks, 3rd Ed.* Prentice-Hall, 1996.
Stallings, W. *Data and Computer Communications, 5th Ed.* Prentice-Hall, 1997.

Tietokannat

Elmasri, R. & Navathe, S. B. *Fundamentals of Database Systems, 3rd Ed.* Benjamin/Cummings 1999.
Ullman, J. D. & Widom, J. *A First Course in Database Systems.* Prentice Hall, 1997.

Algoritmit

Weiss, M. A. *Data Structures and Algorithm Analysis in C++, 2nd Ed.* Addison-Wesley, 1999.
Cormen, T. H., Leiserson, C. E., Rivest, R. L. *Introduction to Algorithms.* The MIT Press, 1990.

Tekoäly

Russell, S. & Norvig, P. *Artificial Intelligence: A Modern Approach, 2nd Ed.* Prentice Hall, 1999.
Rojas, R. *Neural Networks : A Systematic Introduction.* Springer-Verlag, 1996.

.