Tiedon tarkistus ja muisti

Ohjelman esitysmuodot
- Konekäskyt ja rakenteellinen tieto

Tiedon muuttumattomuus
- Pariteettibitti
- Hamming-koodi
- CRC-koodi ja laitteiden monistaminen

Muisti
- Välimuisti, tavallinen muisti

Copyright Teemu Kerola 2005

Tässä luennossa esittelemme viime luennolta jääneet tiedonesitysmuodot eli konekäskyt ja rakenteellisen tiedon. Seuraavaksi käymme läpi tiedon muuttumattomuuden tarkistusmenetelmistä pariteettibitin käytön, virheitä korjaavan Hamming-koodin, CRC-koodin ja laitteiden monistamiseen perustuvien menetelmien idean. Lopuksi esittelemme lyhyesti välimuistin perusidean ja kuinka keskusmuisti on toteutettu.

Konekäskyjen esitysmuoto muistissa

Konekohtainen, joka valmistajalla omansa

- saman valmistajan jotkut suorittimet voivat olla taaksepäin yhteensopivia (suorittimen uudessa versiossa toimii aikaisemman version koodi)

Käskyt 1 tai useamman tavun mittaisia

- SPARC'issa kaikki käskyt 1 sanan eli 4 tavun mittaisia
- PowerPC'ssä kaikki käskyt 1 sanan eli 4 tavun mittaisia
- Pentium II'ssa käskyt 1-16 tavua, paljon pituuksia käytössä

Käskyillä yksi tai useampi muoto, kussakin tietty määrä erilaisia kenttiä

- opcode, Ri, Rj, Rk, osoitusmoodi
- pitkä tai lyhyt vakio

Ei omia konekäskyjä konekäskyjen manipulointiin, käytä aliohjelmia

- kääntäjä generoi konekäskyjä bittimanipulaatioilla, käyttäen valmiita aliohjelmia

Copyright Teemu Kerola 2005

Konekäskyt ovat ihan oma tietotyyppinsä. Konekäsky tietotyyppinä tarkoittaa sellaista tietoa, jonka (jokin) suoritin osaa tulkita konekäskyksi ja toimia sen mukaisesti. Jokainen tietokoneiden valmistaja on speksannut omat konekäskyjen joukkonsa. Eri suorittimilla on siis yleisesti ottaen kaikilla erilainen konekäskykanta ja ne eivät ole keskenään yhteensopivia. Intelin suorittimelle käännettyä koodia ei voi suorittaa IBM'n suorittimella ja päin vastoin. Toisaalta useat Intelin suorittimet ovat taaksepäin yhteensopivia siten, että aikaisemmalle versiolle käännetty koodi voi toimia myös uudemmassa suoritinversiossa, mutta ei päin vastoin!

Konekäskyjen esitysmuoto muistissa

Konekohtainen, joka valmistajalla omansa

- saman valmistajan jotkut suorittimet voivat olla taaksepäin yhteensopivia (suorittimen uudessa versiossa toimii aikaisemman version koodi)

Käskyt 1 tai useamman tavun mittaisia

- SPARC'issa kaikki käskyt 1 sanan eli 4 tavun mittaisia
- PowerPC'ssä kaikki käskyt 1 sanan eli 4 tavun mittaisia
- Pentium II'ssa käskyt 1-16 tavua, paljon pituuksia käytössä

ttk-91
kaikki käskyt 1 sanan eli 4 tavun mittaisia

Käskyillä yksi tai useampi muoto, kussakin tietty määrä erilaisia kenttiä

- opcode, Ri, Rj, Rk, osoitusmoodi
- pitkä tai lyhyt vakio

Ei omia konekäskyjä konekäskyjen manipulointiin, käytä aliohjelmia

- kääntäjä generoi konekäskyjä bittimanipulaatioilla, käyttäen valmiita aliohjelmia

Copyright Teemu Kerola 2005

Konekäskyjen pituus voi olla vakio tai se voi vaihdella. Jos käskyn pituus on vakio, niin se on tietenkin helpompi hakea muistista, kun jo alusta pitäen tiedetään luettavien tavujen lukumäärä. Toisaalta muistitilaa voi mennä hukkaan, jos yksinkertaisillekin käskyille varataan tilaa yhtä paljon kuin monimutkaisillekin käskyille. Yksi keino tämän ongelman välttämiseksi on toteuttaa käskykanta käyttäen ainoastaan yksinkertaisia käskyjä, jotka voidaan kaikki esittää muutamalla tavulla vakiopituisessa kentässä. Monimutkaisten asioiden tekemiseen käytetään sitten useampi konekäsky.

Konekäskyjen esitysmuoto muistissa

opcod	Rk	Ri	Rj

opcode	Rk	Ri	iso_vakio

Konekohtainen, joka valmistajalla omansa

- saman valmistajan jotkut suorittimet voivat olla taaksepäin yhteensopivia (suorittimen uudessa versiossa toimii aikaisemman version koodi)

Käskyt 1 tai useamman tavun mittaisia

- SPARC'issa kaikki käskyt 1 sanan eli 4 tavun mittaisia
- PowerPC'ssä kaikki käskyt 1 sanan eli 4 tavun mittaisia
- Pentium II'ssa käskyt 1-16 tavua, paljon pituuksia käytössä

Käskyillä yksi tai useampi muoto, kussakin tietty määrä erilaisia kenttiä

- opcode, Ri, Rj, Rk, osoitusmoodi
- pitkä tai lyhyt vakio

ttk-91
kaikki käskyt saman pituisia ja saman muotoisia

Ei omia konekäskyjä konekäskyjen manipulointiin, käytä a

- kääntäjä generoi konekäskyjä bittimanipulaatioilla, käyttäen valmiita aliohjelmia

Copyright Teemu Kerola 2005

Konekäsky koostuu aina tietueesta eli muutamasta kentästä, kuten esimerkiksi operaatiokoodista ja operandi- ja tulosrekistereistä. Tietuetyyppejä voi olla vain yksi tai niitä voi olla useampi. Jos tietuetyyppejä on vain yksi, niin käskyn purku kenttiin on helppoa, koska se tehdään aina samalla tavalla. Jos taas tietuetyyppejä on useita, niin kenttiin purku täytyy tehdä useammassa vaiheessa, jolloin siihen kuluu enemmän aikaa.

Konekäskyjen esitysmuoto muistissa

generoi konekäsky ADD R1, =56
CreateInstr(opcADD, 1, 0, 0, 56)

Konekohtainen, joka valmistajalla omansa

- saman valmistajan jotkut suorittimet voivat olla taaksep
  (suorittimen uudessa versiossa toimii aikaisemman ve

Käskyt 1 tai useamman tavun mittaisia

- SPARC'issa kaikki käskyt 1 sanan eli 4 tavun mittaisia
- PowerPC'ssä kaikki käskyt 1 sanan eli 4 tavun mittais
- Pentium II'ssa käskyt 1-16 tavua, paljon pituuksia käytö

Käskyillä yksi tai useampi muoto, kussakin tietty r

- opcode, Ri, Rj, Rk, osoitusmoodi
- pitkä tai lyhyt vakio

Käsky	Operandit	Tulos
LOAD	R1, opc	
SHL	R1, =24	17 0 0 0 0
LOAD	R2, rj	
SHL	R2, =21	
OR	R1, R2	17 1 0 0 0
LOAD	R2, m	
SHL	R2, =19	
OR	R1, R2	17 1 0 0 0
LOAD	R2, rj	
SHL	R2, 16	
OR	R1, R2	17 1 0 0 0
LOAD	R2, addr	
OR	R1, R2	17 1 0 0 56

Ei omia konekäskyjä konekäskyjen manipulointiin, käytä aliohjelmia

- kääntäjä generoi konekäskyjä bittimanipulaatioilla, käyttäen valmiita aliohjelmia

Copyright Teemu Kerola 2005

Konekäskyjen manipulointiin ei omia konekäskyjä, vaikka tietysti jokainen suoritin ymmärtää ja osaa tulkita oman arkkitehtuurinsa konekäskyjä. Kääntäjiin sisältyy aina osana konekielisen koodin generointi, jossa konekäskyt pitää rakentaa kenttä kerrallaan kuntoon. Tätä varten kääntäjissä on omia valmiiksi rakennettuja aliohjelmia, jotka sitten yhdistelevät parametreina annetut kentät uudeksi konekäskyksi.

Ttk-91 konekäskyn rakenne

Käskyn esitys bittitasolla on aina samantyyppinen

ADD R1, =56 ;Ri turha
LOAD R2, R3 ;ADDR turha
NOP ; Rj, M, Ri ;ADDR turhia
LOAD R4, =12345678

(bitit on numeroitu oikealta vasemmalle, nollasta alkaen)

OPER **operaatiokoodi, opcode, 0-112 (kaikki arvot eivät ole laillisia)**
Rj **1. operandi, tulosrekisteri, R0-R7 (yleisrekisteri)**
Ri **indeksirekisteri, R1-R7 (arvo 0 indikoi että indeksirekisteri ei ole käytössä)**
ADDR **osoiteosa, vakio-osa, arvoalue [-32767, +32767]**
M **moodi, muistinoutojen määrä toisen operandin laskemiseksi (ennen mahdollista muistiin talletusta STORE-käskyssä)**

- 0 välitön osoitus (STORE-käsky: suora osoitus)
- 1 suora osoitus (STORE-käsky: epäsuora osoitus)
- 2 epäsuora osoitus (STORE-käsky: epäkelpo arvo, virhetilanne)
- 3 epäkelpo arvo, virhetilanne

Copyright Teemu Kerola 2005

Ttk-91 koneen konekäskyn rakenne on tarkasti käyty läpi jo aikaisemmilla luennoilla. Käskyt ovat kaikki 32-bittisiä ja niissä on aina kaikki samat kentät. Joissakin tapauksissa kaikkia kenttiä ei käytetä, joten ne ovat tavallaan turhia. Esimerkiksi pelkkää vakiota käytettäessä kenttää Ri ei tarvittaisi ja toisaalta pelkästään rekistereillä operoitaessa vakiokenttä on ihan turha. NOP-käskyssä kaikki muut kentät paitsi operaatiokoodi ovat turhia. Toisaalta, konekäskyssä ei voi esittää suuria vakioita tai muistiosoitteita, joten ne täytyy käsitellä jollain muulla tavalla.

Ttk-91 konekäskyn rakenne

LOAD R2, @ptrY

ADD R1, mjaX

Käskyn esitys bittitasolla on aina samantyyppinen

(bitit on numeroitu oikealta vasemmalle, nollasta alkaen)

OPER **operaatiokoodi, opcode, 0-112 (kaikki arvot eivät ole laillisia)**
Rj **1. operandi, tulosrekisteri, R0-R7 (yleisrekisteri)**
Ri **indeksirekisteri, R1-R7 (arvo 0 indikoi että indeksirekisteri ei ole käytössä)**
ADDR **osoiteosa, vakio-osa, arvoalue [-32767, +32767]**
M **moodi, muistinoutojen määrä toisen operandin laskemiseksi**
(ennen mahdollista muistiin talletusta STORE-käskyssä)

- 0 välitön osoitus (STORE-käsky: suora osoitus)
- 1 suora osoitus (STORE-käsky: epäsuora osoitus)
- 2 epäsuora osoitus (STORE-käsky: epäkelpo arvo, virhetilanne)
- 3 epäkelpo arvo, virhetilanne

Copyright Teemu Kerola 2005

Ttk-91 muistinosoitumuotoja on oikeastaan vain 2 kappaletta: suora ja epäsuora muistinosoitus. Rekisterioperandien tai välittömien operandien käyttö ei varsinaisesti on muistissa olevaan tietoon viittaamista, vaikka tällainen tiedon viittaustyyppi määritelläänkin saman moodikentän avulla. Epäsuora muistiviite on aika hidas toteuttaa, koska se vaatii kaksi muistiviitettä, minkä vuoksi sitä ei useinkaan enää käytetä. Toisaalta, ttk-91'ssä voi yhdistää muistista noudon ja matemaattisen operaation, mikä myöskin aiheuttaa toteutustasolla ongelmia, minkä vuoksi nykyaikaisissa koneissa varsinainen työ tehdään usein vain rekistereillä operoiden.

Taulukkojen esitysmuoto

Peräkkäisrakenteena, kuten aikaisemmin esitettiin

Ei yleensä omia konekäskyjä, manipulointi aliohjelmilla tai silmukoilla

Vektorisuorittimissa on konekäskyjä lyhyiden vektoreiden (1-ulotteisten taulukoiden) manipulointiin

- vektorirekisterit tavallisten rekistereiden lisäksi
  - 32 kaksoistarkuuden liukuluvun rekisteri
  - 64-bitin rekisteri (Intelin MMX konekäskyt)
    - Tavallinen kaksoistarkkuuden liukulukurekisteri toimii vektorirekisterinä
- vektorikonekäskyt vain vektorirekistereille: vload, vadd, vmult, vmultadd, vstore
  - MMX: PADDB (Packed Bytes Add), PSUBB (Packed Bytes Substract), etc.

32 * 64 = 2048 bitin rekisteri

8 * 8 bittiä tai
4 * 16 bittiä tai 2 * 32 bittiä

Indeksoitu tiedonosoitusmoodi tukee 1-ulotteisten taulukoiden käyttöä

- joissakin koneissa tiedonosoitusmoodeja 2-ulotteisille taulukoille

Copyright Teemu Kerola 2005

Rakenteellisen tiedon perustyypit ovat taulukot ja tietueet. Taulukot esitetään muistissa peräkkäisrakenteena, joko riveittäin tai sarakettain talletettuna, kuten aikaisemmin esitettiin. Taulukoiden dimensioiden lukumäärä samoin kuin sen alkion pituus tai tietotyyppi voi olla mitä vain.

Taulukkojen esitysmuoto

Peräkkäisrakenteena, kuten aikaisemmin esitettiin

Ei yleensä omia konekäskyjä, manipulointi aliohjelmilla tai silmukoilla

Vektorisuorittimissa on konekäskyjä lyhyiden vektoreiden (1-ulotteisten taulukoiden) manipulointiin

- vektorirekisterit tavallisten rekistereiden lisäksi
  - 32 kaksoistarkuuden liukuluvun rekisteri
  - 64-bitin rekisteri (Intelin MMX konekäskyt)
    - Tavallinen kaksoistarkkuuden liukulukurekisteri toimii vektorirekisterinä
- vektorikonekäskyt vain vektorirekistereille: vload, vadd, vmult, vmultadd, vstore
  - MMX: PADDB (Packed Bytes Add), PSUBB (Packed Bytes Substract), etc.

32 * 64 = 2048 bitin rekisteri

8 * 8 bittiä tai
4 * 16 bittiä tai 2 * 32 bittiä

Indeksoitu tiedonosoitusmoodi tukee 1-ulotteisten taulukoiden käyttöä

- joissakin koneissa tiedonosoitusmoodeja 2-ulotteisille taulukoille

Copyright Teemu Kerola 2005

Yleisesti ottaen taulukoita käsitellään aliohjelmilla. Suoritin voi tukea esimerkiksi yksiulotteisia liukulukutaulukoita, mutta ei oikein hyvin 5-ulotteisia taulukoita, joiden alkio on 8-sanainen tietue. Tällaisissa tapauksissa taulukkoon viittaminen tapahtuu aina kahdessa vaiheessa, jossa ensin lasketaan viitatun alkion osoite talletetun taulukon sisällä ja sitten vasta tehdään muistiviite kyseiseen alkioon.

Taulukkojen esitysmuoto

Peräkkäisrakenteena, kuten aikaisemmin esitettiin

Ei yleensä omia konekäskyjä, manipulointi aliohjelmilla tai silmukoilla

Vektorisuorittimissa on konekäskyjä lyhyiden vektoreiden (1-ulotteisten taulukoiden) manipulointiin

- vektorirekisterit tavallisten rekistereiden lisäksi
  - 32 kaksoistarkuuden liukuluvun rekisteri
  - 64-bitin rekisteri (Intelin MMX konekäskyt)
    - Tavallinen kaksoistarkkuuden liukulukurekisteri toimii vektorirekisterinä
- vektorikonekäskyt vain vektorirekistereille: vload, vadd, vmult, vmultadd, vstore
  - MMX: PADDB (Packed Bytes Add), PSUBB (Packed Bytes Substract), etc.

32 * 64 = 2048 bitin rekisteri

8 * 8 bittiä tai
4 * 16 bittiä tai 2 * 32 bittiä

Indeksoitu tiedonosoitusmoodi tukee 1-ulotteisten taulukoiden käyttöä

- joissakin koneissa tiedonosoitusmoodeja 2-ulotteisille taulukoille

Copyright Teemu Kerola 2005

On kuitenkin tietokonearkkitehtuureja, jotka on optimoitu taulukkojen käsittelyyn. Erityisesti tieteelliseen laskentaan eli numeroiden murskaamiseen suunnitellut ns. superkoneet sisältävät usein erillisiä vektorirekistereitä ja konekäskyjä niillä operointiin. Vektorin pituus voi olla vaikkapa 32 liukulukua, jolloin myös sovellus täytyy huolella suunnitella hyödyntämään tämän pituisia vektoreita. Fortran-kääntäjät on viilattu usein tällaiseen laskentaan. Toisaalta, multimediaa varten myös tavallisissa pöytäkoneissa käytetyt Intelin suorittimet sisältävät MMX-vektorikäskyjä, joilla operoidaan esimerkiksi kahdeksaa 8-bittistä tai neljää 16-bittistä data-alkiota kerrallaan.

Taulukkojen esitysmuoto

```
LOAD R1, R5
MULT  R1, =pituusAlkio
ADD   R1, Tbl
LOAD  R1, @R1
```

Peräkkäisrakenteena, kuten aikaisemmin esitettiin

Ei yleensä omia konekäskyjä, manipulointi aliohjelmilla tai silmukoilla

Vektorisuorittimissa on konekäskyjä lyhyiden vektoreiden (1-ulotteisten taulukoiden) manipulointiin

- vektorirekisterit tavallisten rekistereiden lisäksi
  - 32 kaksoistarkuuden liukuluvun rekisteri (32 * 64 = 2048 bitin rekisteri)
  - 64-bitin rekisteri (Intelin MMX konekäskyt) (8 * 8 bittiä tai 4 * 16 bittiä tai 2 * 32 bittiä)
    - Tavallinen kaksoistarkkuuden liukulukurekisteri toimii vektorirekisterinä
- vektorikonekäskyt vain vektorirekistereille: vload, vadd, vmult, vmultadd, vstore
  - MMX: PADDB (Packed Bytes Add), PSUBB (Packed Bytes Substract), etc.

Indeksoitu tiedonosoitusmoodi tukee 1-ulotteisten taulukoiden käyttöä

- joissakin koneissa tiedonosoitusmoodeja 2-ulotteisille taulukoille

```
LOAD   R1, Tbl(R5)
LOAD  R1, d2Arr(R3)(R4) ;ei ttk-91
```

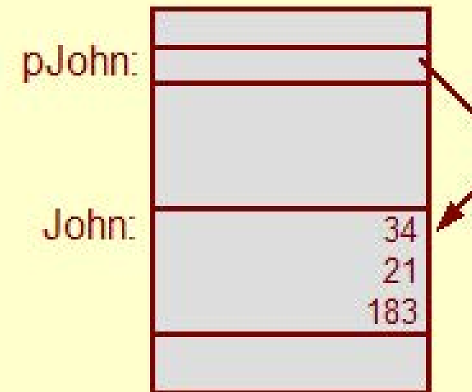
Copyright Teemu Kerola 2005

Useimmissa koneissa on laitteistotukea taulukkojen käsittelyyn indeksoidun tiedonosoitusmoodin muodossa. Sen avulla yksiulotteiseen taulukkoon viittaaminen on helppoa ja se voidaan toteuttaa yhdessä konekäskyssä. Ilman indeksoitua tiedonosoitusmoodia simppelikin taulukkoviite vaatisi ainakin kolme tai neljä konekäskyä, kun taulukon kantaosoite ja alkion pituudella painotettu indeksi pitäisi laskea ensin yhteen viitatun alkion osoitteen laskemiseksi. Indeksoitu tiedonsoitusmoodi tekee tämän suoraan laitteiston piirillä yhden konekäskyn sisällä.

Tietueiden esitysmuoto

Peräkkäisrakenteena

- yhtenäinen alue muistissa



Osoite on jonkin osoitinmuuttujan arvo

Ei omia konekäskyjä, manipulointi aliohjelmilla tai kääntäjän generoimien vakiolisäysten avulla

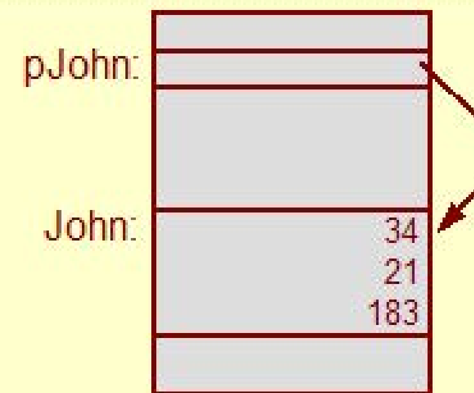
Indeksoitu tiedonosoitusmoodi tukee tietueisiin viittaamista

```
id      EQU 0
age     EQU 1
height EQU 2
pJohn DC 0   ; ptr to John
...
; allocate John
; set pJohn to point to John
...
LOAD R1, pJohn
LOAD R2, age(R1)
LOAD R3, height(R1)
```

Copyright Teemu Kerola 2005

Tietueiden esitysmuoto muistissa on aina peräkkäisrakenne. Tietue on talletettu muistiin yhtenäiselle muistialueelle, jonka alkuosoite on samalla tietueen osoite. Samalla tavalla kuin taulukotkin, tietueet ovat monisanaista tietoa ja ainoastaan yksi osa (kenttä) kerrallaan tuodaan rekisteriin prosessointia varten. Tietueen kenttiin viitataan aina kentän suhteellisen osoitteen ja tietueen alkuosoitteen avulla. Tietueita varataan ja vapautetaan yleensä suoritusaikana keosta. Yleensä niitä ei varata staattisesti esimerkiksi globaalien muuttujien tapaan.

Tietueiden esitysmuoto



Peräkkäisrakenteena

- yhtenäinen alue muistissa

Osoite on jonkin osoitinmuuttujan arvo

Ei omia konekäskyjä, manipulointi aliohjelmilla tai kääntäjän generoimien vakiolisäysten avulla

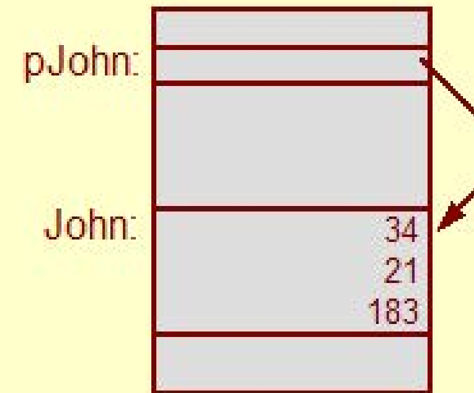
Indeksoitu tiedonosoitusmoodi tukee tietueisiin viittaamista

```
id      EQU  0
age     EQU  1
height EQU  2
pJohn DC  0   ; ptr to John
...
; allocate John
; set pJohn to point to John
...
LOAD  R1, pJohn
LOAD  R2, age(R1)
LOAD  R3, height(R1)
```

Copyright Teemu Kerola 2005

Tietueen osoite pidetään yleensä tallessa jossakin osoitinmuuttujassa tai osoitinmuuttujataulukossa tai jonkin muun tietueen osoitinmuuttujakentässä. Systeemin eheyden kannalta on tärkeätä, että tällaiset osoitinmuuttujat eivät koskaan osoita vanhaan tietoon. Esimerkiksi, jos tietueen 'John' käyttämä muistitila vapautetaan uusiokäyttöön, niin myös kaikki siihen osoittavat osoitinmuuttujat pitäisi joko nollata tai muuten merkitä epäkelvoksi. Jos näin ei tehdä, niin seuraava viite esim. osoitinmuuttujan pJohn kautta kohdistuu kyllä oikeaan muistiosoitteeseen, mutta siellä olevaan ihan eri dataan.

Tietueiden esitysmuoto



Peräkkäisrakenteena

- yhtenäinen alue muistissa

Osoite on jonkin osoitinmuuttujan arvo

Ei omia konekäskyjä, manipulointi aliohjelmilla tai kääntäjän generoimien vakiolisäysten avulla

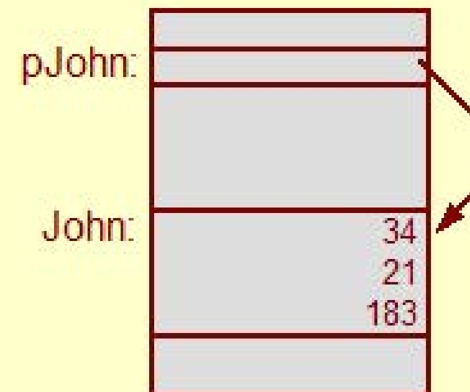
Indeksoitu tiedonosoitusmoodi tukee tietueisiin viittaamista

```
id      EQU  0
age     EQU  1
height  EQU  2
pJohn DC  0   ; ptr to John
...
; allocate John
; set pJohn to point to John
...
LOAD  R1, pJohn
LOAD  R2, age(R1)
LOAD  R3, height(R1)
```

Copyright Teemu Kerola 2005

Tietueiden käsittelyyn ei ole omia konekäskyjä, vaan niitä käsitellään kutakin tietuetyyppiä varten suunnitelluilla aliohjelmilla. Tällaiset aliohjelmat tekevät myös oikeellisuustarkistuksia sen lisäksi, että ne lukevat ja kirjoittavat tietueen eri kenttiä.

Tietueiden esitysmuoto



Peräkkäisrakenteena

- yhtenäinen alue muistissa

Osoite on jonkin osoitinmuuttujan arvo

Ei omia konekäskyjä, manipulointi aliohjelmilla tai kääntäjän generoimien vakiolisäysten avulla

Indeksoitu tiedonosoitusmoodi tukee tietueisiin viittaamista

```
id      EQU 0
age     EQU 1
height  EQU 2
pJohn DC 0   ; ptr to John
...
; allocate John
; set pJohn to point to John
...
LOAD  R1, pJohn
LOAD  R2, age(R1)
LOAD  R3, height(R1)
```

Copyright Teemu Kerola 2005

Indeksoitu tiedonosoitusmoodi tukee taulukoiden lisäksi myös tietueiden käsittelyä. Taulukkoviitteissähän indeksirekisterissä pidetään suht'koht pientä taulukon sisäistä osoitetta eli indeksiä, ja vakiokentässä on taulukon alkuosoite. Tietueviitteissä taas indeksirekisterissä pidetään tietueen alkuosoitetta eli pointteria tietueeseen ja käskyn vakio-osassa on tietueen sisäinen osoite eli tietueen kentän suhteelllinen osoite tietueen sisällä. On hyvin kätevää, että yhdellä ja samalla tiedonosoitusmoodilla voidaan tukea molempien rakenteellisen tiedon perustyypin toteutusta.

Olioiden (object) esitysmuoto

Oliot toteutetaan tietueina ja ne varataan yleensä keosta

Olioihin viitataan niiden osoitteen perusteella

Useat olion kentistä sisältävät osoitteen (pointerin) suoritusaikana varattuun toiseen olioon

Olion metodi toteutetaan kenttänä, jossa on osoite koodiin

Ei omia konekäskyjä, manipulointi muiden tietueiden tapaan aliohjelmilla

object StackA

private data
public data
private methods
public methods
other data

Copyright Teemu Kerola 2005

Oliot ovat käytännössä vain yhden tyyppisiä tietueita. Oliolle on erittäin tyypillistä, että niitä varataan ja vapautetaan suoritusaikana keosta. Olioperustaisissa järjestelmissä olioon viittaaminen tapahtuu hyvin rajoitetusti, jolloin myös kirjapito olioiden käyttämistä muistialueista ja niihin viittaavista osoitinmuuttujista voidaan hoitaa järjestelmällisesti. Useissa järjestelmissä (esim. Java) itse osoitinmuuttujiin ei suoraan pääse käsiksi, vaan olioihin voi ainoastaan viitata ohjelmoirtikielen omien rakenteiden kautta. C-kielessä ei ole tällaista rajoitetta, ja se onkin tiedostettu erääksi C:n heikkoudeksi. Jonkun systeemigurun mielestä taas tämä on juuri eräs C:n hyvistä puolista. Pitäkää varanne!

Olioiden (object) esitysmuoto

Oliot toteutetaan tietueina ja ne varataan yleensä keosta

Olioihin viitataan niiden osoitteen perusteella

Useat olion kentistä sisältävät osoitteen (pointerin) suoritusaikana varattuun toiseen olioon

Olion metodi toteutetaan kenttänä, jossa on osoite koodiin

Ei omia konekäskyjä, manipulointi muiden tietueiden tapaan aliohjelmilla

object StackA

private data
public data
private methods
public methods
other data

Copyright Teemu Kerola 2005

Olioon viittamiseksi meillä täytyy siis olla tiedossa sen muistiosoite. Muiden tietuetyyppien tapaan se on talletettu johonkin osoitinmuuttujaan, joka puolestaan voi sijaita taulukossa tai jossakin toisessa tietueessa eli oliossa. Olion perimien ominaisuuksien määrittely voi sijaita jossakin toisessa oliossa, johon jokin kenttä tässä oliossa viittaa.

Olioiden (object) esitysmuoto

Oliot toteutetaan tietueina ja ne varataan yleensä keosta

Olioihin viitataan niiden osoitteen perusteella

Useat olion kentistä sisältävät osoitteen (pointerin) suoritusaikana varattuun toiseen olioon

Olion metodi toteutetaan kenttänä, jossa on osoite koodiin

Ei omia konekäskyjä, manipulointi muiden tietueiden tapaan aliohjelmilla

object StackA

Copyright Teemu Kerola 2005

Metodit ovat yksinkertaisesti aliohjelmien alkuosoitteita. Tottakai metodin määrittelyyn sisältyy tieto siitä, onko se julkinen vai ainoastaan tämän olion sisältä kutsuttavissa samoin kuin kaikkien parametrien määrittelyt. Olion toteutus konekielen tasolla tuntuu ehkä turhankin yksinkertaiselta ja tällöin helposti unohtuu, että oliokielten voima on juuri hienoissa ohjelmistorakenteissa, joiden avulla automaattisesti hoidetaan olioiden tietorakenteiden käytön ja eheyden valvonta sekä eri tavoin eri oliokielissä toteutettu ominaisuuksien perinnät ja muut olioihin liittyvät voimakkaat käsitteet kuten esimerkiksi polymorfismi.

Olioiden (object) esitysmuoto

Oliot toteutetaan tietueina ja ne varataan yleensä keosta

Olioihin viitataan niiden osoitteen perusteella

Useat olion kentistä sisältävät osoitteen (pointerin) suoritusaikana varattuun toiseen olioon

Olion metodi toteutetaan kenttänä, jossa on osoite koodiin

Ei omia konekäskyjä, manipulointi muiden tietueiden tapaan aliohjelmilla

Copyright Teemu Kerola 2005

Olioden käsittelyyn ei ole omia konekäskyjä ja oliota käsitellään muiden tietueiden tapaan aliohjelmilla. Yleensä nämä aliohjelmat on toteutettu ohjelmistoympäristössä valmiiksi, kuten esimerkiksi Javassa.

Tiedon tarkistus

Tiedon oikeellisuutta ei voi tarkistaa yleisessä tapauksessa.

Tavoitteena on kerran oikein talletetun tiedon muuttumattomuus

- muistiin talletettu X'n arvoksi luku 5.43, X luetaan, onko arvo 5.43
- väylälle kirjoitetaan luku 1234567, mitä tulee ulos väylältä toisesta päässä?

Viasta johtuva virhe voidaan ehkä havaita ja joskus automaattisesti korjata ennen kuin siitä tulee häiriötä

- bitti voi muuttua muistissa tai tiedon siirrossa aiheuttaen virheen
  - muistipiirissä voi olla staattinen vika, joka voi aiheuttaa staattisen virheen
  - transientti vika on esimerkiksi (esim. avaruussäteilyn) alkeishiukkanen, joka muuttaa bitin tiedonsiirron aikana ja aiheuttaa transientin virheen
- korjaamattomasta virheestä voi aiheutua (pienempi tai suurempi) häiriö
  - Matin palkkasumma oli väärin? Videokuvassa oli yhdessä kuvassa tahra?

Tietokannan eheys on eri asia, vaikka sekin on virhe

- henkilörekisterin tietueessa on Pekan palkka, mutta Matin kumulatiivinen vuosipalkka

Copyright Teemu Kerola 2005

Yleisesti ottaen ei ole mitenkään mahdollista, että voisimme tarkistaa, onko jokin tieto laitteistossa oikein tai 'totta'. Tavoitteemme on paljon rajallisempi. Pyrimme vain säilyttämään laitteistoon talletetun tiedon sellaisenaan olettaen, että se on alkuaan ollut tarkoitetun mukaista. Yritämme siis eri konstein taata, että tieto ei muutu esimerkiksi muistitalletuksen tai tiedonsiirron aikana. Tässäkään ei päästä absoluuttiseen tarkkuuteen vaan meidän on tasapainoteltava kustannusten kanssa. Lopputuloksena on tilanne, jossa tiedon muuttumattomuus on hoidettu 'riittävän hyvin' eli jollakin hyvin suurella todennäköisyydellä.

Tiedon tarkistus

Tiedon oikeellisuutta ei voi tarkistaa yleisessä tapauksessa. Tavoitteena on kerran oikein talletetun tiedon muuttumattomuus

- muistiin talletettu X'n arvoksi luku 5.43, X luetaan, onko arvo 5.43
- väylälle kirjoitetaan luku 1234567, mitä tulee ulos väylältä toisesta päässä?

Viasta johtuva virhe voidaan ehkä havaita ja joskus automaattisesti korjata ennen kuin siitä tulee häiriötä

- bitti voi muuttua muistissa tai tiedon siirrossa aiheuttaen virheen
  - muistipiirissä voi olla staattinen vika, joka voi aiheuttaa staattisen virheen
  - transientti vika on esimerkiksi (esim. avaruussäteilyn) alkeishiukkanen, joka muuttaa bitin tiedonsiirron aikana ja aiheuttaa transientin virheen
- korjaamattomasta virheestä voi aiheutua (pienempi tai suurempi) häiriö
  - Matin palkkasumma oli väärin? Videokuvassa oli yhdessä kuvassa tahra?

Tietokannan eheys on eri asia, vaikka sekin on virhe

- henkilörekisterin tietueessa on Pekan palkka, mutta Matin kumulatiivinen vuosipalkka

Copyright Teemu Kerola 2005

Tiedon muuttumattomuuteen liittyy käsitteet vika, virhe ja häiriö. Staattinen vika on jokin laitteistossa oleva yleensä fyysinen vika. Esimerkiksi muistipiirin bitti on rikki ja siihen kirjoitettu tieto tallentuu aina 1:nä. Viasta tai jostain muusta syystä tieto voi muuttua virheelliseksi, joka sinällään ei vielä ole kovin paha juttu. Vasta kun virheestä aiheutuu tiedon käyttäjälle häiriötä, se muuttuu ongelmaksi ja silloinkin vain, jos häiriö on laadultaan merkittävä. Ongelmana on, että laitteisto ei tietoa käsitellessään tiedä, minkä lajin tieto on kyseessä ja onko mahdollinen häiriö vakava vai ei.

Tiedon tarkistus

Tiedon oikeellisuutta ei voi tarkistaa yleisessä tapauksessa.
Tavoitteena on kerran oikein talletetun tiedon muuttumattomuus

- muistiin talletettu X'n arvoksi luku 5.43, X luetaan, onko arvo 5.43
- väylälle kirjoitetaan luku 1234567, mitä tulee ulos väylältä toisesta päässä?

Viasta johtuva virhe voidaan ehkä havaita ja joskus automaattisesti korjata ennen kuin siitä tulee häiriötä

- bitti voi muuttua muistissa tai tiedon siirrossa aiheuttaen virheen
  - muistipiirissä voi olla staattinen vika, joka voi aiheuttaa staattisen virheen
  - transientti vika on esimerkiksi (esim. avaruussäteilyn) alkeishiukkanen, joka muuttaa bitin tiedonsiirron aikana ja aiheuttaa transientin virheen
- korjaamattomasta virheestä voi aiheutua (pienempi tai suurempi) häiriö
  - Matin palkkasumma oli väärin? Videokuvassa oli yhdessä kuvassa tahra?

Tietokannan eheys on eri asia, vaikka sekin on virhe

- henkilörekisterin tietueessa on Pekan palkka, mutta Matin kumulatiivinen vuosipalkka

Copyright Teemu Kerola 2005

Virheellistä tietoa voi järjestelmään syntyä myös ohjelmistovirheiden vuoksi, mutta jätämme tällaisten virheiden havaitsemisen ja korjaamisen muille kursseille. Tyypillinen esimerkki ohjelmistovirheestä on samanaikaisuudenhallinnan puutteellinen toteutus, minkä seurauksena esimerkiksi tietokantaan samaan aikaan tulleet kaksi päivitystä menivät osittain päällekkäin. Tässä tapauksessa henkilörekisterin tietueen kentissä olevat tiedot ovat kenttäkohtaisesti oikein, mutta yhdessä epäkonsistentteja. Samanaikaisuuden aiheuttamia ongelmia käsitellään myöhemmin useammallakin kurssilla, mutta ei tällä kertaa tämän enempää.

Tiedon muuttumattomuuden valvonta

Perusidea: otetaan tietoon mukaan ylimääräisiä bittejä, joiden avulla virheitä voidaan havaita ja ehkä myös korjata

- enemmän bittejä, enemmän tietoa, havaitaan useampi virhe, korjataan useampi virhe

Järjestelmä suorittaa tarkistukset automaattisesti joko laitteistotasolla tai ohjelmistotasolla

- laitteistotasolla nopea, kiinteä tarkistus
  - esim. rekisterissä olevan tiedon kaikkien yhden bitin virheiden korjaus
- ohjelmistotasolla hitaampi, joustava, muokattavissa oleva tarkistus
  - tilinumeron kaikkien yhden bitin virheiden havaitseminen tarkistusnumeron avulla

Valvonnan kustannukset sekä tilassa että ajassa

- ylimääräiset bitit vievät muistitilaa, vaativat lisää johtimia ja vievät tilaa lastulla
- tarkistuksen toteutus laiteistolla vie tilaa lastulla ja ehkä hidastaa suoritusta
- tarkistusohjelman suoritus vie suoritinaikaa ja siis hidastaa suoritusta

Copyright Teemu Kerola 2005

Perusidea tiedon muuttumattomuuden valvonnassa on ottaa mukaan ylimääräisiä bittejä. Mitä useampi ylimääräinen bitti on mukana, sitä todennäköisemmin virheet havaitaan ja voidaan korjata. Jos tarkistusbittejä ei ole lainkaan, niin mitään virheitä ei havaita saati sitten korjata. Jos taas tarkistusbittejä on huomattavasti enemmän kuin tietobittejä, niin on varsin todennäköistä, että kaikki tietobiteissä olleet virheet voidaan korjata. Toisaalta, näin katastrofaalisen tilanteeseen varautuminen yleensä indikoi tilanteesta, jossa tavalliset tiedon muuttumattomuuden valvontamenetelmät eivät enää riitä. Palaamme asiaan myöhemmin tällä luennolla.

Tiedon muuttumattomuuden valvonta

Perusidea: otetaan tietoon mukaan ylimääräisiä bittejä, joiden avulla virheitä voidaan havaita ja ehkä myös korjata

- enemmän bittejä, enemmän tietoa, havaitaan useampi virhe, korjataan useampi virhe

Järjestelmä suorittaa tarkistukset automaattisesti joko laitteistotasolla tai ohjelmistotasolla

- laitteistotasolla nopea, kiinteä tarkistus
  - esim. rekisterissä olevan tiedon kaikkien yhden bitin virheiden korjaus
- ohjelmistotasolla hitaampi, joustava, muokattavissa oleva tarkistus
  - tilinumeron kaikkien yhden bitin virheiden havaitseminen tarkistusnumeron avulla

Valvonnan kustannukset sekä tilassa että ajassa

- ylimääräiset bitit vievät muistitilaa, vaativat lisää johtimia ja vievät tilaa lastulla
- tarkistuksen toteutus laiteistolla vie tilaa lastulla ja ehkä hidastaa suoritusta
- tarkistusohjelman suoritus vie suoritinaikaa ja siis hidastaa suoritusta

Copyright Teemu Kerola 2005

Osa tarkistuksista tapahtuu niin matalalla tasolla, että ne on pakko suorittaa laitteistotasolla. Tämä tarkoittaa sitä, että esimerkiksi konekäskyn suorituspiireissä on virheitä tarkistavia ja korjaavia osia. Laitteistotason virheentarkistukset suoritetaan aina eikä niitä voi myöhemmin muuttaa. Ohjelmistotasolla taas voi tarkistaa ihan mitä vain, mutta tähän tietenkin voi kulua merkittäväkin määrä suoritusaikaa.

Tiedon muuttumattomuuden valvonta

Perusidea: otetaan tietoon mukaan ylimääräisiä bittejä, joiden avulla virheitä voidaan havaita ja ehkä myös korjata

- enemmän bittejä, enemmän tietoa, havaitaan useampi virhe, korjataan useampi virhe

Järjestelmä suorittaa tarkistukset automaattisesti joko laitteistotasolla tai ohjelmistotasolla

- laitteistotasolla nopea, kiinteä tarkistus
  - esim. rekisterissä olevan tiedon kaikkien yhden bitin virheiden korjaus
- ohjelmistotasolla hitaampi, joustava, muokattavissa oleva tarkistus
  - tilinumeron kaikkien yhden bitin virheiden havaitseminen tarkistusnumeron avulla

Valvonnan kustannukset sekä tilassa että ajassa

- ylimääräiset bitit vievät muistitilaa, vaativat lisää johtimia ja vievät tilaa lastulla
- tarkistuksen toteutus laiteistolla vie tilaa lastulla ja ehkä hidastaa suoritusta
- tarkistusohjelman suoritus vie suoritinaikaa ja siis hidastaa suoritusta

Copyright Teemu Kerola 2005

Tiedon muuttumattomuuden valvonta maksaa sekä tilassa että ajassa. Ylimääräiset tarkistusbitit pitää tallettaa muistiin ja rekistereihin, ja niitä varten tulee olla ylimääräisiä johtimia väylissä. Laitteistototeutuksena olevat tarkistusohjelmat vievät lastulla tilaa, jota voisi muuten käyttää vaikkapa suurempaan välimuistiin. Ohjelmakoodina toteutetut tarkistusohjelmat pitää tallettaa muistiin ja niille tulee antaa suoritinaikaa. On tärkeätä, että nämä kaikki kustannukset tasapainotetaan tarkistuksilla saataviin hyötyihin ja mahdollisiin häiriöiden aiheuttamiin riskeihin. Tiedon muuttumattomuuden valvonta ei siis mitenkään ole yksinkertaista toteuttaa.

Esimerkki ohjelmistotason tarkistusmerkki henkilötunnuksessa

Henkilötunnus: 120464-121C

120464121 % 31 = 12

tarkistusmerkki: 0123456789ABCDEFHJKLMNPRSTUVWXY
jakojäännös: 0 1 10 11 12 13 14 30

I, O ja Q puuttuvat sekaannuksien välttämiseksi

Tarkistusmerkin avulla voidaan tarkistaa, että mikään yksi merkki ei ole väärin

- havaitsee kaikki yhden merkin virheet
- havaittua virhettä ei voi automaattisesti korjata
- ei havaitse kaikkia kahden tai useamman merkin virheitä
- välimerkkiä ei tarkisteta lainkaan
  - '+' tarkoittaa 1800-lukua
  - '-' tarkoittaa 1900-lukua
  - 'A' tarkoittaa 2000-lukua

Copyright Teemu Kerola 2005

Hyvä esimerkki ylimääräisestä tarkistustiedosta on henkilötunnuksen tarkistusmerkki. Henkilötunnuksessa on neljä osaa. Ensimmäiset kuusi numeroa ovat syntymäaika, jossa vuosiluvusta on vain kaksi viimeistä numeroa mukana. Sitten on vuosisadan ilmaisema välimerkki ja kolminumeroinen järjestysnumero, joka on pojilla pariton ja tytöillä parillinen. Viimeinen merkki on tarkistusmerkki, joka saadaan jakamalla syntymäajan ja järjestysnumeron ilmaisema 9-numeroinen luku 31'lla ja koodaamalla jakojäännös tarkistusmerkiksi. Henkilötunnus otettiin Suomessa käyttöön vuonna 1964 ja alkuaan sen nimenä oli sosiaaliturvatunnus.

Esimerkki ohjelmistotason tarkistusmerkki henkilötunnuksessa

Henkilötunnus: 120464-121C (oikein)

120464121 % 31 = 12

tarkistusmerkki: 0123456789ABCDEFGHJKLMNPRSTUVWXY
jakojäännös: 0 1 10 11 12 13 14 30

Tarkistusmerkin avulla voidaan tarkistaa, että mikään yksi merkki ei ole väärin

- havaitsee kaikki yhden merkin virheet
- havaittua virhettä ei voi automaattisesti korjata
- ei havaitse kaikkia kahden tai useamman merkin virheitä
- välimerkkiä ei tarkisteta lainkaan
  - '+' tarkoittaa 1800-lukua
  - '-' tarkoittaa 1900-lukua
  - 'A' tarkoittaa 2000-lukua

virheellinen: 120464-123C

Error

120464123 % 31 = 14 = 'E'

Error

VAI?

virheellinen: 120464-123C
oikein: 120464-123E

Copyright Teemu Kerola 2005

Voidaan matemaattisesti todistaa, että mikä tahansa yhden merkin virhe syntymäajassa, järjestysnumerossa tai tarkistusnumerossa havaitaan. Esimerkissä viimeinen numero '1' on joko näppäilyvirheen tai muistivian takia vaihtunut numeroksi '3', jolloin henkilötunnusta seuraavan kerran käytettäessä tarkistuskoodi laskee jakojäännöksen olevan 14 eli tarkistusmerkin pitäisi olla kirjain 'E'. Koska näin ei ole, henkilötunnus havaitaan virheelliseksi. Virhettä ei voi korjata, koska sama virheilmoitus olisi voinut yhtä hyvin aiheutua tilanteesta, jossa henkilötunnuksen tarkistusmerkki olisi vaihtunut E'stä C'ksi.

Esimerkki ohjelmistotason tarkistusmerkki henkilötunnuksessa

Henkilötunnus: 120464-121C (oikein)

120464121 % 31 = 12

tarkistusmerkki: 0123456789ABCDEFGHJKLMNPRSTUVWXY

jakojäännös: 0 1 10 11 12 13 14 30

Tarkistusmerkin avulla voidaan tarkistaa, että mikään yksi merkki ei ole väärin

- havaitsee kaikki yhden merkin virheet
- havaittua virhettä ei voi automaattisesti korjata
- ei havaitse kaikkia kahden tai useamman merkin virheitä
- välimerkkiä ei tarkisteta lainkaan
  - '+' tarkoittaa 1800-lukua
  - '-' tarkoittaa 1900-lukua
  - 'A' tarkoittaa 2000-lukua

virheellinen: 120464-123*E*

OK

120464123 % 31 = 14 = 'E'

Copyright Teemu Kerola 2005

Tarkistusmerkki ei suojaa kaikilta virheiltä. Se on alkuaan suunniteltu nimenomaan tiedon syötössä tapahtuvia näppäilyvirheitä vastaan ja se toimii oikein hyvin yhden virheen paljastamiseksi. Jos virheitä on 2 tai useampi, niin on varsin mahdollista, että virhettä ei havaita. Esimerkissä viimeinen 1-merkki on vaihtunut 3:ksi, mutta tätä virhettä kompensoi täsmälleen tarkistusmerkin vaihtuminen C-kirjaimesta E-kirjaimeksi. Useamman virheen sattuessa on kuitenkin vain noin 3% (eli 1/31) todennäköisyys, että virhettä ei havaita.

Esimerkki ohjelmistotason tarkistusmerkki henkilötunnuksessa

Henkilötunnus: 120464-121C (oikein)

120464121 % 31 = 12

tarkistusmerkki: 0123456789ABCDEFGHJKLMNPRSTUVWXY

jakojäännös: 0 1 10 11 12 13 14 30

olemassa: 030405-060C

ei oteta käyttöön: 030405A060C

Tarkistusmerkin avulla voidaan tarkistaa, että mikään yksi merkki ei ole väärin

- havaitsee kaikki yhden merkin virheet
- havaittua virhettä ei voi automaattisesti korjata
- ei havaitse kaikkia kahden tai useamman merkin virheitä
- välimerkkiä ei tarkisteta lainkaan
  - '+' tarkoittaa 1800-lukua
  - '-' tarkoittaa 1900-lukua
  - 'A' tarkoittaa 2000-lukua

virheellinen: 120464A121C

OK

120464121 % 31 = 12 = 'C'

Copyright Teemu Kerola 2005

Henkilötunnuksen kehityshistorian vuoksi tarkistusmerkki ei ota huomioon välimerkkiä lainkaan. Tämän vuoksi mitään välimerkissä olevia virheitä ei havaita lainkaan, mikä aiheuttaa huomattavia rajoituksia välimerkin käyttämiseen ohjelmistoissa. Sitä ei itse asiassa saisi lainkaan hyödyntää, mistä myös seuraa, että hetussa olevaa 6-numeroista syntymäaikaakaan ei oikeastaan saisi käyttää syntymäaikatietona, koska sen vuosiluku on puutteellinen. Väestörekisterikeskuksessa välimerkkiä ei huomioida ja kaikki käytössä olevat henkilötunnukset eroavat toisistaan myös jonkun muun merkin kuin välimerkin osalta. Asian olisi voinut suunnitella paremminkin.

Bittitason tarkistukset

esim. hetu: 120464-121C

Muistipiirit, väylät, tiedonsiirrot, levyt

Monenko bitin muuttumisvirheet havaitaan?

Monenko bitin muuttumisvirheet voidaan automaattisesti korjata?

Montako ylimääräistä bittiä tarvitaan virheiden havaitsemiseen tai korjaamiseen?

- lisämuistitilan tai lisälevytilan tarve?
- lisäjohtimien tarve väylällä?

Tehdäänkö tarkistukset laitteisto vai ohjelmistotasolla?

- paljonko aikaa tarkistuksiin kuluu?

Copyright Teemu Kerola 2005

Bittitason tarkistuksia tehdään koko ajan tietoa käsiteltäessä laitteistossa. Kaikki suorittimen ulkoiset ja sisäiset rekisterit ja suorittimen sisäiset väylät on laitteistotasolla suojattu virheitä vastaan. Tavallisten pöytäkoneiden muistikin voi olla virheitä korjaavaa lajia, jolloin myös muistipiireihin sisältyy yhden virheen havaitsemis- ja korjauspiirit. Laitteiston väylät on yleensä suojattu, ainakin jollakin tavalla. Levytiedostotkin voi olla suojattu, mutta niitä varten on kehitetty ihan omat vähän monimutkaisemmat standardit, joista puhumme myöhemmin kurssilla.

Bittitason tarkistukset

esim. hetu: 120464-121C

Muistipiirit, väylät, tiedonsiirrot, levyt

Monenko bitin muuttumisvirheet havaitaan?

hetu: 1 bitti
(2 tai useampi, jos samassa merkissä)

Monenko bitin muuttumisvirheet voidaan automaattisesti korjata?

jos yhden virheen todennäköisyys: 1/1 000 000 eli 10^{-6}

niin kahden samanaikaisen virheen todennäköisyys: $(1/1\,000\,000)^2$ eli 10^{-12}

ja kolmen samanaikaisen virheen todennäköisyys: $(1/1\,000\,000)^3$ eli 10^{-18}

Montako ylimääräistä bittiä tarvitaan virhei[...] havaitsemiseen tai korjaamiseen?

- lisämuistitilan tai lisälevytilan tarve?
- lisäjohtimien tarve väylällä?

Tehdäänkö tarkistukset laitteisto vai ohjelmistotasolla?

- paljonko aikaa tarkistuksiin kuluu?

Copyright Teemu Kerola 2005

Bittitason virheen havaitsemis- ja korjausmenetelmiä voidaan luokitella muutaman olennaisen piirteen perusteella. Ehkä tärkein piirre on se, kuinka monen bitin muuttuminen vielä varmasti havaitaan. Mitä useamman bitin muuttuminen varmasti havaitaan, sitä parempi. Usein virheet ovat satunnaisia ja useamman virheen samanaikainen tapahtuminen on huomattavasti epätodennäköisempää kuin yhden virheen tapahtuminen. Tällä tavoin periaatteessa voidaan laskea riski sille, että virhe jää havaitsematta ja mahdollisesti aiheuttaa häiriön. Havaitsemattoman virheen todennäköisyys on siis hyväksytty ja harkittu riski.

Bittitason tarkistukset

esim. hetu: 120464-121C

Muistipiirit, väylät, tiedonsiirrot, levyt

Monenko bitin muuttumisvirheet havaitaan?

Monenko bitin muuttumisvirheet voidaan automaattisesti korjata?

hetu: 0 bittiä (virheitä ei voi koskaan korjata)

Montako ylimääräistä bittiä tarvitaan virheiden havaitsemiseen tai korjaamiseen?

- lisämuistitilan tai lisälevytilan tarve?
- lisäjohtimien tarve väylällä?

Tehdäänkö tarkistukset laitteisto vai ohjelmistotasolla?

- paljonko aikaa tarkistuksiin kuluu?

Copyright Teemu Kerola 2005

Toinen vastaava piirre virheen korjaavissa järjestelmissä on, että kuinka monen bitin virheet voidaan automaattisesti havaita ja korjata. Esimerkiksi voi olla, että 1 bitin virheet voidaan kaikki havaita ja korjata, kahden bitin virheet kaikki havaitaan mutta niitä ei voi korjata, mutta osa kolmen bitin virheistä jää jo havaitsematta. Riskianalyysissä sitten pätellään, kuinka todennäköinen kolmen bitin virhe on ja kuinka pahan häiriön se voisi mahdollisesti aiheuttaa.

Bittitason tarkistukset

esim. hetu: 120464-121C

Muistipiirit, väylät, tiedonsiirrot, levyt

Monenko bitin muuttumisvirheet havaitaan?

Monenko bitin muuttumisvirheet voidaan automaattisesti korjata?

Montako ylimääräistä bittiä tarvitaan virheiden havaitsemiseen tai korjaamiseen?

- lisämuistitilan tai lisälevytilan tarve?
- lisäjohtimien tarve väylällä?

hetu: 1 merkki eli 10% lisää
hetu: 11 merkkiä eli 88 b
tark.merkki: 8 b

Tehdäänkö tarkistukset laitteisto vai ohjelmistotasolla?

- paljonko aikaa tarkistuksiin kuluu?

Copyright Teemu Kerola 2005

Tilakustannus on merkittävä virheen havaitsemis/korjaus-järjestelmien piirre. Hetussa ylimääräinen tilakustannus on 10%, mikä ei ole ihan pieni määrä. Suorittimen sisäisen väylän suojaamisessa virheitä korjaavilla biteillä tarvitaan usein vielä enemmän ylimääräisiä bittejä. Toisaalta, tätä ei pidä ajatella kovin negatiivisesti. Suorittimen toiminnassa on kaikkialla muuallakin tärkeimpänä tavoitteena, että se ei tee minkäänlaisia virheitä. Virheiden torjunta ei siis ole mitenkään ylimääräistä vaan olennainen osa normaalia toimintaa.

Bittitason tarkistukset

esim. hetu: 120464-121C

Muistipiirit, väylät, tiedonsiirrot, levyt

Monenko bitin muuttumisvirheet havaitaan?

Monenko bitin muuttumisvirheet voidaan automaattisesti korjata?

Montako ylimääräistä bittiä tarvitaan virheiden havaitsemiseen tai korjaamiseen?

- lisämuistitilan tai lisälevytilan tarve?
- lisäjohtimien tarve väylällä?

Tehdäänkö tarkistukset laitteisto vai ohjelmistotasolla?

- paljonko aikaa tarkistuksiin kuluu?

hetu: ohjelmistotasolla

Copyright Teemu Kerola 2005

Useat virheen havaitsemis- ja korjausalgoritmit voidaan yhtä hyvin toteuttaa laitteistolla tai ohjelmistolla. Kumpaa tapaa käytetään, riippuu muista parametreista kuten algoritmin nopeus- ja muokattavuusvaatimuksista. Esimerkiksi hetun tarkistaminen tehdään kuitenkin aina ohjelmistotasolla, koska (a) se on aika lailla vain suomalainen piirre ja (b) hetua käsiteltäessä sen tarkistukseen kuluu oikeastaan vain hyvin vähän aikaa muuhun käsittelyaikaan verrattuna. Toisaalta, esimerkiksi muistiväylän virheenkorjaus on pakko nopeusvaatimusten vuoksi tehdä laitteistolla.

Pariteettibitti

1001110?

Yksi ylimääräinen bitti per tietoalkio
- tavu, sana, tietoliikennepaketti?

10011101 10001111 00110111 00011100
01010001 11100011 00001111 00101010
????????

Parillinen (tai pariton) pariteetti: 1-bittien lukumäärä <u>pariteettibitti mukaanlukien</u> on aina parillinen (tai pariton)

Havaitsee 1 bitin muuttumisen

Ei voi korjata koskaan

Esimerkkejä (parillinen pariteetti)

0 0 1 0 0 0 1 ?

1 0 0 0 1 1 0 1 1 1 1 1 0 0 1 ?

Copyright Teemu Kerola 2005

Yksinkertaisin tiedontarkistusmenetelmä on käyttää pariteettibittiä. Siinä jokaiselle tietoalkiolle varataan ylimääräinen bitti tiedon muuttumattomuustarkistusta varten. Jos tietoalkiossa on 7-bittiä, niin pariteettibitin kanssa alkion pituus on sitten 8 bittiä. Toisaalta, jos 8-bittistä merkkiä halutaan suojata pariteettibitillä, niin tietoalkion pituudeksi tulee inhoittavat 9 bittiä. Tällöin usein pariteettibitit sijoitetaankin omaan tavuunsa siten, että esimerkiksi joka 9. tavu sisältää pariteettibitin edeltäville 8 tavulle. Tiedon tarkistus on tietenkin tällöin vähän epätriviaalia.

Pariteettibitti

Yksi ylimääräinen bitti per tietoalkio

- tavu, sana, tietoliikennepaketti?

Parillinen (tai pariton) pariteetti: 1-bittien lukumäärä pariteettibitti mukaanlukien on aina parillinen (tai pariton)

1101 1101

1101 1000

Havaitsee 1 bitin muuttumisen

Ei voi korjata koskaan

Esimerkkejä (parillinen pariteetti)

0 0 1 0 0 0 1 ?

1 0 0 0 1 1 0 1 1 1 1 1 0 0 1 ?

Copyright Teemu Kerola 2005

Pariteettibitti lasketaan data-biteistä sillä tavoin, että 1-bittien kokonaislukumäärä on parillista pariteettia käytettäessä parillinen. Esimerkiksi Intelin arkkitehtuureissa on tilarekisterissä erityinen pariteettibitti, joka ilmaisee operaation tuloksen pariteetin jokaisen operaation jälkeen. Pariteettibitin arvo lasketaan ja kirjoitetaan joka kerta uuden arvon yhteydessä ja pariteettibitin oikeellisuus lasketaan joka kerta kyseistä tietoa käytettäessä.

Pariteettibitti

Yksi ylimääräinen bitti per tietoalkio

- tavu, sana, tietoliikennepaketti?

Parillinen (tai pariton) pariteetti: 1-bittien lukumäärä pariteettibitti mukaanlukien on aina parillinen (tai pariton)

alkuper. oikea data
1101 1101

Havaitsee 1 bitin muuttumisen

1101 1001

Ei voi korjata koskaan

1101 1100

parillinen pariteetti,
1-bittejä 5 kpl
⇨ VIRHE JOSSAKIN

Esimerkkejä (parillinen pariteetti)

0 0 1 0 0 0 1 ?

1 0 0 0 1 1 0 1 1 1 1 1 0 0 1 ?

Copyright Teemu Kerola 2005

Pariteettibitin avulla havaitaan minkä tahansa yhden bitin muuttuminen. Sillä ei havaita mitään kahden bitin muutoksia, koska ne kumoavat toisensa. Kuten muissakin tiedon muuttumattomuuden valvontamenetelmissä, pariteetibitti siis talletetaan aivan samalla tavalla kuin varsinaiset databititkin. On myös mahdollista, että pariteettibitti itse on muuttunut ja kaikki databitit ovat ennallaan, jolloin virheellinen pariteettibitti itse asiassa antaa tavallaan väärän hälyytyksen, koska kaikki databitit ovat edelleen oikein. Virheellisen pariteetin tapauksessa meillä ei ole mitään tietoa virheen sijainnista, joten virheitä ei voi korjata.

Pariteettibitti

Yksi ylimääräinen bitti per tietoalkio

- tavu, sana, tietoliikennepaketti?

Parillinen (tai pariton) pariteetti: 1-bittien lukumäärä <u>pariteettibitti mukaanlukien</u> on aina parillinen (tai pariton)

Havaitsee 1 bitin muuttumisen

Ei voi korjata koskaan

Esimerkkejä (parillinen pariteetti)

0010 001?

1000 1101 1111 001?

Copyright Teemu Kerola 2005

Harjoitellaanpa vähän. Mikä pariteettibitin arvo suojaa näitä 7 data-bittiä?

Pariteettibitti

Yksi ylimääräinen bitti per tietoalkio

- tavu, sana, tietoliikennepaketti?

Parillinen (tai pariton) pariteetti: 1-bittien lukumäärä pariteettibitti mukaanlukien on aina parillinen (tai pariton)

Havaitsee 1 bitin muuttumisen

Ei voi korjata koskaan

Esimerkkejä (parillinen pariteetti)

0 0 1 0 0 0 1 **0**

1 0 0 0 1 1 0 1 1 1 1 1 0 0 1 ?

2 kpl 1-bittejä, siis parit. bitti on 0

Copyright Teemu Kerola 2005

Käytössä on parillinen pariteetti ja ykkösbittejä on databiteissä 2 kappaletta. Pariteettibitin pitää siis olla nolla, jotta kaikkien bittien 1-bittien lukumäärä pysyy parillisena.

Pariteettibitti

Yksi ylimääräinen bitti per tietoalkio

- tavu, sana, tietoliikennepaketti?

Parillinen (tai pariton) pariteetti: 1-bittien lukumäärä <u>pariteettibitti mukaanlukien</u> on aina parillinen (tai pariton)

Havaitsee 1 bitin muuttumisen

Ei voi korjata koskaan

Esimerkkejä (parillinen pariteetti)

0 0 1 0 0 0 1 ?

1 0 0 0 1 1 0 1 1 1 1 1 0 0 1 ?

Copyright Teemu Kerola 2005

No entäpä mikä olisi pariteettibitin arvo suojaamaan näitä 15 data-bittiä?

Pariteettibitti

Yksi ylimääräinen bitti per tietoalkio

- tavu, sana, tietoliikennepaketti?

Parillinen (tai pariton) pariteetti: 1-bittien lukumäärä pariteettibitti mukaanlukien on aina parillinen (tai pariton)

Havaitsee 1 bitin muuttumisen

Ei voi korjata koskaan

Esimerkkejä (parillinen pariteetti)

0 0 1 0 0 0 1 ?

1 0 0 0 1 1 0 1 1 1 1 1 0 0 1 **1**

9 kpl 1-bittejä, siis parit. bitti on 1

Copyright Teemu Kerola 2005

1-bittejä on databiteissä 9 kappaletta, joten parilliseen pariteettiin päästäksemme pariteettibitin arvon tulee olla 1.

Hamming-etäisyys

Miten monta bittiä missä tahansa koodijärjestelmässä (esim. ISO Latin-9) esitetyssä koodissa täytyy muuttua, että se muuttuu johonkin toiseen (mihin tahansa) lailliseen koodiin?

ISO Latin-9'n Hamming-etäisyys on 1

Pariteettibitillä vahvistettuna ISO Latin-9'n Hamming-etäisyys on 2

- mikä todennäköisyys kahden bitin virheeseen?
- riittävän pieni?
- miten pariteettibitti sijoitetaaan?
- joka 9. tavu on parit. tavu?

Prob {2 bitin virhe} = (Prob {yhden bitin virhe}) ^{2}

Prob {3 bitin virhe} = (Prob {yhden bitin virhe}) ^{3}

Copyright Teemu Kerola 2005

Tiedon muuttumattomuuden käsittelyssä Hamming-etäisyys on tärkeä. Se kertoo, kuinka monta bittiä pitää vähintään muuttua, jotta laillinen tieto voisi muuttua toiseksi lailliseksi tiedoksi. Jos bittejä muuttuu esimerkiksi virheen takia vähemmän kuin Hamming-etäisyyden osoittama määrä, niin tuloksena on aina jokin laiton esitystapa, jolloin virhe varmasti havaitaan. Jos taas virheellisiä muutoksia on Hamming-etäisyyden osoittama määrä, niin on varsin mahdollista, että uusi bittiyhdistelmä on jokin laillinen merkki ja virhe jää havaitsematta.

Hamming-etäisyys

Miten monta bittiä missä tahansa koodijärjestelmässä (esim. ISO Latin-9) esitetyssä koodissa täytyy muuttua, että se muuttuu johonkin toiseen (mihin tahansa) lailliseen koodiin?

ISO Latin-9'n Hamming-etäisyys on 1

'A' = 0x41 = 0100 0001

kahden bitin ero

'B' = 0x42 = 0100 0010

yhden bitin ero

'C' = 0x43 = 0100 0011

Pariteettibitillä vahvistettuna ISO Latin-9'n Hamming-etäisyys on 2

- mikä todennäköisyys kahden bitin virheeseen?
- riittävän pieni?
- miten pariteettibitti sijoitetaan?
- joka 9. tavu on parit. tavu?

Prob {2 bitin virhe} = (Prob {yhden bitin virhe}) ^{2}

Prob {3 bitin virhe} = (Prob {yhden bitin virhe}) ^{3}

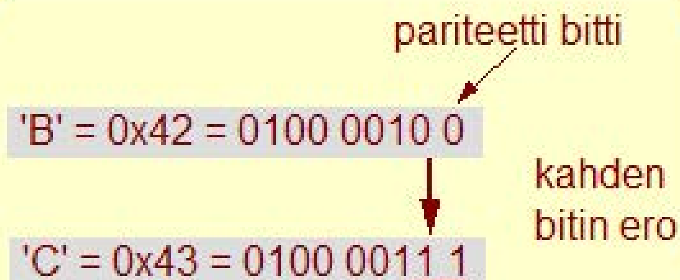
Copyright Teemu Kerola 2005

Esimerkiksi 8-bittisessä ISO Latin-9 koodistossa A- ja B-kirjainten esitysmuodoissa on kahden bitin ero, kun taas B- ja C-kirjainten välillä on vain yhden bitin ero. Pienimmillään siis Latin-9 koodistoa käyttäen vain yhden bitin muuttuminen voi muuttaa laillisen merkin (esim. 'B') toiseksi lailliseksi merkiksi (esim. 'C'). Koodiston Hamming-etäisyys on siten 1.

Hamming-etäisyys

Miten monta bittiä missä tahansa koodijärjestelmässä (esim. ISO Latin-9) esitetyssä koodissa täytyy muuttua, että se muuttuu johonkin toiseen (mihin tahansa) lailliseen koodiin?

ISO Latin-9'n Hamming-etäisyys on 1



Pariteettibitillä vahvistettuna ISO Latin-9'n Hamming-etäisyys on 2

- mikä todennäköisyys kahden bitin virheeseen?
- riittävän pieni?
- miten pariteettibitti sijoitetaaan?
- joka 9. tavu on parit. tavu?

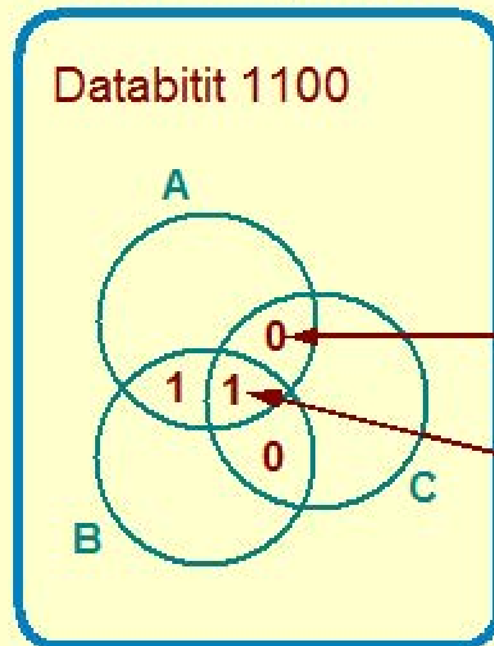
Prob {2 bitin virhe} = (Prob {yhden bitin virhe}) ^{2}

Prob {3 bitin virhe} = (Prob {yhden bitin virhe}) ^{3}

Copyright Teemu Kerola 2005

Jos jokaiseen merkkiin lisätään pariteettibitti, niin Hamming-etäisyydeksi tulee 2. Nyt jokainen 1 bitin virhe aiheuttaa virheellisen pariteetin, jolloin kyseinen 9-bitin yhdistelmä ei vastaa mitään laillista Latin-9 merkkiä. Jos yhden bitin muuttumisen todennäköisyys on vaikkapa 1 miljoonasta, niin sen voidaan ajatella olevan liian suuri riski. Hamming-etäisyyden ollessa kaksi havaitsemattoman virheen todennäköisyys on vain 1 biljoonasta, joka sitten voikin olla jo tarpeeksi pieni kyseiseen sovellukseen. Jos tämäkään ei riitä, käytetään koodistoa, jossa Hamming-etäisyys on 3, jolloin havaitsemattoman virheen todennäköisyys on jo 1 triljoonasta.

Virheen korjaava Hamming-koodin idea



Jokainen databitti kuuluu erilaiseen kokelmaan pariteettijoukkoja

Tämä 0-bitti kuuluu joukkoihin A ja C, eikä mikään muu databitti kuulu täsmälleen joukkoihin A ja C

Tämä 1-bitti kuuluu kaikkiin joukkoihin A, B ja C, eikä mikään muu databitti kuulu kaikkiin joukkoihin A, B ja C

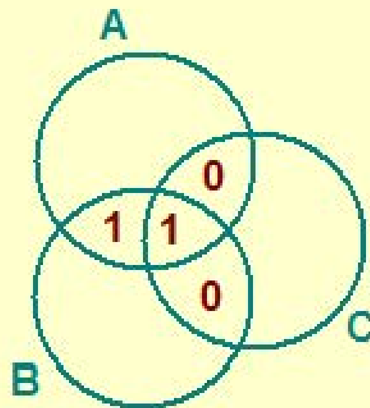
[Tane99, Fig. 2-14]

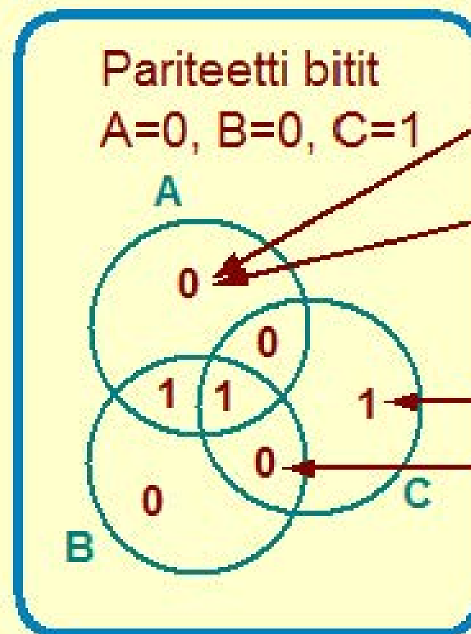
Copyright Teemu Kerola 2005

Virheen korjaavalla Hamming-koodilla ei ole mitään tekemistä Hamming-etäisyyden kanssa, paitsi tietenkin idean isä. Tarkastellaan nyt tilannetta, jossa meillä on vain 4 databittiä. Databitit sijoitetaan pariteettiluokkiin A, B ja C sillä tavoin, että jokainen databitti kuuluu uniikkiin kokoelmaan pariteettiluokkia.

Virheen korjaava Hamming-koodin idea

Databitit 1100





Pariteettiluokan A pariteettibitti

Tämä bitti kuuluu pariteettijoukkoon A, mutta ei joukkoon B tai C

Pariteettiluokan C pariteettibitti

Tämä bitti kuuluu pariteettijoukkoihin B ja C, mutta ei joukkoon A

Tarvitaan 3 kpl pariteettibittejä, yksi kullekin pariteettiluokalle (parillinen pariteetti)

Jokainen data- ja pariteettibitti kuuluu uniikkiin kokoelmaan pariteettijoukkoja

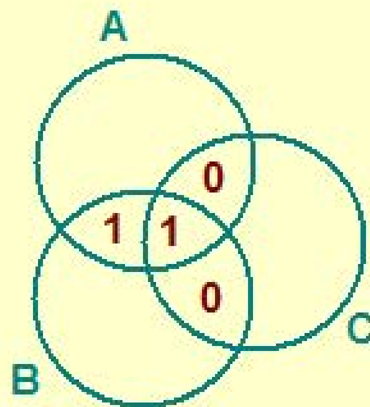
[Tane99, Fig. 2-14]

Copyright Teemu Kerola 2005

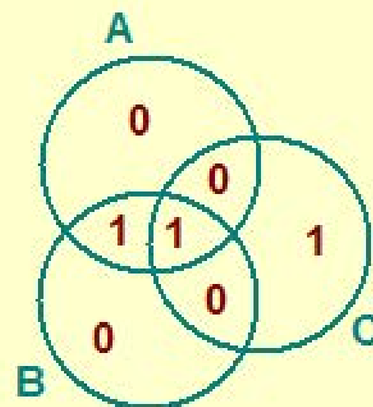
Jokaiseen pariteettiluokkaan sisältyy yksi pariteettibitti, joka asetetaan parilliseen pariteettiin. Bittejä on nyt yhteensä 7 kpl, jotka voidaan kukin identifioida noilla kolmella pariteettibitillä. Kolmella bitillähän voidaan esittää 8 eri tilaa, joten systeemissä olisi vielä tilaa yhdelle databitille lisää. Toisaalta, kaksi pariteettiluokkaa ei riittäisi, koska kahdella pariteettibitillä ei pysty identifiomaan kuutta eri bittiä. Tyydytään nyt kuitenkin näihin neljään databittiin. Olennaista on siis kuitenkin se, että jokainen 7'stä bitistä kuuluu erilaiseen kokoelmaan pariteettiluokkia.

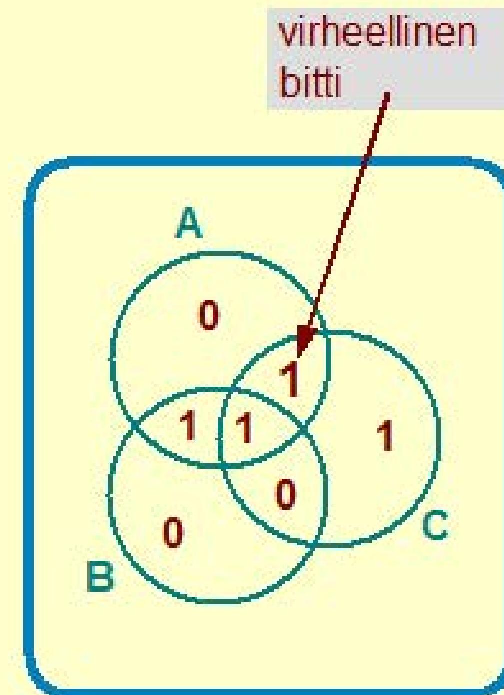
Virheen korjaava Hamming-koodin idea

Databitit 1100



Pariteetti bitit
A=0, B=0, C=1





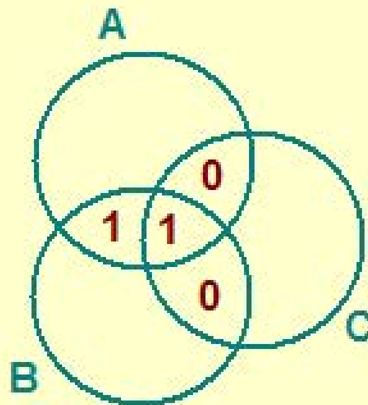
[Tane99, Fig. 2-14]

Copyright Teemu Kerola 2005

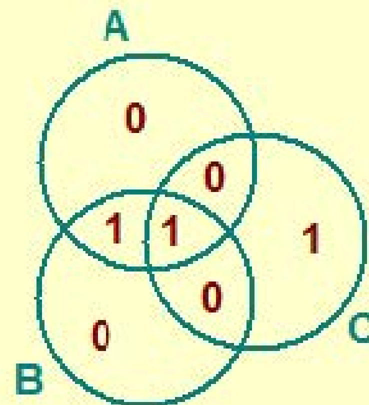
Tarkastellaan nyt tilannetta, jossa tapahtunut virhe korjataan. Oletetaan, että tietty 0-bitti on syystä tai toisesta muuttunut ykköseksi. Tässä esimerkissä se sattuu olemaan databitti, mutta virheen korjaava algoritmi toimii ihan yhtä hyvin myös pariteettibitin kohdalla.

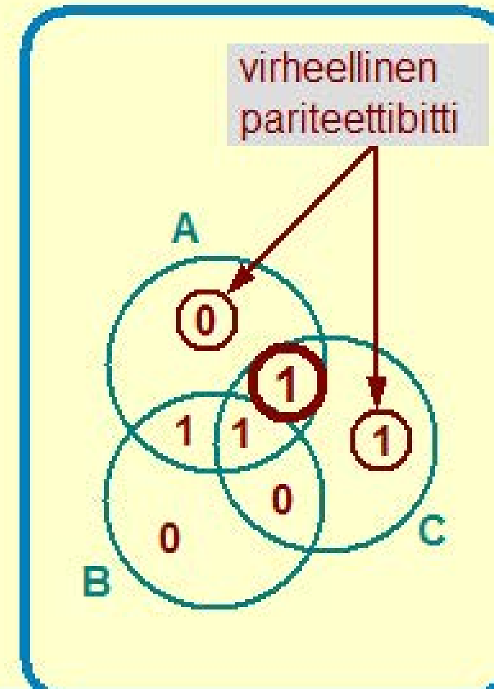
Virheen korjaava Hamming-koodin idea

Databitit 1100



Pariteetti bitit
A=0, B=0, C=1





Pariteettiluokkien A ja C pariteettibitit havaitsevat virheen. Vain yksi bitti (data-biteistä ja pariteettibiteistä) on sellainen, että se kuuluu täsmälleen pariteettiluokkiin A ja C. Virheellinen bitti identifioidaan.

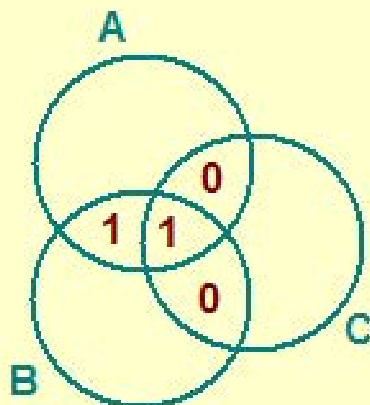
[Tane99, Fig. 2-14]

Copyright Teemu Kerola 2005

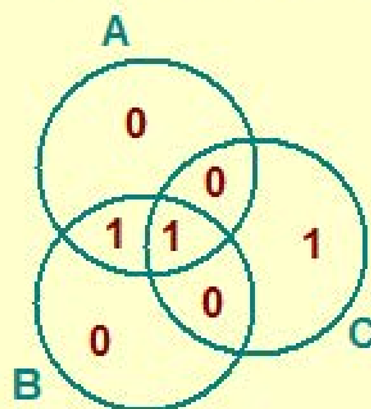
Dataa seuraavan kerran käsiteltäessä pariteettiluokkien A ja C pariteettibitit ovat virheellisiä ja nostavat asiasta asiaankuuluvan 'metakan'. Virheenkorjausalgoritmi tutkii tilannetta ja havaitsee, että ainoastaan yksi bitti seitsemästä on sellainen, joka kuuluu pariteettiluokkiin A ja C, mutta ei B'hen, joten siis tuon bitin täytyy olla virheellinen.

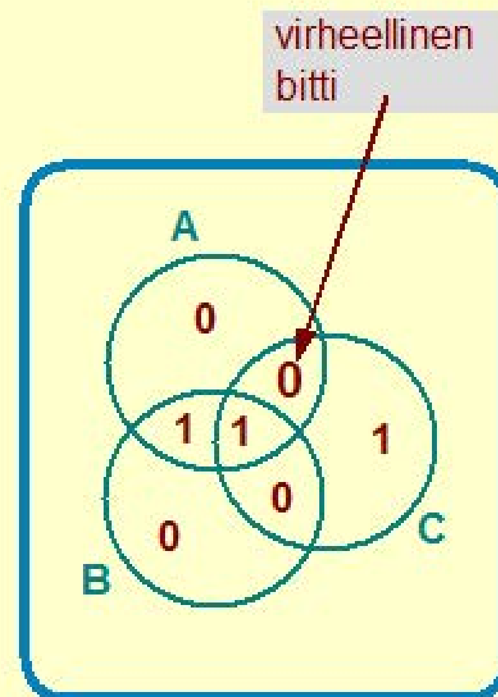
Virheen korjaava Hamming-koodin idea

Databitit 1100



Pariteetti bitit
A=0, B=0, C=1





Virheellinen bitti korjataan, eli "flipataan"

[Tane99, Fig. 2-14]

Copyright Teemu Kerola 2005

Virheenkorjaaminen ei tietenkään tarkoita vain sitä, että virheellisen bitin sijainti löydetään. Virheellinen tieto pitää myös korjata. Onneksi meillä on käytössä binäärijärjestelmä, jolloin virheellisen bitin ollessa kyseessä vaihtoehtoisia oikeita arvo on jäljellä vain 1 kpl. Oikea arvo saadaan 'flippaamalla' virheellinen bitti takaisin oikein päin eli muuttamalla nolla ykköseksi ja päin vastoin.

Hamming-koodi

Käytetään montaa pariteettibittiä, yksi kussakin pariteettiluokassa

4 bittiä dataa
3 pariteettiluokkaa
3 pariteettibittiä

Data ja parit. bitit: **100 1100**
Bitti nro: 765 **4**3**21**

Havaitaan kahden bitin muuttuminen
Korjataan 1 bitin muuttuminen

Kaikki bitit nro 2^i ovat pariteettibittejä, muut bitit ovat databittejä

Pariteettiluokan muodostavat ne bitit, joiden järjestysnumerossa binääriesityksessä on mukana kyseinen pariteettibitti

Kutakin databittiä tarkistavat ne pariteettibitit, jotka ovat mukana kyseisen databitin järjestysnumeron binääriesityksessä

Copyright Teemu Kerola 2005

Hamming-koodi perustuu edellisellä sivulla kuvattuun ideaan, mutta ei ole ihan samanlainen. Esimerkissämme on vain 7 bittiä, mutta menetelmä toimii vastaavasti mille tahansa bittien lukumäärälle. Jos käytettävissä on yhteensä 7 bittiä, niin siitä ainoastaan 4 bittiä on dataa ja loput 3 bittiä pariteettibittejä. Databitit jaetaan kolmeen pariteettiluokkaan, jotka tässä esimerkissä sattuvat olemaan kaikki saman kokoisia, mutta yleisesti ottaen eivät ole. Edelleenkin pätee, että jokainen bitti (esimerkissä siis 7 bittiä) kuuluu uniikkiin kokoelmaan pariteettijoukkoja.

Hamming-koodi

Käytetään montaa pariteettibittiä, yksi kussakin pariteettiluokassa

4 bittiä dataa
3 pariteettiluokkaa
3 pariteettibittiä

Data ja parit. bitit: **100 1100**

Bitti nro: 765 **4**3**21**

Havaitaan kahden bitin muuttuminen
Korjataan 1 bitin muuttuminen

Kaikki bitit nro 2^i ovat pariteettibittejä, muut bitit ovat databittejä

Pariteettiluokan muodostavat ne bitit, joiden järjestysnumerossa binääriesityksessä on mukana kyseinen pariteettibitti

Kutakin databittiä tarkistavat ne pariteettibitit, jotka ovat mukana kyseisen databitin järjestysnumeron binääriesityksessä

Copyright Teemu Kerola 2005

Järjestelmä toimii periaatteessa kuten edellisellä sivulla esitettiin ja kaikki 1 bitin virheet voidaan korjata sen avulla automaattisesti. Järjestelmä myös havaitsee kaikki kahden bitin virheet, mutta ei pysty identifioimaan niitä virheen korjaamiseksi. Kaikkia kolmen bitin virheitä ei pystytä havaitsemaan.

Hamming-koodi

Käytetään montaa pariteettibittiä, yksi kussakin pariteettiluokassa

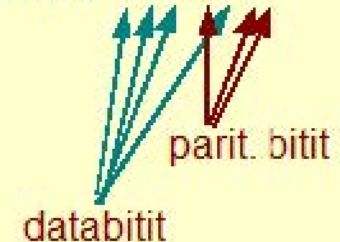
4 bittiä dataa
3 pariteettiluokkaa
3 pariteettibittiä

Havaitaan kahden bitin muuttuminen
Korjataan 1 bitin muuttuminen

Kaikki bitit nro 2^i ovat pariteettibittejä, muut bitit ovat databittejä

Data ja parit. bitit: **100 1100**

Bitti nro: 765 **4**3**21**



Pariteettiluokan muodostavat ne bitit, joiden järjestysnumerossa binääriesityksessä on mukana kyseinen pariteettibitti

Kutakin databittiä tarkistavat ne pariteettibitit, jotka ovat mukana kyseisen databitin järjestysnumeron binääriesityksessä

Copyright Teemu Kerola 2005

Hamming-koodin hieno idea perustuu bittien numerointiin, alkaen numerosta 1 toisin kuin yleensä tehdään. Yleensähän bitit numeroidaan nollasta alkaen. Numeroinnin voi tehdä vasemmalta oikealle tai oikealta vasemmalle, sillä ei ole väliä. Tässä numerointi on nyt tehty oikealta vasemmalle. Kantava idea on, että kaikki bitit, joiden numero on jokin kakkosen potenssi, ovat pariteettibittejä ja loput databittejä. Tämän idean perusteella voi sitten aina päätellä, kuinka monta pariteettibittiä tiettyyn määrään databittejä tarvitaan.

Hamming-koodi

Käytetään montaa pariteettibittiä, yksi kussakin pariteettiluokassa

4 bittiä dataa
3 pariteettiluokkaa
3 pariteettibittiä

Data ja parit. bitit: **100 1100**

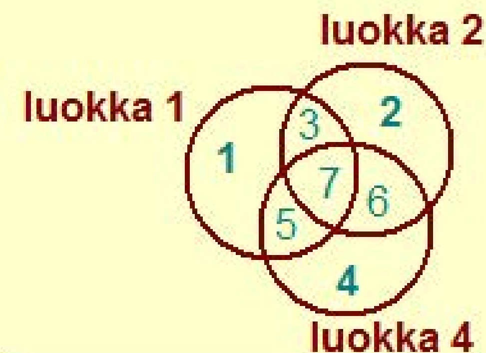
Bitti nro: 765 **4**3**21**

Havaitaan kahden bitin muuttuminen
Korjataan 1 bitin muuttuminen

luokka 1	luokka 2	luokka 4
1 = 001	2 = 010	4 = 100
3 = 011	3 = 011	5 = 101
5 = 101	6 = 110	6 = 110
7 = 111	7 = 111	7 = 111

Kaikki bitit nro 2^i ovat pariteettibittejä, muut bitit ovat databittejä

Pariteettiluokan muodostavat ne bitit, joiden järjestysnumerossa binääriesityksessä on mukana kyseinen pariteettibitti



Kutakin databittiä tarkistavat ne pariteettibitit, jotka ovat mukana kyseisen databitin järjestysnumeron binääriesityksessä

Copyright Teemu Kerola 2005

Paritettiluokat perustuvat bittien järjestysnumeroon ja erityisesti tuon järjestysnumeron binääriesitykseen. Kaikki numerothan voidaan esittää uniikisti binäärilukuna ja pariteettiluokat määritellään tämän mukaisesti. Kuhunkin pariteettiluokkaan kuuluvat ne bitit, joiden järjestysnumeron binääriesityksessä on mukana kyseinen pariteettibitti. Esimerkiksi pariteettiluokkaan 2 eli pariteettibitin 2 valvomaan ryhmään kuuluvat bitit 2, 3, 6 ja 7. Esimerkiksi, jos luku 6 esitetään binääriesityksessä eli kakkosen potenssien summana, niin mukana on bitti 2 eli kakkonen potenssiin 1.

Hamming-koodi

Käytetään montaa pariteettibittiä, yksi kussakin pariteettiluokassa

4 bittiä dataa
3 pariteettiluokkaa
3 pariteettibittiä

Havaitaan kahden bitin muuttuminen
Korjataan 1 bitin muuttuminen

Kaikki bitit nro 2^i ovat pariteettibittejä, muut bitit ovat databittejä

Pariteettiluokan muodostavat ne bitit, joiden järjestysnumerossa binääriesityksessä on mukana kyseinen pariteettibitti

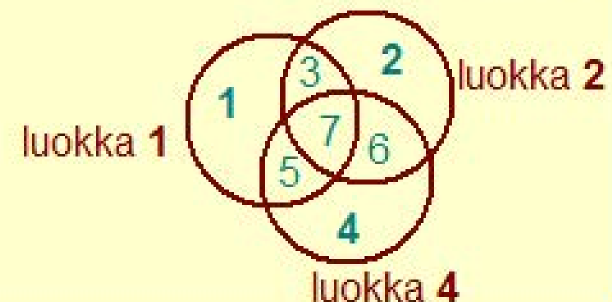
Kutakin databittiä tarkistavat ne pariteettibitit, jotka ovat mukana kyseisen databitin järjestysnumeron binääriesityksessä

Data ja parit. bitit: **100 1100**
Bitti nro: 765 **4321**

luokka **421**
1 = 001
2 = 010
3 = 011
4 = 100
5 = 101
6 = 110
7 = 111

bittiä 3 tarkistaa parit.bitit 2 ja 1

bittiä 6 tarkistaa parit.bitit 4 ja 2



Copyright Teemu Kerola 2005

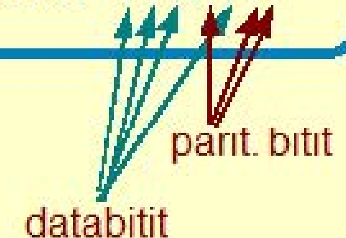
Samalla tavalla kunkin bitin binääriesityksestä voidaan päätellä, mihin kaikkiin pariteettiluokkiin kyseinen bitti kuuluu. Bitti kuuluu niihin pariteettiluokkiin, joiden kohdalla on ykkönen kyseisen bitin järjestysnumeron binääriesityksessä. Esimerkiksi bitin 6 järjestysnumeron binääriesitys on 110, joten bittiä 6 valvoo pariteettiluokat ja siis pariteettibitit 4 ja 2. Vastaavasti bittiä 3 valvoo bitit 2 ja 1. Pariteettibittiä 1 valvoo ainoastaan se itse eli bitti 1. Jokaista bittiä valvoo kuitenkin vähintään 1 bitti ja jokaista bittiä valvoo uniikki kokoelma pariteettibittejä.

Esimerkki: Virheen korjaava Hamming-koodi

Alkuperäinen data

originaali data ja parit. bitit: 100 **?1??**

bitti nro: 765 **4**3**21**



Pariteettibitit

Virheellinen data

Virheen havaitseminen

Virheellisen bitin paikallistaminen

Virheen korjaus

Copyright Teemu Kerola 2005

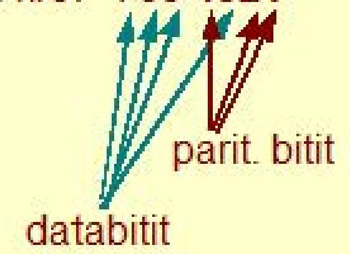
Tarkastellaan Hamming-koodia vielä yksinkertaisen esimerkin avulla. Oletetaan, että meillä on 4-bittinen tieto, joka talletetaan Hamming-koodilla suojattuun 7-bittiseen kenttään. Varsinainen data voisi olla vaikkapa etumerkitön pieni kokonaisluku 9 tai mikä tahansa muu 4-bittinen tieto. Joka tapauksessa sen bittiarvot ovat 1001.

Esimerkki: Virheen korjaava Hamming-koodi

pariteettiluokka 1: 1 0 1 ?

originaali data ja parit. bitit: 100 1100

bitti nro: 765 4321



Alkuperäinen data

Pariteettibitit

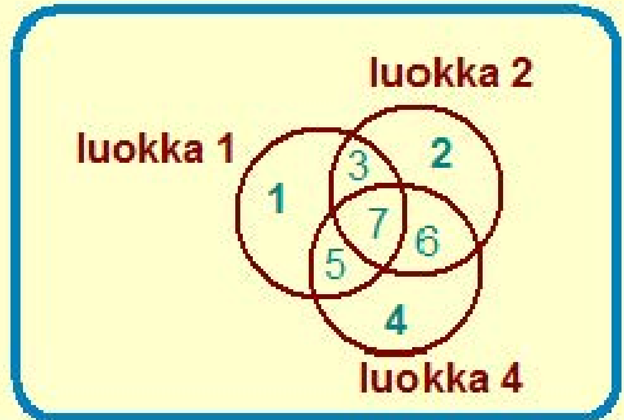
Luokka 1
bittien 7, 5, 3 arvot 1, 0, 1
parillinen pariteetti
⇨ parit. bitti 1:n arvo on 0

Virheellinen data

Virheen havaitseminen

Virheellisen bitin paikallistaminen

Virheen korjaus



Copyright Teemu Kerola 2005

Tietoa talletettaessa sen bitit luokitellaan automaattisesti Hamming-koodin mukaisiin pariteettiluokkiin ja kunkin luokan pariteettibiteille asetetaan parillisen pariteetin mukaiset arvot. Tästä lähtien tuo neljän bitin tieto talletetaan ja liikkuu järjestelmässä 7-bittisenä datana.

Esimerkki: Virheen korjaava Hamming-koodi

Alkuperäinen data

originaali data ja parit. bitit: 100 **1100**
bitti nro: 765 **4**3**21**

Pariteettibitit

virhe

virheellinen data ja parit. bitit: 110 **1100**
Bitti nro: 765 **4**3**21**

Virheellinen data

Virheen havaitseminen

Virheellisen bitin paikallistaminen

Virheen korjaus

Copyright Teemu Kerola 2005

Oletetaan nyt edelleen, että jonkin staattisen tai transientin vian vuoksi 7-bittisessä kentässä 2. bitti vasemmalta eli Hamming-koodissa käytetyn bittinumeroinnin mukaisesti bitti 6 muuttuu virheelliseen arvoon 1.

Esimerkki: Virheen korjaava Hamming-koodi

Alkuperäinen data

originaali data ja parit. bitit: 100 **1100**

bitti nro: 765 **4**3**21**

Pariteettibitit

Virheellinen data

Virheen havaitseminen

Virheellisen bitin paikallistaminen

Virheen korjaus

virhe

virheellinen data ja parit. bitit: 110 **1100**

Bitti nro: 765 **4**3**21**

Virheellinen parillinen pariteettibitti

Copyright Teemu Kerola 2005

Kun tätä tietoa seuraavan kerran käsitellään, esimerkiksi rekisteriin talletettaessa tai muistipiiristä luettaessa, pariteettibitit tarkistetaan. Jos yksikin pariteettibitti on väärin, niin tiedossa on siis jokin virhe. Tästä raportoidaan ehkä jollain tavoin, minkä jälkeen virhettä yritetään paikallistaa olemassaolevan tiedon eli virheellisten pariteettibittien perusteella. Jos virheitä on vain yksi, niin virhe voidaan aina paikallistaa.

Esimerkki: Virheen korjaava Hamming-koodi

Alkuperäinen data

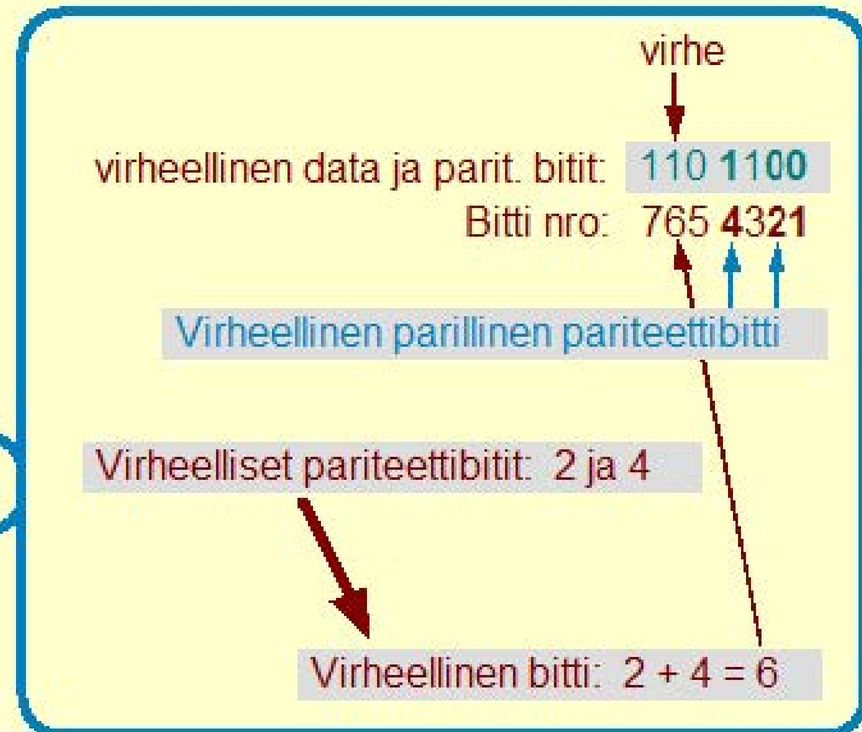
Pariteettibitit

Virheellinen data

Virheen havaitseminen

Virheellisen bitin paikallistaminen

Virheen korjaus



Copyright Teemu Kerola 2005

Pariteettibitit 2 ja 4 havaitsivat virheen. Toisaalta bitti 6 on ainoa bitti, jota nimenomaan pariteettibitit 2 ja 4 (mutta ei pariteettibitti 1) tarkastavat. Täten siis bitin 6 täytyy olla virheellinen. Tällainen päättely onnistuu ainoastaan siinä tapauksessa, että virheellisiä bittejä oli 1 kappale. Jos virheitä olisi useampi, niin voi olla, että virhe jää kokonaan havaitsematta, tai sitten virheellisen bitin sijainti päätellään väärin.

Esimerkki: Virheen korjaava Hamming-koodi

Alkuperäinen data

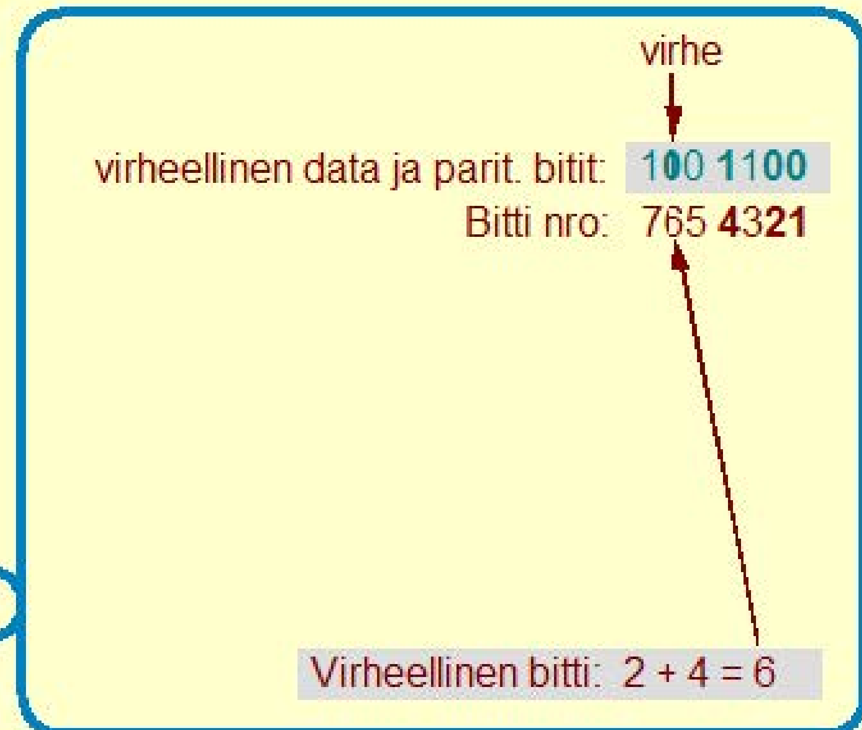
Pariteettibitit

Virheellinen data

Virheen havaitseminen

Virheellisen bitin paikallistaminen

Virheen korjaus



Copyright Teemu Kerola 2005

Totta kai virheellisen bitin paikannuksen jälkeen se pitää automaattisesti myös korjata. Korjaaminen on helppoa, koska binäärijärjestelmässä bitin virheellisestä arvosta voidaan helposti päätellä sen oikea arvo, muuttamalla 0 ykköseksi ja päin vastoin. Tässä tapauksessa siis virheellinen 1-arvo muutetaan nollaksi.

CRC - Cyclic Redundancy Code

Tiedonsiirrossa verkon yli käytetty tarkistusmenetelmä

- jos virheitä tulee, niitä tulee usein paljon
- pariteettibitit eivät riitä, virheitä ei korjata

Tarkistussumma (16-bittinen CRC) isolle tietojoukolle

- laske CRC = f(viesti) % 2^{16} (eli ota 16 viimeistä bittiä)
- lähetä viesti ja CRC (2 ylimääräistä tavua)
- vastaanota viesti ja CRC
- laske CRC ja tarkista onko sama kuin viestin mukana oleva
- jos väärin, niin pyydä uudelleenlähetystä

CRC-CCITT CRC's detect:	**All single-bit and double-bit errors**
	All errors of an odd number of bits
	All error bursts of 16 bits or less
	In summary, 99.998% of all errors

Copyright Teemu Kerola 2005

Tietoliikenneverkossa Hamming-koodin perusoletus korkeintaan yhdestä virheellisestä bitistä ei päde. Tietoliikenneyhteyksissä on nimenomaan havaittu, että bittivirheet eivät ole toisistaan riippumattomia. Eli jos jos yksikin bitti on virheellinen, niin on aika todennäköistä, että virheitä on useampikin. Tällaisessa tilanteessa käytetään Hamming koodin asemesta tarkistussummia, joilla pyritään havaitsemaan mahdollisimman usea virhe.

CRC - Cyclic Redundancy Code

Tiedonsiirrossa verkon yli käytetty tarkistusmenetelmä

- jos virheitä tulee, niitä tulee usein paljon
- pariteettibitit eivät riitä, virheitä ei korjata

Tarkistussumma (16-bittinen CRC) isolle tietojoukolle

- laske CRC = f(viesti) % 2^{16} (eli ota 16 viimeistä bittiä)
- lähetä viesti ja CRC (2 ylimääräistä tavua)
- vastaanota viesti ja CRC
- laske CRC ja tarkista onko sama kuin viestin mukana oleva
- jos väärin, niin pyydä uudelleenlähetystä

CRC-CCITT CRC's detect:
All single-bit and double-bit errors
All errors of an odd number of bits
All error bursts of 16 bits or less
In summary, 99.998% of all errors

Copyright Teemu Kerola 2005

Tarkistussumma lasketaan yleisesti jonkin sopivan funktion avulla kaikista tiedonsiirtopaketin biteistä. Itse funktion määrittely optimaaliseksi on tietenkin hyvin mielenkiintoinen ongelma, mutta se ei onneksenne kuulu tähän kurssiin. Tarkistussumma lähetetään viestin mukana, ja tarkistetaan vastaanottopäässä. Jos viestissä on jotain vikaa, niin se pyydetään lähettämään uudelleen. Tiedonsiirto verkon yli on muutenkin niin hidasta, että virheenkorjauksella ei saavutettaisi merkittävää hyötyä verrattuna sen kustannuksiin. Sitäpaitsi, uudelleenlähetyksiä tarvitaan muutenkin kokonaan hävinneiden pakettien varalta.

CRC - Cyclic Redundancy Code

Tiedonsiirrossa verkon yli käytetty tarkistusmenetelmä

- jos virheitä tulee, niitä tulee usein paljon
- pariteettibitit eivät riitä, virheitä ei korjata

Tarkistussumma (16-bittinen CRC) isolle tietojoukolle

- laske CRC = f(viesti) % 2^{16} (eli ota 16 viimeistä bittiä)
- lähetä viesti ja CRC (2 ylimääräistä tavua)
- vastaanota viesti ja CRC
- laske CRC ja tarkista onko sama kuin viestin mukana oleva
- jos väärin, niin pyydä uudelleenlähetystä

CRC-CCITT CRC's detect:

All single-bit and double-bit errors

All errors of an odd number of bits

All error bursts of 16 bits or less

In summary, 99.998% of all errors

Copyright Teemu Kerola 2005

Esimerkiksi CCITT tarkistussumma havaitsee 99.998% kaikista bittivirheistä. Tämä tarkoittaa käytännössä, että 1/50000 osaa bittivirheistä ei havaita ja ne menevät tästä tarkistuksesta lävitse. On toivottavaa, että ne sitten havaitaan jossakin ylemmän tiedonsiirtotason tarkistuksissa. On siis tärkeätä tarkkaan määritellä mahdolliset virhelähteet, virheiden ilmenemismuodot ja hyväksyttävä riskitaso, ja sitten toteuttaa virheen havaitsemis- ja korjaustoimenpiteet näiden mukaisiksi.

Virheiden tarkistusmenetelmien käyttöalueet

Mitä lähempänä suoritinta, sitä tärkeämpää tiedon muuttumattomuus on

- suorittimen sisällä siis erittäin tärkeätä

Sisäinen väylä, muistiväylä, rekisterit, levy

- virheet lennossa korjaava Hamming-koodi

Paikallisverkko, Intranet, Internet

- uudelleenlähetyksen vaativa CRC

Copyright Teemu Kerola 2005

Järjestelmän eri tasoilla käytetään siis erilaisia tarkistusmenetelmiä. Perusperiaatteena on, että mitä lähempänä suoritinta ollaan, sitä tärkeämpää on tiedon muuttumattomuus. Erityisesti, suorittimen sisällä kaiken datan eheyttä pitää valvoa maksimaalisen hyvin, vaikka siitä aiheutuisikin suuria kustannuksia. Suorittimen laskennan lopputuloksen täytyy aina olla matemaattisesti oikein.

Virheiden tarkistusmenetelmien käyttöalueet

Mitä lähempänä suoritinta, sitä tärkeämpää tiedon muuttumattomuus on

- suorittimen sisällä siis erittäin tärkeätä

Sisäinen väylä, muistiväylä, rekisterit, levy

- virheet lennossa korjaava Hamming-koodi

Paikallisverkko, Intranet, Internet

- uudelleenlähetyksen vaativa CRC

Copyright Teemu Kerola 2005

Suorittimen sisällä tiedon muuttumattomuutta valvotaan yleisesti Hamming-koodilla. Tästä aiheutuu huomattavia kustannuksia ylimääräisten bittien tallettamisen ja niiden vaatimien ylimääräisten johtimien vuoksi. Toisaalta, tiedon muuttumattomuus tällä tasolla on niin tärkeätä, että nuo ns. 'ylimääräiset' kustannukset ovat olennainen osa suorittimen ja piirikortin toteutusta. Tiedon talletukseen käytetyt levyköt voidaan myöskin suojata pariteettibiteillä tai Hamming-koodilla. Palaamme näihin RAID-levyihin myöhemmällä luennolla.

Virheiden tarkistusmenetelmien käyttöalueet

Mitä lähempänä suoritinta, sitä tärkeämpää tiedon muuttumattomuus on

- suorittimen sisällä siis erittäin tärkeätä

Sisäinen väylä, muistiväylä, rekisterit, levy

- virheet lennossa korjaava Hamming-koodi

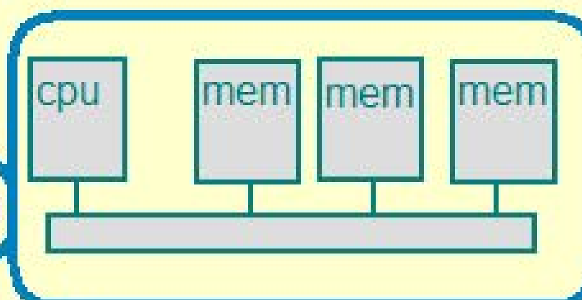
Paikallisverkko, Intranet, Internet

- uudelleenlähetyksen vaativa CRC

Copyright Teemu Kerola 2005

Verkon yli kommunikoidessa ei virheen korjaavaa koodia yleensä käytetä. Tiedonsiirto on suhteellisen hidasta ja havaitun virheen yhteydessä yleensä pyydetään lähettäjää eksplisiittisesti lähettämään paketti uudelleen. Toinen vaihtoehto on jättää paketin perilletulo kuittaamatta, jolloin pienen ajan kuluttua lähettäjä päättelee koko paketin hävinneen ja lähettää sen uudelleen.

Laitteiden monistaminen



Monta muistipiiriä tai levyä, samat tiedot monistettu

Monta suoritinta, samat käskyjen suoritukset monistettu

- äänestys, enemmistö voittaa

Monta laitteistoa, samat ohjelmistot monistettu

- äänestys, enemmistö voittaa
- monimutkainen valvonta? hidas?
- virheitä tekevä laitteisto suljetaan pois häiriköimästä

Monta samaa tai erityyppistä laitteistoa, samankaltaiset ohjelmistot

- ohjelmistojen toteutuksessa samat speksit, eri ohjelmoijat
- suorituksessa monta laitteistoa, samoilla syötteillä
- äänestys, enemmistö voittaa

Copyright Teemu Kerola 2005

Tiedon muuttumattomuutta ja eheyttä voidaan myös valvoa vielä tehokkaammin, jos tarpeet ja rahat vain riittävät. Jos muistipiirien toiminta halutaan varmennettua, niin on helppo rakentaa järjestelmä, jossa kukin muistipiiri on monistettu. Tällöin mikään edes yhden kokonaisen muistipiirin viottuminen ei haittaa, koska samat tiedot ovat tallessa kahdella muulla identtisellä piirillä. Kustannukset tietenkin kasvavat. Muistia tarvitaan enemmän ja datan vertailuun tarvitaan suoritinaikaa tai erikoislaitteistoa.

Laitteiden monistaminen

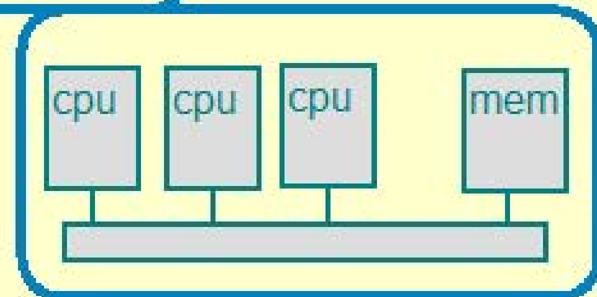
Monta muistipiiriä tai levyä, samat tiedot monistettu

Monta suoritinta, samat käskyjen suoritukset monistettu

- äänestys, enemmistö voittaa

Monta laitteistoa, samat ohjelmistot monistettu

- äänestys, enemmistö voittaa
- monimutkainen valvonta? hidas?
- virheitä tekevä laitteisto suljetaan pois häiriköimästä



Monta samaa tai erityyppistä laitteistoa, samankaltaiset ohjelmistot

- ohjelmistojen toteutuksessa samat speksit, eri ohjelmoijat
- suorituksessa monta laitteistoa, samoilla syötteillä
- äänestys, enemmistö voittaa

Copyright Teemu Kerola 2005

Jos taas suorittimen toiminta halutaan varmistaa, niin monistetaan suorittimia. Esimerkiksi Intelin i432 järjstelmässä neljä suoritinta saattoi tehdä kaikkea laskentaa. Joka väylälle kirjoittamisen yhteydessä sitten valvottiin, että kaikilla suorittimilla oli sama käsitys väylälle kirjoitettavasta datasta. Jos oli niin, yksi sai sen kirjoittaa. Jos ei ollut, niin joku häiriköi. Jos häiriköinti toistui, niin kyseinen suoritin voitiin automaattisesti sulkea pois jatkossa. Operaattori pystyi sitten vaihtamaan rikki menneen suorittimen ehjään yksinkertaisest vain piirikorttia vaihtamalla.

Laitteiden monistaminen

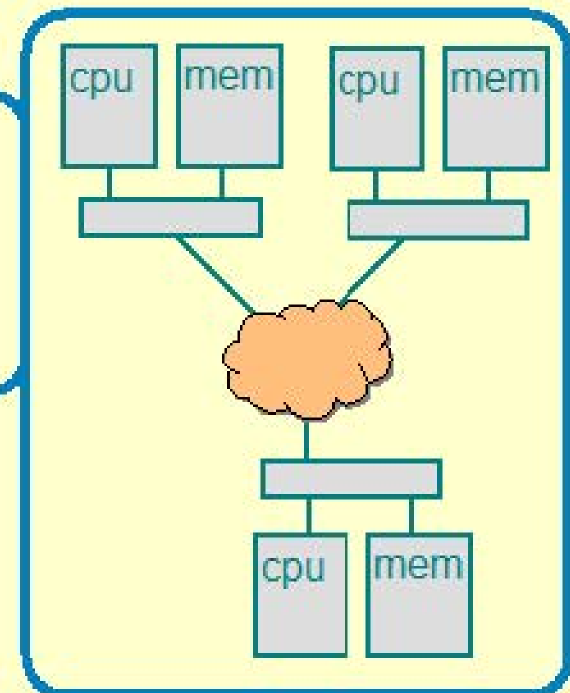
Monta muistipiiriä tai levyä, samat tiedot monistettu

Monta suoritinta, samat käskyjen suoritukset monistettu

- äänestys, enemmistö voittaa

Monta laitteistoa, samat ohjelmistot monistettu

- äänestys, enemmistö voittaa
- monimutkainen valvonta? hidas?
- virheitä tekevä laitteisto suljetaan pois häiriköimästä



Monta samaa tai erityyppistä laitteistoa, samankaltaiset ohjelmistot

- ohjelmistojen toteutuksessa samat speksit, eri ohjelmoijat
- suorituksessa monta laitteistoa, samoilla syötteillä
- äänestys, enemmistö voittaa

Copyright Teemu Kerola 2005

Lentokoneissa yleensä käytetään useaa rinnakaista kokonaista järjestelmää, jotka aina aika ajoin synkronoivat toimintansa. Esimerkiksi, lentopintoja ei saa muuttaa, ellei enemmistö järjestelmistä ole samaa mieltä. Jos joku järjestelmä osoittautuu vialliseksi, niin se voidaan lennon aikana vaihtaa mukana olevaan varakoneeseen.

(http://www.hq.nasa.gov/office/pao/History/computers/contents.html)

Laitteiden monistaminen

"Four of the five computers (IBM AP-101) on the Columbia ran identical software and compared results with each other before giving the go-ahead to take a specific action. The fifth computer (also IBM AP-101) ran a different

Monta muistipiiriä tai levyä, samat tiedot monistettu

Monta suoritinta, samat käskyjen suoritukset monistettu

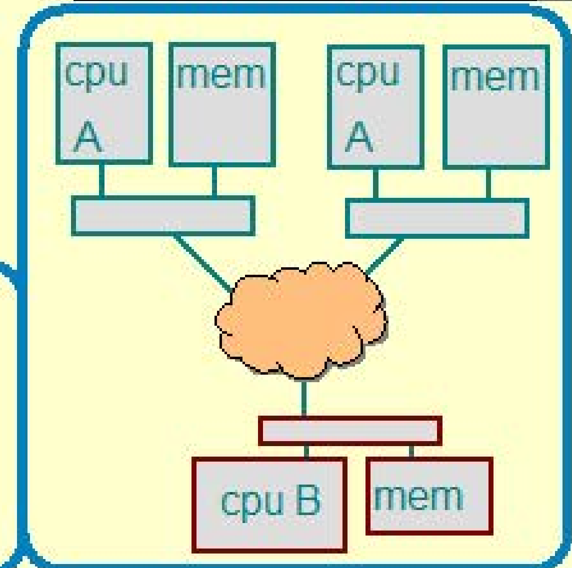
- äänestys, enemmistö voittaa

Monta laitteistoa, samat ohjelmistot monistettu

- äänestys, enemmistö voittaa
- monimutkainen valvonta? hidas?
- virheitä tekevä laitteisto suljetaan pois häirikoimästä

Monta samaa tai erityyppistä laitteistoa, samankaltaiset ohjelmistot

- ohjelmistojen toteutuksessa samat speksit, eri ohjelmoijat
- suorituksessa monta laitteistoa, samoilla syötteillä
- äänestys, enemmistö voittaa



Copyright Teemu Kerola 2005

Ydinvoimaloissa ja avaruusaluksissa voi käyttää vielä vaativampia redundansseja systeemejä. Jo Apollo-aluksissa oli mukana kahden eri valmistajan tekemiä järjestelmiä, joissa oli toisistaan riippumattomien ohjelmistotiimien tuottamat täysin erilliset ohjelmistot. Tällä tavoin pyrittiin minimoimaan ohjelmistoissa olevien katastrofaalisten virheiden vaikutus. Oli mahdollista milloin tahansa luopua pääasiallisesta monen identtisen koneen järjestelmästä ja vaihtaa varalaitteistoon, jossa ohjelmistokin oli eri tiimin tekemä. Useimmiten kannattaisi ehkä kuitenkin tehdä yksi ohjelmisto erinomaisesti kuin kaksi oikein hyvin, vai kannattaisiko?

Välimuistin (cache memory) perusidea

Ongelma: keskusmuisti on aika kaukana suorittimesta

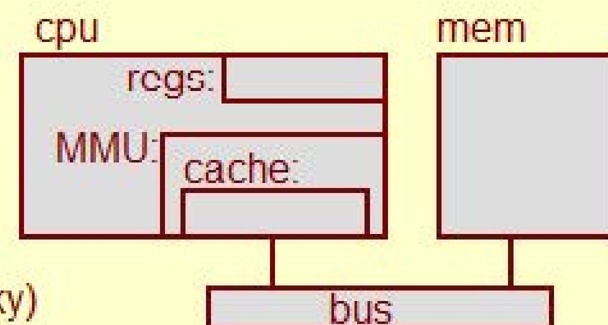
rekisterin viittausaika:	X
muistin viittausaika:	10X

Ratkaisu: pidetään osa keskusmuistin tiedoista erikoislaitteistossa (välimuistissa) lähellä suoritinta

- suorittimella tai ainakin suorittimen puolella väylää
- välimuistissa pidetään kopioita viime aikoina viitatuista keskusmuistin alueista

Jokainen muistiviite on nyt seuraavanlainen

- varmista, että tieto on välimuistissa?
  - jos tietoa ei löydy välimuistista, hae se sinne
  - suoritin odottaa muistinoudon ajan
- tee viittaus välimuistissa olevaan tietoon (data tai käsky)
- (talleta välimuistissa oleva muutettu tieto muistiin)



Copyright Teemu Kerola 2005

Vaikka tietokoneen keskusmuisti tuntuu ihmisen mittakaavassa olevan älyttömän nopea, niin suorittimen näkökulmasta se hidas. Muisti on vielä nykyäänkin toteutettu erillisillä piireillä suorittimesta ja muistiviitteen tekemiseksi tietoa ja kontrollisignaaleja pitää siirtää aika lailla muistiväylää pitkin. Tämän vuoksi tiedon hakemiseen muistista kuluu huomattavasti enemmän aikaa kuin esimerkiksi tiedon hakuun suorittimen sisäisestä rekisteristä. Muistelkaapa juustokakkuesimerkkiä luennolta 1. Muistiviittaus kestää niin kauan, että useat nykyiset arkkitehtuurit eivät salli muistiviitteitä esimerkiksi aritmetiikkakäskyjen yhteydessä.

Välimuistin (cache memory) perusidea

Ongelma: keskusmuisti on aika kaukana suorittimesta

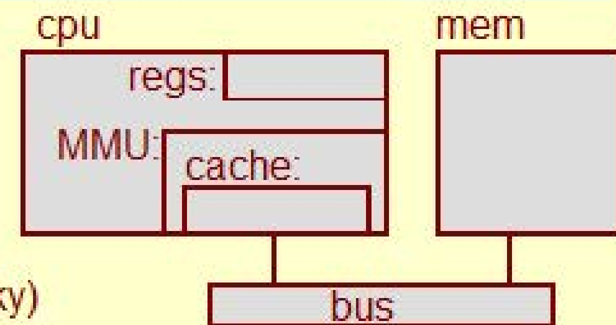
Ratkaisu: pidetään osa keskusmuistin tiedoista erikoislaitteistossa (välimuistissa) lähellä suoritinta

- suorittimella tai ainakin suorittimen puolella väylää
- välimuistissa pidetään kopioita viime aikoina viitatuista keskusmuistin alueista

rekisterin viittausaika:	X
välimuistin viittausaika:	2X
muistin viittausaika:	10X

Jokainen muistiviite on nyt seuraavanlainen

- varmista, että tieto on välimuistissa?
  - jos tietoa ei löydy välimuistista, hae se sinne
  - suoritin odottaa muistinoudon ajan
- tee viittaus välimuistissa olevaan tietoon (data tai käsky)
- (talleta välimuistissa oleva muutettu tieto muistiin)



Copyright Teemu Kerola 2005

Ratkaisuna muistin hitauteen on lisälaitteisto, välimuisti. Välimuisti toteutetaan joko samalle lastulle suorittimen kanssa tai välittömästi sen yhteyteen, kuitenkin niin, että välimuistiin päästään käsiksi ilman muistiväylän käyttöä. Jos siis tieto löytyy välimuistista, se löytyy sekä nopeasti että rasittamatta muistiväylää. Väimuisti ei näy mitenkään suorittavalle ohjelmalle, vaan sen käyttö on automaattista siten, että jotkut keskusmuistin viimeksi käytetyistä alueista ovat kopioituna välimuistiin. Käskyn suorituksen alkaessa ei tiedetä, onko käskyn viittaama muistialue välimuistissa vai ei.

Välimuistin (cache memory) perusidea

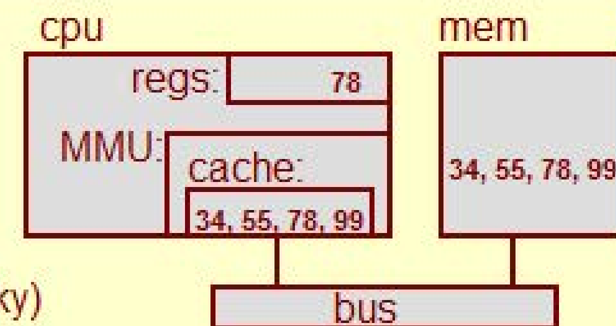
Ongelma: keskusmuisti on aika kaukana suorittimesta

Ratkaisu: pidetään osa keskusmuistin tiedoista erikoislaitteistossa (välimuistissa) lähellä suoritinta

- suorittimella tai ainakin suorittimen puolella väylää
- välimuistissa pidetään kopioita viime aikoina viitatuista keskusmuistin alueista

Jokainen muistiviite on nyt seuraavanlainen

- varmista, että tieto on välimuistissa?
  - jos tietoa ei löydy välimuistista, hae se sinne
  - suoritin odottaa muistinoudon ajan
- tee viittaus välimuistissa olevaan tietoon (data tai käsky)
- (talleta välimuistissa oleva muutettu tieto muistiin)



Copyright Teemu Kerola 2005

Välimuistin olemassaolo vaikuttaa jokaisen konekäskyn suoritukseen. Jokainen konekäskyhän periaatteessa haetaan muistista ja se voi vielä suoritusaikana tehdä lisääkin muistiviitteitä. Jokainen muistiviite tapahtuu nyt välimuistin kautta. Muistista luku on vielä yksinkertaista, koska tieto kopioidaan välimuistiin aina kun siihen on kohdistunut lukuoperaatio ja viitattu alue ei ennestään ole välimuistissa. Yleensä välimuistiin haetaan aina muutama sana kerrallaan, koska järjestelmä toimii näin tehokkaammin. Kirjoitukset ovat ongelmallisia, koska tieto pitää ehkä ensin hakea keskusmuistista ja joskus myös kirjoittaa sinne.

Välimuistin piirteitä

Tuntumaton suorittimelle

- jos viitattu tieto ei ole saatavilla, niin suoritin odottaa muistinoudon ajan
- hyperthread suoritin: suorita toista ohjelmaa muistiviitteen odotuksen aikana

Toteutettu usein nopeammalla teknologialla ku

Toteutettu usein samalla lastulla kuin suoritin

Ei täysin ratkaise nopeusero-ongelmaa

- rekisteri X, välimuisti 2X, muisti 10X
- monen tasoisia välimuisteja: rekisteri X, välimuisti

Ttk-91 koneessa ei ole välimuistia

Lisää tietoa muilta kursseilta

- Tietokoneen rakenne ja Käyttöjärjestelmät

Hyper-Threading Technology provides thread-level-parallelism (TLP) on each processor resulting in increased utilization of processor execution resources. As a result, resource utilization yields higher processing throughput. Hyper-Threading Technology is a form of simultaneous multi-threading technology (SMT) where multiple threads of software applications can be run simultaneously on one processor. This is achieved by duplicating the architectural state on each processor, while sharing one set of processor execution resources. Hyper-Threading Technology also delivers faster response times for multi-tasking workload environments. By allowing the processor to use on-die resources that would

http://www.intel.com/technology/hyperthread/

Copyright Teemu Kerola 2005

Välimuisti on siis tuntumaton suorittimelle. Suorittimen ja sillä suorittavien ohjelmien ei tarvitse tietää välimuistista lainkaan. Riittää, kun useimmiten keskusmuisti tuntuu toimivan huomattavasti nopeammin, kuin mitä voisi tapahtua ilman välimuistia. On myös mukavaa ohjelmoijien näkökulmasta, että heidän ei tarvitse ohjelmia suunnitellessaan pohtia, mikä tieto pidetään välimuistissa ja milloin. Suoritin voi myös hyödyntää muistiviitteen odotteluajan muiden käskyjen ja jopa muiden ohjelmien laskennassa, kuten esimerkiksi Intelin Pentium 4 Hyperthread -suorittimet tekevät.

Välimuistin piirteitä

Tuntumaton suorittimelle

- jos viitattu tieto ei ole saatavilla, niin suoritin odottaa muistinoudon ajan
- hyperthread suoritin: suorita toista ohjelmaa muistiviitteen odotuksen aikana

Toteutettu usein nopeammalla teknologialla kuin keskusmuisti

Toteutettu usein samalla lastulla kuin suoritin

Ei täysin ratkaise nopeusero-ongelmaa

- rekisteri X, välimuisti 2X, muisti 10X
- monen tasoisia välimuisteja: rekisteri X, välimuisti 2X, 4X, 6X, muisti 10X

Ttk-91 koneessa ei ole välimuistia

Lisää tietoa muilta kursseilta

- Tietokoneen rakenne ja Käyttöjärjestelmät

Copyright Teemu Kerola 2005

Välimuistien nopeus tulee siitä, että (a) ne ovat lähellä suoritinta, (b) ne ovat pieniä ja (c) ne on toteutettu nopeammalla teknologialla kuin keskusmuisti. Useissa tapauksissa välimuisti on toteutettu suoraan samalle lastulle suorittimen kanssa, jolloin niiden toteutusteknologia on tietenkin samanlainen kuin suorittimenkin.

Välimuistin piirteitä

Tuntumaton suorittimelle

- jos viitattu tieto ei ole saatavilla, niin suoritin odottaa muistinoudon ajan
- hyperthread suoritin: suorita toista ohjelmaa muistiviitteen odotuksen aikana

Toteutettu usein nopeammalla teknologialla kuin keskusmuisti

Toteutettu usein samalla lastulla kuin suoritin

Ei täysin ratkaise nopeusero-ongelmaa

- rekisteri X, välimuisti 2X, muisti 10X
- monen tasoisia välimuisteja: rekisteri X, välimuisti 2X, 4X, 6X, muisti 10X

Intel XEON MP	
L1 välimuisti	8 KB
L2 välimuisti	512 KB
L3 välimuisti	1024 KB

Ttk-91 koneessa ei ole välimuistia

Lisää tietoa muilta kursseilta

- Tietokoneen rakenne ja Käyttöjärjestelmät

Copyright Teemu Kerola 2005

Suorittimen yhteydessä oleva välimuisti ei täysin ratkaise eri laitteistojen nopeuseroista aiheutuvia ongelmia. Jos välimuistin ja keskusmuistin välinen nopeusero on edelleen 5-kertainen, niin se voi olla liikaa. Tämä ongelma taas on ratkaistu rakentamalla useamman tasoisia välimuisteja

Välimuistin piirteitä

Tuntumaton suorittimelle

- jos viitattu tieto ei ole saatavilla, niin suoritin odottaa muistinoudon ajan
- hyperthread suoritin: suorita toista ohjelmaa muistiviitteen odotuksen aikana

Toteutettu usein nopeammalla teknologialla kuin keskusmuisti

Toteutettu usein samalla lastulla kuin suoritin

Ei täysin ratkaise nopeusero-ongelmaa

- rekisteri X, välimuisti 2X, muisti 10X
- monen tasoisia välimuisteja: rekisteri X, välimuisti 2X, 4X, 6X, muisti 10X

Ttk-91 koneessa ei ole välimuistia

Lisää tietoa muilta kursseilta

- Tietokoneen rakenne ja Käyttöjärjestelmät

Copyright Teemu Kerola 2005

Esimerkkikoneessamme ttk-91 ei ole välimuistia, vaikka mikään ei tietenkään estäisi sen toteuttamista myös ttk-91'een. Toisaalta, välimuistin tarkempi tuntemus ei kuulu tälle kurssille. Välimuistin toimintaan ja tarkempaan rakenteeseen tutustutaan Tietokoneen rakenne -kurssilla ja välimuistin hyödyntämistä ja huomioonottamista samanaikaisuuden hallinnassa käsitellään lisää käyttöjärjestelmäkursseilla.

Muistin toteutus

Eri teknologioita eri tasoisiin muisteihin

- suorittimen rekisteri
- välimuisti, tasot L1, L2, L3
- keskusmuisti

RAM eli Random-Access semiconductor Memory

- anna osoite ja lue/kirjoita signaali
- mistä vain voi lukea/kirjoittaa samassa ajassa
- häviävää muistia: tieto pyyhkiytyy virran katketessa (volatile memory)
- kaikki nykyiset muistit ovar "random access"

Copyright Teemu Kerola 2005

Tietokonejärjestelmässä on siis monentasoisia muisteja. Huomaa, että myös suorittimen rekisterit ovat muistia, vaikka ne onkin nimetty vähän erikoisella tavalla. Myös välimuistit ovat muistia, vaikka niihin ohjelmissa ei voi suoraan osoitteella viitata. Yleensä käsitteellä muisti tarkoitetaan kuitenkin keskusmuistia, mutta älkää unohtako näitä muita, vaikka ne onkin nimetty eri tavalla. Näiden eri tasoisten muistien toteutuksessa käytetään erilaisia toteutusteknologioita, joiden avulla kunkin tason muistin koko ja nopeus saadaan sille tasolle sopivaksi.

Muistin toteutus

Eri teknologioita eri tasoisiin muisteihin

- suorittimen rekisteri
- välimuisti, tasot L1, L2, L3
- keskusmuisti

RAM eli Random-Access semiconductor Memory

- anna osoite ja lue/kirjoita signaali
- mistä vain voi lukea/kirjoittaa samassa ajassa
- häviävää muistia: tieto pyyhkiytyy virran katketessa (volatile memory)
- kaikki nykyiset muistit ovar "random access"

Copyright Teemu Kerola 2005

Kaikki muisti nykyisissä koneissa on hajasaanti eli random access muistia, mikä tarkoittaa yksinkertaisesti sitä, että jokainen muistipaikka on aina yhtä nopeasti saavutettavissa. Joskus kauan aikaa sitten tietokoneiden varhaishistoriassa oli myös muistiteknologioita, joissa tietyt muistiosoitteet olivat jollakin hetkellä 'lähempänä' eli nopeammin saavutettavissa kuin muut. Nimi random access juontaa noilta ajoilta ja sitä käytetään nyt yleisnimenä keskusmuistille. Nykyisin käytössä olevan muistiteknologian huonona puolena on RAM-muistin häviävyys eli tietojen haipuvuus unholaan virran katkettua.

RAM muistin kaksi toteutusteknologiaa

DRAM eli dynaaminen RAM

- halvempi, hitaampi
- tietoja pitää virkistää vähän väliä (esim. joka 2 ms)
- tavallinen keskusmuisti useimmissa koneissa nyt
- toteutettu kondensaattoreilla, joiden varaus "vuotaa"

SRAM eli staattinen RAM

- kalliimpi (esim. 10x), nopeampi (esim. 10x)
- ei vaadi tietojen virkistämistä
- toteutettu samalla teknologialla kuin suorittimen logiikka
- välimuisti useimmissa koneissa (sama tai eri lastu suorittimen kanssa)
- keskusmuisti superkoneissa ja high-end palvelimissa

SDRAM, DDR SDRAM, FPM DRAM, EDO DRAM, BEDO DRAM, EDRAM, Video-RAM, Direct Rambus DRAM, SLDRAM, ECC DRAM, ...

- erilaisia optimoituja ratkaisuja edellisistä
- nopeus, kaistanleveys, koko, virheentarkistus, hinta?

Copyright Teemu Kerola 2005

Tavallisen muistin toteutusteknologioita on kahta perustyyppiä: DRAM ja SRAM. DRAM on vähän vanhempaa teknologiaa, mutta edelleen halvempaa ja hitaampaa kuin SRAM. DRAM'in huonona puolena on sen toteutusmenetelmä kondensaattoreilla, joissa olevaa varausta pitää aika ajoin virkistää tiedon säilymiseksi. DRAM'in hyvänä puolena on halpuus, minkä vuoksi se on tällä hetkellä yleisin keskusmuistin toteutusteknologia.

RAM muistin kaksi toteutusteknologiaa

DRAM eli dynaaminen RAM

- halvempi, hitaampi
- tietoja pitää virkistää vähän väliä (esim. joka 2 ms)
- tavallinen keskusmuisti useimmissa koneissa nyt
- toteutettu kondensaattoreilla, joiden varaus "vuotaa"

SRAM eli staattinen RAM

- kalliimpi (esim. 10x), nopeampi (esim. 10x)
- ei vaadi tietojen virkistämistä
- toteutettu samalla teknologialla kuin suorittimen logiikka
- välimuisti useimmissa koneissa (sama tai eri lastu suorittimen kanssa)
- keskusmuisti superkoneissa ja high-end palvelimissa

SDRAM, DDR SDRAM, FPM DRAM, EDO DRAM, BEDO DRAM, EDRAM, Video-RAM, Direct Rambus DRAM, SLDRAM, ECC DRAM, ...

- erilaisia optimoituja ratkaisuja edellisistä
- nopeus, kaistanleveys, koko, virheentarkistus, hinta?

Copyright Teemu Kerola 2005

Staattinen RAM toteutetaan samalla teknologialla kuin suorittimen logiikka. Tähän tutustutaan vähän (mutta ei paljon) tarkemmin jatkokurssilla Tietokoneen rakenne. SRAM on kalliimpi, mutta nopeampi teknologia, joten se sopii hyvin tavallisten suorittimien välimuistin toteutukseen. Välimuistin kannalta on myös mukavaa, että SRAM voidaan toteuttaa samalla lastulla samalla teknologialla kuin suoritinkin. Suurta nopeutta vaativissa koneissa myös koko keskusmuisti voi olla toteutettu SRAM'illa vaikka silloinkin se on suuren koonsa vuoksi hitaampaa kuin samalla teknologialla toteutettu pienempi välimuisti.

RAM muistin kaksi toteutusteknologiaa

DRAM eli dynaaminen RAM

- halvempi, hitaampi
- tietoja pitää virkistää vähän väliä (esim. joka 2 ms)
- tavallinen keskusmuisti useimmissa koneissa nyt
- toteutettu kondensaattoreilla, joiden varaus "vuotaa"

SRAM eli staattinen RAM

- kalliimpi (esim. 10x), nopeampi (esim. 10x)
- ei vaadi tietojen virkistämistä
- toteutettu samalla teknologialla kuin suorittimen logiikka
- välimuisti useimmissa koneissa (sama tai eri lastu suorittimen kanssa)
- keskusmuisti superkoneissa ja high-end palvelimissa

SDRAM, DDR SDRAM, FPM DRAM, EDO DRAM, BEDO DRAM, EDRAM, Video-RAM, Direct Rambus DRAM, SLDRAM, ECC DRAM, ...

- erilaisia optimoituja ratkaisuja edellisistä
- nopeus, kaistanleveys, koko, virheentarkistus, hinta?

Copyright Teemu Kerola 2005

Muistiteknologioita on näistä perustapauksista kehitetty vielä paljon pidemmälle, mutta nämä piirteet eivät kuulu tälle kurssille. Mielenkiintoinen yksittäistapaus on Rambus-muisti, jossa DRAM on ensin nopeutettu perustoteutuksesta ja sitten siihen on liitetty oma erikoismuistiväylän ohjain, minkä avulla useasta DRAM-palikasta koottu kokonaisuus saadaan huomattavasti perustoteutusta tehokkaammaksi. Rambusin käyttö ei ole kovin paljoa yleistynyt, mutta Sony on lisenssoinut sen uudelle PlayStation-3 suorittimelleen, jossa pelitoteutukset tarvitsevat hyvin nopeata ja laajakaistaista muistia.

ROM teknologia

ROM eli Read-Only memory

Haipumaton, tieto säilyy ilman sähkövirtaa — non-volatile

ROM on myös RAM

- kaikki muistipaikat yhtä 'lähellä'
- yleensä hitaampi kuin RAM

Järjestelmän alustustiedot (BIOS) — koodi

- alusta väylä ja boot-kelpoiset laitteet (kovalevy, CD, DVD, ...)
- käy laitteet läpi, kunnes jostakin löytyy boot-lohko
- järjestelmän alustus boot-lohkon avulla

Alkuaan: järjestelmän alustus, kirjoitus kerran, luetaan usein

Sitten: järjestelmän alustus, kirjoitus harvoin, luetaan usein

Nyt: eri käyttötarkoituksia, kirjoitus ehkä aika usein, luku ehkä aika usein

- vähän kallis ja vähän hidas, mutta haipumaton muisti — Flash — koodi ja data
- järjestelmän alustus, digikamerat, muistitikut, yleinen tiedon henkilökohtainen tallennus

Copyright Teemu Kerola 2005

RAM-muistiteknologian suurena puutteena on tiedon häviäminen virran katkettua, mistä aiheutui ongelma. Miten alustaa tietokone muistissa olevasta ohjelmasta, kun muisti oli tyhjä virran kytkemisen jälkeen. Ongelman ratkaisuna oli pysyväismuisti, ROM, johon jollakin tavalla voidaan tallettaa tietoa, joka säilyy siellä ilman sähkövirtaa. Nimestään huolimatta myös ROM-muistit ovat hajasaantimuistia, mutta ne olivat alkuaan ja ne ovat edelleenkin hitaampia kuin tavallinen RAM. Tästä syystä useissa tapauksissa ohjelmakoodi on vähän turhan hidasta lukea ROM-muistista, vaan sen sijaan on usein edullista kopioida se ensin keskusmuistiin ja suorittaa sieltä.

ROM teknologia

ROM eli Read-Only memory

Haipumaton, tieto säilyy ilman sähkövirtaa — non-volatile

ROM on myös RAM

- kaikki muistipaikat yhtä 'lähellä'
- yleensä hitaampi kuin RAM

Järjestelmän alustustiedot (BIOS) — koodi

- alusta väylä ja boot-kelpoiset laitteet (kovalevy, CD, DVD, ...)
- käy laitteet läpi, kunnes jostakin löytyy boot-lohko
- järjestelmän alustus boot-lohkon avulla

Alkuaan: järjestelmän alustus, kirjoitus kerran, luetaan usein

Sitten: järjestelmän alustus, kirjoitus harvoin, luetaan usein

Nyt: eri käyttötarkoituksia, kirjoitus ehkä aika usein, luku ehkä aika usein

- vähän kallis ja vähän hidas, mutta haipumaton muisti — Flash — koodi ja data
- järjestelmän alustus, digikamerat, muistitikut, yleinen tiedon henkilökohtainen tallennus

Copyright Teemu Kerola 2005

ROM-muistin alkuperäinen käyttö oli nimenomaan järjestelmän alustamisessa. Alustuskoodissa ensin annetaan alkuarvot usealle systeemimuuttujalle ja alustetaan kaikki sellaiset laitteet, joilta järjestelmän voisi bootata. Seuraavaksi boottikelpoiset laitteet käydään läpi ennaltamäärätyssä järjestyksessä ja valitaan ensimmäinen laite, jolta löytyy suorituskelpoinen boottikoodi. Alustuksen voi myös keskeyttää, usein jollakin sovitulla näppäimellä, ja antaa käyttäjän myös muuttaa tätä alustuskoodia tai systeemiparametrien arvoja. Tätä ei tietenkään kannata tehdä, ellei tunne järjestelmän toimintaa aika hyvin.

ROM teknologia

ROM eli Read-Only memory

Haipumaton, tieto säilyy ilman sähkövirtaa — non-volatile

ROM on myös RAM

- kaikki muistipaikat yhtä 'lähellä'
- yleensä hitaampi kuin RAM

Järjestelmän alustustiedot (BIOS) — koodi

- alusta väylä ja boot-kelpoiset laitteet (kovalevy, CD, DVD, ...)
- käy laitteet läpi, kunnes jostakin löytyy boot-lohko
- järjestelmän alustus boot-lohkon avulla

Alkuaan: järjestelmän alustus, kirjoitus kerran, luetaan usein

Sitten: järjestelmän alustus, kirjoitus harvoin, luetaan usein

Nyt: eri käyttötarkoituksia, kirjoitus ehkä aika usein, luku ehkä aika usein

- vähän kallis ja vähän hidas, mutta haipumaton muisti — Flash — koodi ja data
- järjestelmän alustus, digikamerat, muistitikut, yleinen tiedon henkilökohtainen tallennus

Copyright Teemu Kerola 2005

Alkuaan ROM-muistit olivat vain erkoistarkoituksiin suunniteltuja muistipiirejä, joita hintansa ja kirjoitusoperaation vaikeuden vuoksi käytettiin nimenomaan alustamaan laitteistoja. Niissä olevat ohjelmat ja data olivat staattisia. Nykyisin ROM-muistit eivät ole oikeastaan lainkaan enää Read-Only tyyppisiä muisteja, vaan pikemminkin yleisiä tietonsa ilman sähkövirtaa säilyttäviä muistipiirejä, joiden eräs käyttöalue on tietokoneiden alustuskoodin ja -tietojen säilytys. ROM-muisti nimenäkin alkaa jo jäädä unholaan sen uusimman toteutusteknologian, Flash-muistin, nimen jalkoihin. Ne ovat kuitenkin edelleen hitaampia ja vähän kalliimpia kuin RAM.

ROM muistiteknologian kehitys

Mask-ROM

- kirjoitus lastun valmistuksen yhteydessä

PROM eli Programmable ROM

- kirjoitus tyhjälle lastulle jälkikäteen "polttamalla" ("burn") erikoislaitteistolla

EPROM eli Erasable PROM

- koko lastu voidaan nollata (tyhjentää) UV-säteilyllä erikoislaitteistolla
- lastun uudelleenkäyttö mahdollista

EEPROM eli Electronically Erasable PROM

- tiedot voidaan pyyhkiä tavukohtaisesti suurella jännitteellä erikoislaitteistolla

Flash eli Flash EEPROM

- tietoja voidaan pyyhkiä normaalijännitteellä konekäskyillä lastun ollessa paikallaan
- nopeampi kuin tavallinen EEPROM

Copyright Teemu Kerola 2005

Ensimmäiset ROM'it olivat staattisia. Niihin talletettiin ohjelma ja data lastun valmistushetkellä, samalla tavalla kuin suorittimen toteutuslogiikkakin. Jos ohjelmassa tai datassa oli virhe, niin sitä ei voinut mitenkään korjata. Ainoa mahdollisuus oli tilata uusi lastu valmistajalta, mikä oli aika kömpelöä. Lisäksi kaikki lastulle menevä ohjelma ja data piti etukäteen antaa lastutehtaalle, mikä ei myöskään ollut aina lastujen käyttäjäyrityksille mieleen. Lastulle talletettavien ohjelmistojen kehitys oli tarkkaa puuhaa.

ROM muistiteknologian kehitys

Mask-ROM

- kirjoitus lastun valmistuksen yhteydessä

PROM eli Programmable ROM

- kirjoitus tyhjälle lastulle jälkikäteen "polttamalla" ("burn") erikoislaitteistolla

EPROM eli Erasable PROM

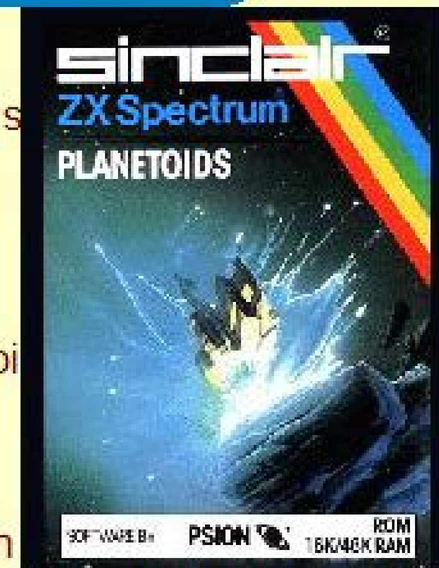
- koko lastu voidaan nollata (tyhjentää) UV-säteilyllä erikoislaitteis
- lastun uudelleenkäyttö mahdollista

EEPROM eli Electronically Erasable PROM

- tiedot voidaan pyyhkiä tavukohtaisesti suurella jännitteellä erikoi

Flash eli Flash EEPROM

- tietoja voidaan pyyhkiä normaalijännitteellä konekäskyillä lastun
- nopeampi kuin tavallinen EEPROM

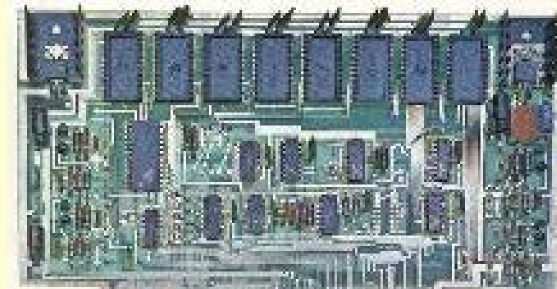


Copyright 1982 Psion Ltd

Copyright Teemu Kerola 2005

Seuraava askel oli selvä parannus tilanteeseen. Lastut voitiin ostaa tyhjinä aihioina tehtaalta ja sitten asiakkaan omassa pajassa ohjelmoida. Jos ensimmäinen versio oli virheellinen, niin kaikkia samalla kertaa tilattuja lastuja ei tarvinnut heittää roskiin. Lastut olivat kuitenkin edelleen kertakäyttöisiä. Nyt kuitenkin ensimmäisen kerran niitä voitiin räätälöidä eri tarkoituksiin suht'koht helposti. Tässä vaiheessa ROM-muisteja alettiin jo käyttää pelien levitykseen erillisillä ROM-muistikorteilla.

ROM muistiteknologian kehitys



Altair 88-PMC PROM Memory Board

Mask-ROM

- kirjoitus lastun valmistuksen yhteydessä

PROM eli Programmable ROM

- kirjoitus tyhjälle lastulle jälkikäteen "polttamalla" ("burn") erikoislaitteistolla

EPROM eli Erasable PROM

1971, Dov Frohman, Intel 1701

- koko lastu voidaan nollata (tyhjentää) UV-säteilyllä erikoislaitteistolla
- lastun uudelleenkäyttö mahdollista

"Read-Mostly Memory" keksitty

EEPROM eli Electronically Erasable PROM

- tiedot voidaan pyyhkiä tavukohtaisesti suurella jännitteellä erikoislaitteistolla

Flash eli Flash EEPROM

- tietoja voidaan pyyhkiä normaalijännitteellä konekäskyillä lastun ollessa paikalaan
- nopeampi kuin tavallinen EEPROM

Copyright Teemu Kerola 2005

Seuraava kehitysvaihe oli uusiokäyttöinen PROM eli EPROM. Lastu voitiin irroittaa piirilevyltä ja sijoit:aa (pienikokoiseen) ultraviolettiuuniin, jossa se tietyn aallonpituuden esim. 20 minuutin säteilytyksessä nollaantui, minkä jälkeen siihen voitiin 'polttaa' kopioimalla tavallisesta muistista uusi ohjelma ja/tai data. Tämä oli jälleen merkittävä askel käyttäjäystävällisyydessä, koska nyt ROM-muistin ohjelmointi oli käytännössä kenen tahansa tehtävissä. Altairilla oli esimerkiksi PROM-piirien 'polttamista' varten piirilevy, joka voitiin sijoittaa Altair-mikrotietokoneeseen.

ROM muistiteknologian kehitys

Altair 88-PMC8 8K PROM Memory Board

Mask-ROM

- kirjoitus lastun valmistuksen yhteydessä

PROM eli Programmable ROM

- kirjoitus tyhjälle lastulle jälkikäteen "polttamalla" ("burn") erikoislaitteistolla

EPROM eli Erasable PROM

- koko lastu voidaan nollata (tyhjentää) UV-säteilyllä erikoislaitteistolla
- lastun uudelleenkäyttö mahdollista

EEPROM eli Electronically Erasable PROM

1980, Intel 2816

- tiedot voidaan pyyhkiä tavukohtaisesti suurella jännitteellä erikoislaitteistolla

Flash eli Flash EEPROM

- tietoja voidaan pyyhkiä normaalijännitteellä konekäskyillä lastun ollessa paikallaan
- nopeampi kuin tavallinen EEPROM

Copyright Teemu Kerola 2005

Tästä vielä askel parempaan oli, kun lastua ei tarvinnut kokonaan tyhjentää, vaan siitä voitiin vain korjata virheellinen osa. Tämä oli toivottu käyttötapa esimerkiksi silloin, kun lastulla oleva tietty järjestelmän alustustieto haluttiin vain päivittää koskematta ohjelmistoon tai muuhun dataan. Myös tällaista käyttöä varten kehtettiin myös mikrotietokoneisiin sopivia laitteistoja.

ROM muistiteknologian kehitys

Mask-ROM

- kirjoitus lastun valmistuksen yhteydessä

PROM eli Programmable ROM

- kirjoitus tyhjälle lastulle jälkikäteen "polttamalla" ("burn") erikoislaitteistolla

EPROM eli Erasable PROM

- koko lastu voidaan nollata (tyhjentää) UV-säteilyllä erikoislaitteistolla
- lastun uudelleenkäyttö mahdollista

EEPROM eli Electronically Erasable PROM

- tiedot voidaan pyyhkiä tavukohtaisesti suurella jännitteellä erikoislaitteistolla

Flash eli Flash EEPROM

1984, Fujio Masuoka, Toshiba

- tietoja voidaan pyyhkiä normaalijännitteellä konekäskyillä lastun ollessa paikallaan
- nopeampi kuin tavallinen EEPROM

Copyright Teemu Kerola 2005

Nykytilanteeseen päästiin, kun kehitettiin Flash EEPROM, eli nykyään vain Flash-muisti, joka toimikin jo tavallisen muistipiirin tavoin. Flash-muistia voidaan lukea ja kirjoittaa tavallisen muistin tapaan, tosin isompi lohko yhdellä välähdyksellä eli Flash'llä. Yksittäisiä tavuja ei voida tyhjentää, mutta käytännössä tästä ei ole haittaa. Muu tyhjennettävällä lohkolla oleva data luetaan ensi tavalliseen muistiin ja kirjoitetaan sitten tyhjennyksen jälkeen takaisin. Flash-muisteja käytetään nykyisin esimerkiksi digikameroiden muisteina ja muistitikuissa.

Tiedon tarkistus ja muisti

Ohjelman esitysmuodot
- Konekäskyt ja rakenteellinen tieto

Tiedon muuttumattomuus
- Pariteettibitti
- Hamming-koodi
- CRC-koodi ja laitteiden monistaminen

Muisti
- Välimuisti, tavallinen muisti

Copyright Teemu Kerola 2005

Olemme nyt käyneet läpi kaiken mahdollisen tiedon esitysmuodon laitteistossa. On merkittävää muistaa, että suoritin tunnistaa vain muutaman tietotyypin ja loput käsitellään kuvaamalla ne ensin näihin muutamaan toteutettuun tietotyyppiin. Teillä pitäisi olla selkeä mielikuva tiedon muuttumattomuuden suojausmenetelmistä. Olemme myös käsitelleet tietokoneen muistin erilaisia toteutustapoja sekä perusidean välimuistin toimintatavasta keskusmuistin käyttöä nopeuttavana laitteistona. Muistipiirien toteutusmekanismeista kävimme läpi kolme eri nopeuksista toteutusta, jotka ovat hitaammasta nopeampaan flash, DRAM ja SRAM.