



RĒZEKNES AUGSTSKOLA
INŽENIERU FAKULTĀTE
DATORZINĀTŅU UN MATEMĀTIKAS
KATEDRA

Pēteris GRABUSTS

DATORU ARHITEKTŪRA

Rēzekne 2008



Materiāls tika izstrādāts Eiropas Savienības finansētā projekta „Studiju programmas „Inženieris programmētājs” modernizācija” ietvaros
(projekts Nr. 2007/0085/VPD1/ESF/PIAA/06/APK/3.2.3.2./0078/0160)

Recenzenti:

Latvijas Universitātes Fizikas un matemātikas fakultātes Datorikas nodaļas lektors
Rihards Rūmnieks

Rīgas Tehniskās universitātes Datoru tīklu un sistēmas katedras asoc.profesors
Dr.habil.dat. *Aldis Baums*

Redaktore: *Vita Ansona*

Mācību līdzeklī „Datoru arhitektūra” ir dots plašs pārskats par datorsistēmu konfigurāciju un datoru aparatūras programmēšanas galvenajiem principiem, apskatot svarīgākos datoru organizācijas un funkcionēšanas aspektus no programmētāja viedokļa, raksturota datorsistēmas ar Intel procesoru bāzi uzbūve, uzmanību akcentējot uz principiem, kas ir kopēji dažādas arhitektūras skaitļotājiem, kā arī izanalizētas būtiskāko datora arhitektūras komponentu īpatnības un to ietekme uz programmas koda kvalitāti.

Mācību līdzeklis paredzēts Rēzeknes Augstskolas Inženieru fakultātes studiju programmas „Programmēšanas inženieris” studentiem. To var izmantot arī citu datorzinātņu studiju programmu studenti, kā arī visi interesenti, kas patstāvīgi vēlas iepazīties ar mūsdienu datoru arhitektūras pamatnostādņem.

© **Pēteris Grabusts. Datoru arhitektūra**

© **Rēzeknes Augstskola**

Atbrīvošanas alejā 90, Rēzeknē, LV 4600

RA Izdevniecība, 2008

Atbrīvošanas alejā 115, Rēzeknē, LV 4600

Rēzekne, 2008. – 140 lpp.

Tirāža 100 eks.

ISBN 978-9984-779-86-7

SATURA RĀDĪTĀJS

	Lpp.
Ievads.....	4
Apzīmējumu skaidrojums.....	5
1. Datoru arhitektūras koncepcija.....	6
1.1. Datoru arhitektūras jēdziens.....	6
1.2. Neimana arhitektūra un tradicionālās datorsistēmas uzbūve.....	10
1.3. Galveno jēdzienu skaidrojums.....	13
2. Ievadizvades arhitektūra un atmiņas organizācija.....	19
2.1. Ievadizvades arhitektūra.....	19
2.2. PCI kopnes realizācijas piemērs.....	22
2.3. Ievadizvades ierīces.....	27
2.4. Fiziskās atmiņas organizācija.....	29
3. Atmiņas loģiskā organizācija un virtuālā atmiņa.....	34
3.1. Virtuālās atmiņas struktūra.....	34
3.2. Atmiņas pārvaldnieks.....	38
3.3. Virtuālā atmiņa un lietojumprogrammu veiktspēja.....	42
4. <i>Intel</i> procesoru organizācija un iespējas.....	47
4.1. Mūsdienu procesoru funkcionēšanas pamati.....	47
4.2. Komandu konveijerapstrāde.....	51
4.3. Komandu paralēlā izpilde.....	51
4.4. <i>Intel Pentium 4</i> arhitektūras īpatnības.....	53
5. Procesora programmu modelis.....	57
5.1. Procesora reģistri.....	57
5.2. Procesora komandu sistēma un datu adresācija.....	59
6. <i>Intel</i> procesoru multivides paplašinājums.....	69
6.1. MMX tehnoloģija.....	69
6.2. SSE un SSE2 tehnoloģija.....	75
7. Datu un komandu kešdarbe.....	82
8. Datorsistēmas ievadizvades apakšsistēma.....	90
8.1. Ievadizvades pieslēgvietas.....	90
8.2. Ievadizvades ierīces <i>Windows</i> vidē.....	93
9. Paralēlā pieslēgvietā un tās programmēšana.....	96
9.1. Paralēlās pieslēgvietas signāli.....	96
9.2. EPP režīms.....	100
9.3. ECP protokols.....	102
9.4. Lietotāja programmu saskarne.....	106
10. Datu ievadizvade caur seriālo pieslēgvietu.....	111
10.1. Seriālās pieslēgvietas raksturojums.....	111
10.2. RS-232 saskarne.....	113
10.3. Seriālās pieslēgvietas programmēšanas aspekti.....	115
11. USB kopnes arhitektūra.....	122
12. Datorsistēmu veiktspējas analīze.....	126
Izmantotās literatūras un avotu saraksts.....	139

IEVADS

Mērķis un uzdevumi

Mācību līdzekļa „Datoru arhitektūra” galvenais mērķis ir sniegt padziļinātas zināšanas par datorsistēmu konfigurāciju un datoru aparatūras programmēšanas galvenajiem principiem ar nolūku:

- apskatīt svarīgākos datoru organizācijas un funkcionēšanas aspektus no programmētāja viedokļa;
- dot priekšstatu par tipisku datorsistēmu ar *Intel* procesoru bāzi uzbūvi, uzmanību akcentējot uz principiem, kas ir kopēji dažādas arhitektūras skaitļotājiem;
- izskatīt būtiskāko datora arhitektūras komponentu īpatnības un to ietekmi uz programmas koda kvalitāti.

Adresāts

Mācību līdzeklis paredzēts Rēzeknes Augstskolas Inženieru fakultātes studiju programmas „Programmēšanas inženieris” studentiem. To var izmantot arī citu datorzinātņu studiju programmu studenti, kā arī visi interesenti, kas patstāvīgi vēlas iepazīties ar mūsdienu datoru arhitektūras pamatnostādņem.

Priekšlikumi materiāla sekmīgai apguvei

Mācību vielas sekmīgai apguvei nepieciešamas priekšzināšanas par datora uzbūvi un operētājsistēmu funkcionēšanu, jābūt iemaņām programmēšanā. Obligāts nosacījums ir piemēru izanalizēšana. Teorētisko zināšanu nostiprināšanai doti programmas koda piemēri zema līmeņa programmēšanas valodā *Assembler* vai *Microsoft Visual Studio 2005(2008)* (C++, C#) vidē, kā arī uzdevumi patstāvīgam darbam (katras nodaļas beigās).

Tekstā izmantotie apzīmējumi, kas atvieglo materiāla izpratni



– definīcijas, apzīmējumi, termini



– svarīgs skaidrojums vai rekomendācija



– piemēri, vingrinājumi



– piemēri, kurus ieteicams izpildīt ar datoru



– uzdevumi patstāvīgam darbam

Iegūstamās kompetences

- datoru arhitektūras koncepcijas apguve;
- datorsistēmu uzbūves un funkcionēšanas pārzināšana;
- mūsdienu procesoru organizācijas un iespēju izpratne;
- svarīgāko datora komponentu struktūras un funkciju pārzināšana;
- orientēšanās datorsistēmas ierīču funkcionēšanas programmas kodā.

APZĪMĒJUMU SKAIDROJUMS

AGP	(<i>Accelerated Graphics Port</i>) – paātrinātā grafikas pieslēgvietā
ALU	(<i>Arithmetic - Logical Unit</i>) – aritmētiski loģiskais bloks
ASCII	(<i>American Standard Code for Information Interchange</i>) – Informācijas apmaiņas standartkods
ATA	(<i>Advanced Technology Attachment</i>) – tehnoloģija ATA
ATAPI	(<i>AT Attachment Program Interface</i>) – tehnoloģijas AT piesaistes paketes saskarne
BIOS	(<i>Basic Input/Output System</i>) – ievadizvades pamatsistēma
CISC	(<i>Complete Instruction Set Computer</i>) – pilnas instrukciju kopas dators
CMP	(<i>Chip Multiprocessor</i>) – paralēlās darbības mikroprocesori
CPU	(<i>Central Processing Unit</i>) – centrālais procesors
DMA	(<i>Direct Memory Access</i>) – tiešās pieejas atmiņa
DSP	(<i>Digital Signal Processing</i>) – specializēti procesori
ECP	(<i>Extended Capabilities Port</i>) – paralēlās pieslēgvietas datu apmaiņas režīms
EIDE	(<i>Extended IDE</i>) – pilnveidots saskarnes IDE standarts
EPP	(<i>Enhanced Parallel Port</i>) – paralēlās pieslēgvietas datu apmaiņas režīms
EPROM	(<i>Electrically Programmable Read Only Memory</i>) – elektriski programmējama ROM
FPU	(<i>Floating Point Unit</i>) – matemātiskais līdzprocesors
GPP	(<i>General Purpose Processors</i>) – plaša lietojuma procesori
I/O	(<i>Input/Output</i>) – ievadizvade
IA-32	(<i>Intel Architecture, 32-bit</i>) – 32 bitu <i>Intel</i> arhitektūra
IDE	(<i>Integrated Drive Electronics</i>) – cieto disku diskdziņu saskarnes standarts
IRQ	(<i>Interrupt Request</i>) – pārtraukumu pieprasījumu mehānisms
L1	Pirmā līmeņa kešatmiņa
L2	Otrā līmeņa kešatmiņa
MMX	(<i>MultiMedia eXtension</i>) – SIMD instrukciju kopa multivides pielietojumiem
OS	(<i>Operating System</i>) – operētājsistēma
PCI	(<i>Peripheral Component Interconnect</i>) – ātrdarbīgas lokālās kopnes specifikācija
PCI-E	(<i>PCI Express</i>) – specifikācija PCI-E
PnP	(<i>Plug-and-Play</i>) – standarts PnP
RAM	(<i>Random Access Memory</i>) – brīvpieejas atmiņa jeb operatīvā atmiņa
RISC	(<i>Reduced Instruction Set Computer</i>) – samazinātas instrukciju kopas dators
ROM	(<i>Read Only Memory</i>) – pastāvīga lasāmatmiņa
RS-232	(<i>Recommended Standard 232</i>) – informācijas pārsūtīšanas virknes formātā standarts
SIMD	(<i>Single Instruction Multiple Data</i>) – viena instrukciju daudzkārtēju datu kopu apstrādē
SSE	(<i>Streaming SIMD Extensions</i>) – SIMD instrukciju kopa datu ar peldošo punktu apstrādei
SSE2	(<i>Streaming SIMD Extensions 2</i>) – instrukciju kopa 128 bitu datu ar peldošo punktu apstrādei
USB	(<i>Universal Serial Bus</i>) – universālā seriālā kopne
VLIW	(<i>Very Long Instruction Word</i>) – ļoti gara instrukcijas vārda arhitektūra
WinAPI	(<i>Windows Application Programming interface</i>) – lietojumprogrammu saskarne
x86	(<i>original Intel x86 processor architecture</i>) – saderība ar 8086 procesoru

1. DATORU ARHITEKTŪRAS KONCEPCIJA

Datortehnikai strauji attīstoties, nemainīgs ir palicis fundamentāls princips par to, ka datorsistēmu var traktēt kā līmeņu hierarhiju, kur katrs līmenis veic attiecīgu funkciju.

Mūsdienu datoru resursi ir tik jaudīgi, ka daudzas programmas strādās ar pieņemamu veikspēju, neatkarīgi no tā, cik labi tās ir uzrakstītas. Tomēr nopietnas lietojumprogrammas uzrakstīšana praktiski nav iespējama bez datora aparatūras funkcionēšanas īpatnību izmantošanas.

Datora aparatūras arhitektūras pārzināšana ļauj programmētājam izstrādāt daudz kvalitatīvāku programmnodrošinājumu, jo daudzas problēmas, kas rodas programmas izstrādes laikā, kļūst vieglāk atrisināmas.

Datora aparatūras daļas darbības principu izpratne dod iespēju:

- noteikt izstrādājamajā programmā „šaurās” vietas, kas pazemina tās darbības efektivitāti;
- izmantot datora arhitektūras īpatnības programmas veikspējas palielināšanā;
- dziļāk izprast atsevišķu datorsistēmas komponentu darbību.

1.1. Datoru arhitektūras jēdziens

Nespeciālistam vārds „arhitektūra” saistās ar celtniecību. Nepareizi būtu domāt, ka arhitektūra saistās tikai ar kādas ēkas ārējo izskatu, augstumu, interjeru.

Arhitektam jācenšas, lai ēka atbilstu arī savam funkcionālajam pielietojumam:

- būtu ērta ekspluatācijā;
- droša;
- ar pieņemamiem ekonomiskajiem rādītājiem utt.

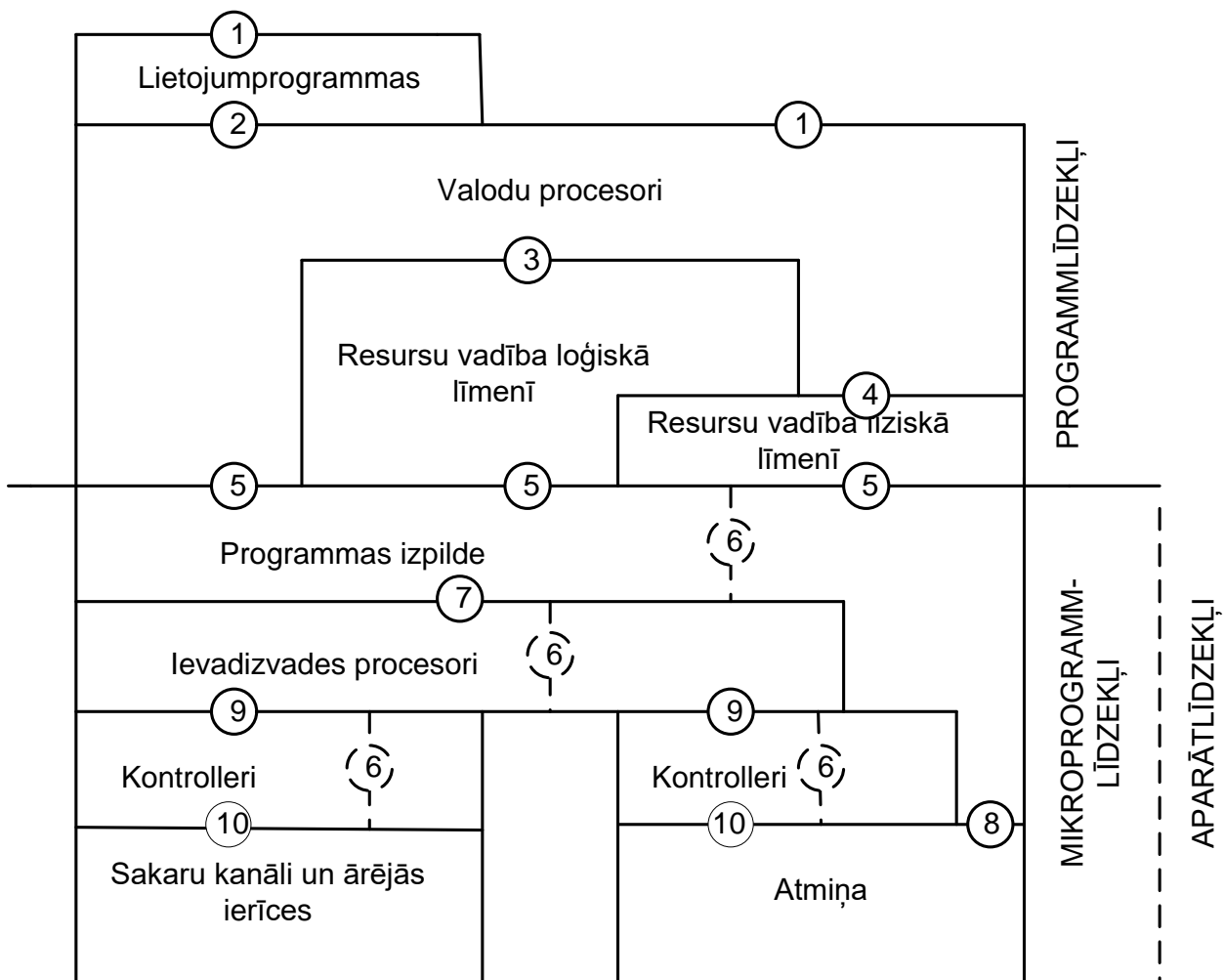
Arhitektūra ir viena no inženierijas formām.



Attiecībā uz skaitļošanas sistēmām terminu „arhitektūra” agrīnajā priekšstatā [10] definē kā realizējamās sistēmas funkciju sadalījumu pa atsevišķiem līmeņiem un konkrētu robežu noteikšana starp šiem līmeņiem.

Tātad, ja sistēmas arhitektūrai tiek piešķirts kaut kāds līmenis, tad vispirms jānosaka, kādas sistēmas funkcijas izpilda tā komponenti, kas ir augstāk jeb zemāk par uzdoto līmeni. Pēc šā uzdevuma veikšanas nākamais solis ir apskatāmā līmeņa saskarnes definēšana. Tādējādi skaitļošanas sistēmas arhitektūra ir daudzlīmeņu organizācija (sk. 1.1. att.).

1. līmenis - sistēmas arhitektūra.
- 2., 3., 4. - Dažādi programmnodrošinājuma līmeņi.
Programmnodrošinājuma arhitektūra.
5. - Robeža starp OS un aparatūras nodrošinājumu.
Svarīga daļa skaitļotāja arhitektūras izstrādē.
6. - Mikroprogrammu vadības arhitektūra.
7. - Nosaka, kādas funkcijas realizē procesors un kādas - ievadizvades procesori jeb kanāli.
8. - Saskaņot starp procesoru un atmiņu.
- 9., 10. - Kontrolleri tiek nodalīti no kanāliem.
7., 8., 9. - *Fiziskā ievadizvades arhitektūra.*
6., 8. - *Procesora arhitektūra jeb procesora organizācija.*



1.1.att. Agrīnais priekšstats par skaitļotāju arhitektūru

Termins „arhitektūra” pirmo reizi tika pielietots skaitļotāju konstrukcijā, izveidojot *IBM Stretch* skaitļotāju, un tas parādījās publikācijās 1962.gadā. Tas tika precizēts publikācijās, kas saistījās ar *System/360* un tagad tiek plaši pielietots.

Skaitļotāja arhitektūra - abstrakts priekšstats vai fiziskās sistēmas definējums no programmētāja viedokļa, kas izstrādā programnodrošinājumu vai kompilatorus.

⚠ Arhitektūra nosaka skaitļošanas sistēmas principus un procesora funkcijas un neapskata tādas problēmas kā datu vadība un pārraide procesora iekšpusē, loģisko shēmu konstruktīvās īpašības un to tehnoloģijas specifiku (*dažkārt tas kļūdaini tiek ietverts arhitektūras jēdzienā*).

Skaitļotāja arhitektam jāpieņem lēmumi par trim būtiskām lietām:

- programmas uzdošanas un interpretācijas veidu;
- datu adresācijas formu šajās programmās;
- datu uzdošanas paņēmieni.

Lai to paveiktu, ir jānosaka:

- minimāli adresējamās atmiņas apjoms;
- datu tipi un formāti;
- atmiņas adresācija un aizsardzība;
- komandu izpildes mehānisms;
- pārtraukumu mehānisms;
- saskarne ar ievadizvades ierīcēm.

Ideālā variantā skaitļotāja arhitektam jārisina šie uzdevumi šādi:

- nosacījumu analīze - programmēšanas valodu daudzums un specifika, perifērijas raksturojums, saskarne ar ārējo vidi, tehniski un ekonomiskie jautājumi;
- specifiku sastādīšana - projektējamās sistēmas parametru specifika - cena, darbietilpība, realizācijas grūtības pakāpe, iespēja paplašināt, saderība ar citām sistēmām;
- zināmo risinājumu pētīšana - salīdzināšana ar citām sistēmām (kas bieži tiek ignorēts);
- funkcionālskāmas sastādīšana - atbilstības starp dažādiem sistēmas līmeņiem noteikšana;
- struktūrskāmas sastādīšana - valodas semantika, datu uzdošana, adresācija, t.i., arhitektūras detalizācija;
- projekta lāgošana;
- projekta novērtēšana.

Skaitļotāju arhitektūras projektēšanai ir iteratīvs raksturs un parasti daudzkārt nākas atkārtot vienu vai visus etapus.

Līdzīgas prasības attiecas arī uz mūsdienu personālajiem datoriem, bet pirms vairāk nekā 40 gadiem tās bija jaunas un mainīja datoru arhitektūras attīstības virzienu. Daudzas no šīm prasībām netika ņemtas vērā personālo datoru izveidošanā, jo galvenā sākotnējā personālo datoru prasība bija tā, ka tiem jābūt ļoti lētiem. Līdz ar to šie datori atkārtoja sākotnējo lieldatoru arhitektūras trūkumus. Pamazām šie trūkumi tiek novērsti - pieaugot prasībām un krītoties tehnoloģiju cenām.

1. Iepriekšējie argumenti, ka katrai lietotāju klasei ir savdabīgi uzdevumi, kas attaisno specializētu datoru uzbūvi, kļūva vājāki. Dators daudz laika „velta” sistēmas kontroles darbam, kas ir kopējs visiem lietotājiem. Te ir ieskaitītas tādas funkcijas kā darbu izpildes plānošana, programmu izvietošana, ievada un izvada operāciju kontrole un kompilēšana. Šo darbu sadalījums tagad skaidri redzams mūsdienu personālajos datoros. Pirmkārt, ir tikai viena tipa personālie datori – nav specializētu datoru teksta apstrādei un finanšu aprēķiniem, un, otrkārt, dators daudz vairāk laika tērē dažādām *Windows* sistēmas funkcijām nekā, piemēram, apstrādājot lietotāja e-pastu vai dokumentu.
2. Bija vēlama vienvēidīga metode ievada un izvada ierīču pievienošanai un kontrolei, kas prasīja kopēju ierīču pievienošanas saskarni visām sistēmām, kā arī vienvēida programmas instrukcijas. Personālajiem datoriem līdzīga situācija tika atrisināta pavisam nesen, kad ieviesa universālo seriālo kopni, tā saukto USB. Tas ir kopējs savienojums visām jaunajām personālo datoru ievada un izvada ierīcēm.
3. Lai ļautu datoriem autonomi darboties un mazinātu prasības pēc operatora iejaukšanās, arhitektūrai vajadzēja dot iespēju datoram darboties vienlaicīgi ar vairākām programmām multiprogrammu režīmā un būt spējīgam atrast un reaģēt uz datorsistēmas kļūdām. Jaunās personālo datoru *Windows* versijas arvien vairāk gādā par to, lai dators nekad nebūtu jāaptur un par jaunu jāiedarbina (lieldatoru sistēmas tagad strādā mēnešiem ilgi bez neparedzētas apstāšanās).
4. Jaunajām sistēmām vajadzēja daudz lielāku operatīvo atmiņu un jaunajai arhitektūrai vajadzēja attiecīgas adresēšanas spējas, kas bija viena no galvenajām prasībām.
5. Programmatūras attīstīšanas izdevumi kļuva arvien lielāki attiecībā pret datoru uzbūvi – bija nepieciešama kopēja arhitektūra mazām un lielām sistēmām dažādām lietotāju klasēm, lai varētu dalīties ar ieguldījumu programmatūrā [21].

Arhitektūrai ir jāatbilst lietotāju vajadzībām, bet labai arhitektūrai ir vēl citas prasības. Ja tā ir komplicēta, mākslīga un nedabīga, to ir grūti saprast, iemācīties un atcerēties. To ir arī grūtāk iestrādāt datorā un programmēt.

- Arhitektūrai ir jābūt mērķtiecīgai un vienkāršai.

- Tai ir jābūt konsekventai ar izturētu likumsakarību. Tur nedrīkst būt pārsteigumu – ja daļa ir zināma, pārējais ir paredzams. Piemēram, kvadrātsaknes instrukcijai ir jāseko citu aritmētisko instrukciju likumībai un tās definēšanai nevajag jaunu izgudrojumu.
- Arhitektūrai ir jābūt elementārai un vispārējai. Ir dabīga tendence iestrādāt datorā visas funkcijas, ko prasa kāda plaši lietota programma. Bet vai prasības nemainīsies nākotnē? Labāk ir iestrādāt datorā elementāras darbības, ar kuru palīdzību var programmēt lietotāja uzdevumu. Arhitektam tomēr vienmēr ir jāizdomā, kā funkcija varētu būt programmējama un kādi izņēmuma gadījumi ir iespējami, lai elementāras darbības tiešām ļautu atrisināt uzdevumu.
- Taupības un ekonomijas nolūkā labā arhitektūrā katrai elementārai funkcijai ir jākalpo vairākiem mērķiem. No otras puses, katram uzdevumam drīkst būt tikai viens rīks. Ja ir vairāki savstarpēji konkurējoši rīki kāda uzdevuma veikšanai, uz kāda pamata lietotājs tos izvēlēsies un kuru konstruktors izveidos optimālu? Jāņem vērā, ka šāda izvēle bieži būs ieprogrammēta kompilatorā.
- Bieži arhitektam ir grūti izšķirties, tāpēc pastāv tendence iebūvēt arhitektūrā visas iespējas zem režīma kontroles. Tas var būt vēlami, ja prasības ir tiešām dažādas vai ir sagaidāms, ka tās mainīsies. Bet nedrīkst novelt izšķiršanos uz lietotāju, ja tam nebūs pietiekami informācijas izvēles pamatošanai. Režīma kontrole ir papildus funkcija, kas padara sistēmu komplicētāku un dārgāku.
- Arhitektūrai vienmēr ir jābūt paplašināmai, atstājot rezervē neizmantotus kodus un bitu pozīcijas, lai vēlāk varētu ieviest jaunas funkcijas un režīma kontroles. Turklāt datoram ir jāuzmana, lai programma neizmanto šos rezervētos laukus saviem nolūkiem, savādāk tāda programma nedarbosies pareizi datorā ar jauno funkciju [21].

Mūsdienās terminu „arhitektūra” bieži lieto kā šaurākā nozīmē, tā arī plašākā nozīmē.



Šaurākā nozīmē ar arhitektūru saprot instrukciju kopas arhitektūru. Tā ir robeža starp OS un datora aparatūru un nosaka to sistēmas daļu, kas ir paredzama programmētājam vai kompilatoru izstrādātājam. Jāatzīmē, ka tas ir biežāk pieņemtais termina „datoru arhitektūra” lietojums. Plašākā nozīmē arhitektūra ietver sevī sistēmas organizāciju – atmiņas organizāciju, sistēmas kopnes struktūru, ievadizvades organizāciju utt.

Divas pamata instrukciju kopas arhitektūras mūsdienās ir CISC un RISC arhitektūras.

CISC (*Complete Instruction Set Computer*) ir pilnas instrukciju kopas dators. Tā pamatlicējs ir kompānija *IBM* ar savu bāzes arhitektūru *System/360*, kuras kodols tiek izmantots no 1964.gada līdz pat mūsdienām (lieldatoros *IBM ES/9000*). Līderis mikroprocesoru izstrādē ar CISC arhitektūru ir kompānija *Intel* ar savu sēriju *x86*. Šī arhitektūra ir kļuvusi par *de facto* standartu mūsdienu mikroprocesoru tirgū. CISC procesoriem ir raksturīgs:

- relatīvi neliels kopējās nozīmes reģistru skaits;
- liels mašīnkomandu skaits;
- liels adresācijas metožu skaits;
- liels komandu formātu skaits (pārsvarā divadrešu komandas);
- reģistru-atmiņas tipa komandas.

Darbstaciju un serveru platformas pamatā ir samazinātas instrukciju kopas datori - RISC (*Reduced Instruction Set Computer*). Šīs arhitektūras pirmsākumi ir atrodamī skaitļotājos *CDC6600*, kuru izstrādātāji izprata īsu komandu nozīmi ātru skaitļotāju uzbūvē. Tradīcija izmantot samazinātas instrukcijas tika turpinātas arī kompānijas *Cray research* superdatoros. RISC procesoros galvenā ideja bija nodalīt relatīvi lēno atmiņu no ātrdarbīgajiem reģistriem un veikt efektīvu komandu konveijerapstrādi. Komandu sistēma tika izstrādāta tādā veidā, lai jebkuras komandas izpilde aizņemtu pēc iespējas mazāk darba taktis. Pati komandu izpildes loģika tika orientēta nevis uz mikroprogrammu realizāciju, bet balstīta uz aparatūras realizāciju. Starp citām

RISC arhitektūras īpatnībām var atzīmēt lielu reģistru skaitu (tipveida RISC datoros izmanto 32 un vairāk reģistru pretstatā 8-16 reģistriem CISC arhitektūrā), kas dod iespēju uzglabāt lielākus datu apjomus reģistros.

1.2. Neimana arhitektūra un tradicionālās arhitektūras uzbūve

Neimana arhitektūra ir 1946. gadā matemātiķa Dž. fon Neimana (*John von Neumann*) un viņa kolēģu H. Goldšteina (*Goldstine H.H*) un A. Berksa (*Burks A.W*) formulētā un mūsdienās datortehnikā vispārpieņemtā bloku darbības principu un to mijiedarbes kārtība. Šīs arhitektūras galvenā raksturiezīme ir tā, ka lineāri adresējama atmiņa ir kopēja programmām un datiem un ka vienlaicīgi var izpildīt tikai vienu procesu, t. i., instrukcijas tiek secīgi analizētas un attiecīgi apstrādātas.

Pirmās skaitļojamās mašīnas *ENIAC* (un vēlāk *EDVAC*) projekts ieinteresēja Džonu fon Neimānu, un viņš kā matemātikas konsultants sāka izstrādāt loģisko shēmu, kurā būtu iespējams izmantot tādu atmiņā ierakstīto programmu, ko varētu viegli mainīt, nepārbūvējot visu mašīnas shēmu. Šīs idejas tika izklāstītas 1945. gadā publicētajā rakstā "Sākotnējais ziņojums par mašīnu *EDVAC*", kurā viņš aprakstīja mašīnu un tās loģiskās īpašības. Tajā bija izteikti divi pamatprincipi, kas ieguva praktisku lietojumu visās mūsdienu elektronu skaitļojamās mašīnās: pamatota binārās skaitīšanas sistēmas izmantošana skaitļu attēlošanā un atmiņā glabājamā programma. Atmiņā uzglabājamā programma ļāva pārvarēt būtiskāko *ENIAC* trūkumu – laika patēriņu programmas salikšanai un sagatavošanai uz komutāciju paneļa. Programmu, tāpat kā sākuma skaitļus, viņš piedāvāja glabāt mašīnas atmiņā. Ar vadības iekārtu atsevišķas komandas tika izsauktas no atmiņas nepieciešamo darbību izpildei un rezultāta nosūtīšanai atmiņā. Raksts saturēja arī svarīgas rekomendācijas mašīnas konstruēšanai un programmēšanas metodikai. Kā izriet no ziņojumiem par šo tēmu, 1946. gadā Džons fon Neimāns pirmo reizi skaitļojamās mašīnas shēmā izdalīja 4 galvenos blokus un nosauca tos par aritmētiski loģisko iekārtu, atmiņu datiem un komandām, vadības iekārtu, ievadizvades iekārtu [10,20]. Neimana aprakstītās idejas vēlāk ieguva viņa vārdu, tādējādi, viņam tika piedēvēta visa projekta autorība, kas turpmāk izraisīja pat tiesvedību ar partneriem par patentiem.

Neimana izstrādātās idejās ir šādas:

- aritmētiski loģiskā iekārta glabā matemātiskās un loģiskās operācijas;
- iekārta ievada datus skaitļotājā;
- kontroles iekārta vada skaitļotāja pavēļu (komandu jeb programmas) secību;
- iekšējā (primārā jeb operatīvā) atmiņa glabā datus un programmas;
- ievades iekārta ievada datus skaitļotājā;
- izvades iekārta izvada datus no skaitļotāja;
- ārējā (sekundārā) atmiņā dati un programmas tiek glabāti patstāvīgi.



Mūsdienu traktējumā Neimana principi ir četri [14]:

Binārās kodēšanas princips

Visa informācija – kā dati, tā arī programmas – tiek kodētas ar bināriem cipariem 0 un 1. Katram informācijas tipam ir sava bitu konsekvence jeb formāts.

Programmējamās vadības princips

Algoritms tiek pasniegts programmas veidā, kas sastāv no atsevišķām komandām, ko realizē dators. Programmas komandas datora atmiņā glabājas konsekvētās šūnās un tiek izpildītas to dabiskajā secībā (t.i., viena pēc otras). Programmas komandu izpildes secības maiņai pastāv speciālas beznosacījuma vai nosacījuma zarojuma komandas.

Atmiņas vienveidības princips

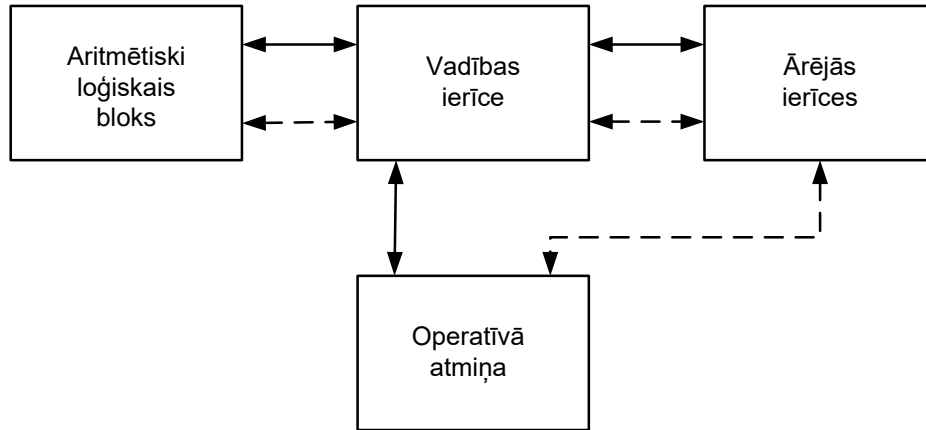
Komandas un dati glabājas vienotā atmiņā un ārēji nav atšķirami. Tas ļauj ar instrukcijām veikt tās pašas darbības kā ar datiem.

Adresēšanas princips

Atmiņas pamatā ir šūnu masīvs ar unikāli numurētām šūnām. Procesoram jebkurā brīdī ir pieejama jebkura atmiņas šūna, bet, lai tai piekļūtu, izmanto atbilstošo šūnas numuru jeb adresi.

1.2.attēlā parādīts, kādām vajadzētu būt saitēm starp datora iekārtām saskaņā ar Neimana principiem (ordinārās līnijas parāda vadības saites, punktētās – informatīvās saites).

Praktiski visas Neimana idejas turpmāk tika pielietotas pirmo triju paaudžu skaitļotāju konstruēšanā.

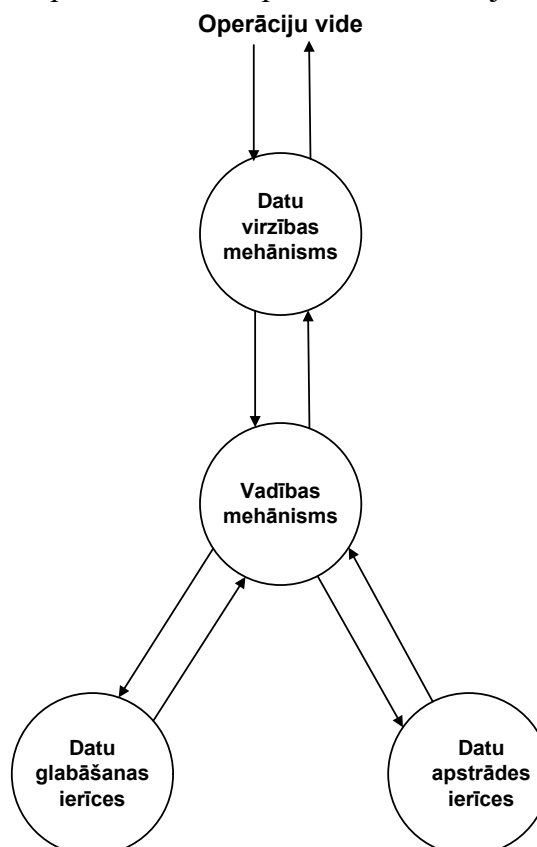


1.2. att. Saites starp datora ierīcēm (pēc Neimana arhitektūras principiem)

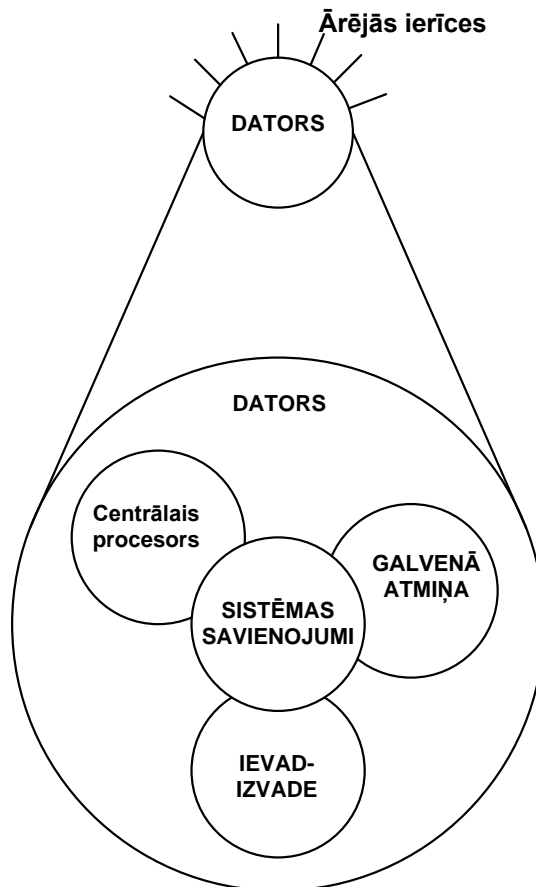
Mūsdienu skatījumā datoru arhitektūrā mēdz izdalīt dažādu arhitektūras līmeņu struktūru un funkcijas [7]:

- struktūra – veids, kādā tiek savienotas datora komponentes;
- funkcijas – operācijas katrā komponentē kā daļa no struktūras.

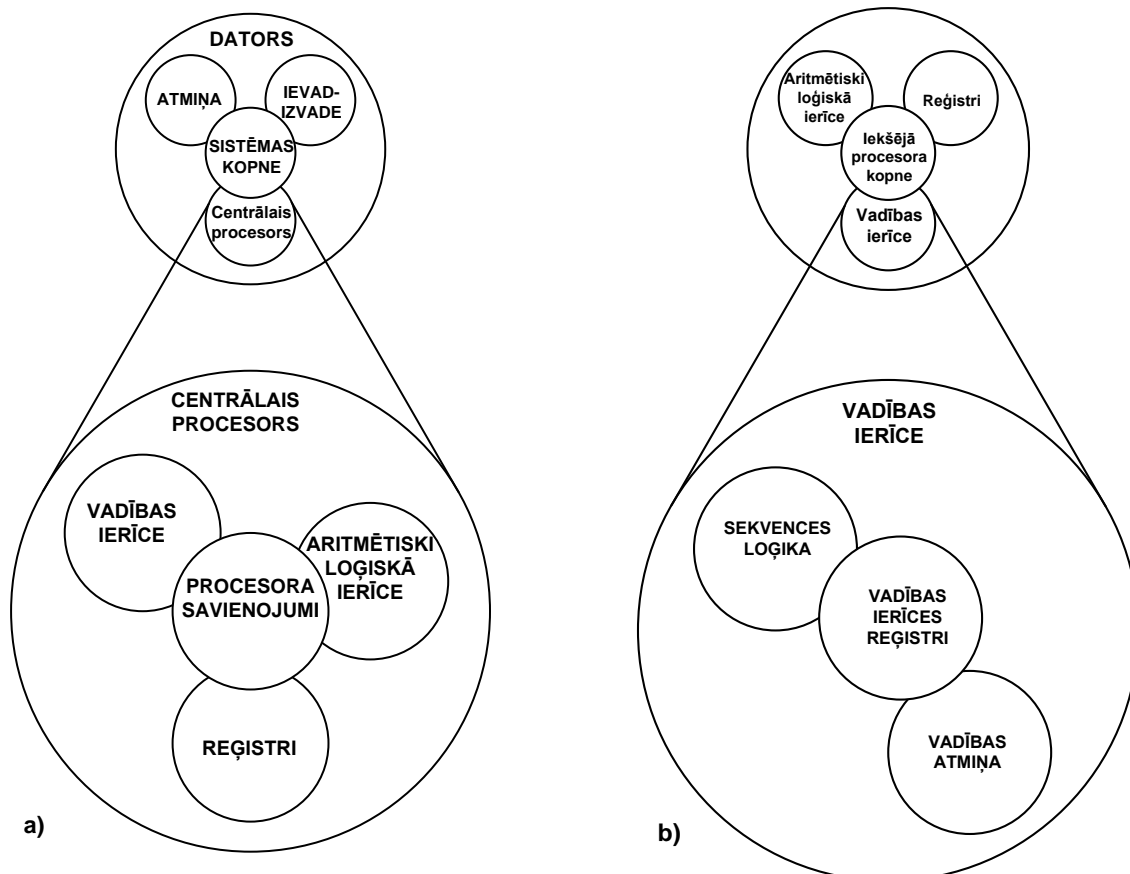
1.3. – 1.6. attēlos parādīts datorsistēmas funkcionālais skats un atsevišķu komponentu struktūra [7] (sīkāk saites starp komponentiem tiks aprakstītas nākamajās nodaļās).



1.3.att. Datorsistēmas funkcionālais skats



1.4.att. Datorsistēmas augšējā līmeņa struktūra



1.5.att. CPU struktūra (a) un vadības ierīces struktūra (b)

1.3. Galveno jēdzienu skaidrojums

i **Centrālais procesors** (*Central Processing Unit, CPU*), bieži vien saukts vienkārši par procesoru, ir procesors, kas datu apstrādes gaitā veic instrukciju interpretāciju un vada citu datora bloku darbību. Centrālais procesors sastāv no vadības bloka, aritmētiski loģiskā bloka un atmiņas. Personālajos datoros centrālo procesoru parasti veido, izmantojot vienu mikroprocesoru.

Svarīgākie centrālo procesoru raksturojošie lielumi ir takts frekvence, konveijera dziļums, kešatmiņas apjoms, tranzistoru izmērs, kristāla virsmas izmērs, ligzdas tips un atbalstītās tehnoloģijas.

Takts frekvence, ko mēra hercos, tika uzskatīta par visnozīmīgāko rādītāju, tomēr mūsdienās izrādās, ka svarīgākais ir pareiza proporcija visos rādītājos.

Konveijera dziļums nozīmē, cik procesi var būt vienlaicīgi apstrādē, protams, katrs savā gatavības formā. Šeit ir problēma, ka pārāk dziļš konveijers ir par lielu un daudzas taktis tiek laistas pa tukšo (ja programmā ir sazarojums, konveijeru nākas iztukšot (pabeigt visus iesāktos procesus) un ja konveijers ir garāks, stipri sazarotām programmām tas ir neefektīvāk).

L1 ir *kešatmiņa*, kas noder, lai ātri piegādātu atkārtoti nepieciešamo informāciju. L1 ir pirmā līmeņa kešatmiņa, kas parasti strādā ar procesora kodola frekvenci, L2 ir otrā līmeņa kešatmiņa, un tā var būt lēnāka.

i CISC – procesori

- Neimana arhitektūra
- Asamblera programmēšana
- Liels instrukciju skaits
- Datoru saimes
- Kompilatoru izmantošana
- Lietojums: universālie mikrodatori

i **Datora arhitektūra** ir datorsistēmas konceptuālais modelis, tā darbības struktūra un resursi, kas pieejami programmētājam. Tā nosaka datora darbības principus, informatīvās saites un pamat mezglu savstarpējo saistību.

Datora arhitektūra tiek iedalīta vismaz trīs apakškategoriās.

- Instrukciju kopas arhitektūra ir datora abstraktais modelis no programmatūras (mašīnkoda) viedokļa, kas ietver instrukciju kopu, atmiņas adresēšanas veidus, procesora reģistrus un datu formātus.
- Mikroarhitektūra ir datorsistēmas organizācijas zemākais līmenis. Tā nosaka, kā konkrēta modeļa sistēmas daļas sadarbojas savā starpā saskaņā ar instrukciju kopas arhitektūru.
- Sistēmas aparatūras modelis, kas ietver visus citus aparatūras komponentus datorsistēmā.

i **Dators** (*data processor* – datu apstrādātājs) ir iekārta, kas apstrādā datus pēc iepriekš definētas procedūras. Datori ir konstruēti no komponentiem, kas veic vienkāršas, iepriekš noteiktas darbības. Šo komponentu savstarpējā saziņa ļauj datoram veikt informācijas apstrādes procesus. Pareizi sakonfigurēts (ieprogrammēts) dators spēj risināt noteiktas problēmas, kā arī būt lielākas sistēmas sastāvdaļa. Pareizi sakonfigurētam datoram padodot ievaddatus, programma tos apstrādā, un dators spēj atrisināt problēmu vai prognozēt sistēmas uzvedību.

i Draiveris /dzinis (*driver*)

Programma vai speciāla ierīce, kas vada dažādu procesu norisi vai ierīču darbību (piemēram, signālu pārsūtīšanu pa sakaru līnijām, diskdziņu, printeru u.c. ierīču funkcionēšanu), lai saskaņotu šo procesu izpildi vai ierīču sadarbību ar datoru.

i **Instrukciju kopa** ir visu instrukciju (komandu) kopums, kuras var izpildīt centrālais procesors.

Instrukcijas pēc funkcionalitātes tiek iedalītas:

- o aritmētiskās (saskaitīšana, atņemšana),
- o loģiskās (UN, VAI, NE),
- o nobīdes,
- o datu manipulācijas (pārsūtīšana, ievadizvade)
- o vadības (tiešā, netiešā pāreja, apakšprogrammas izsaukšana u.c.),
- o citas.

Instrukcijas formāts nosaka komandas elementus, kuri komandas izpildes laikā jāinterpretē noteiktā veidā. Šie elementi var būt: instrukcijas kods, atmiņas adrese, procesora reģistrs, ārējās ierīces adrese, adresācijas režīms, pārejas parametri, u.c. Komandas kods nosaka to, cik un kādi operandi būs instrukcijai. Atkarībā no instrukciju kopas arhitektūras, to var noteikt atsevišķi biti instrukcijas kodā vai arī viss kods. Pēc parametru skaita instrukcijas iedala: bez operandu, ar vienu vai ar diviem operandiem. Pēc instrukcijas garuma var būt 1, 2, 3 un vairāku baitu komandas.

Palielinot instrukciju garumu no 16 uz 32 bitiem, izdodas izvairīties no ierobežojumiem, piemēram, kuri no reģistriem tiek izmantoti operācijām. Tas tāpēc, ka, jo garāks instrukcijas vārds, jo vairāk atšķirīgu instrukciju var apzīmēt, tā pieļaujot instrukciju kopai veidot vairāk kombināciju izpildāmo operāciju un reģistru starpā.

i **Kešatmiņa** (*cache memory*)

Atmiņa, kurai piekļuves laiks ir ievērojami mazāks nekā operatīvajai atmiņai. Kešatmiņu izmanto kā buferatmiņu starp procesoru un operatīvo atmiņu.

i **Kontrolleris** (*controller*)

Ierīce, kas vada datu apmaiņu starp datora centrālo procesoru un citiem tā funkcionālajiem blokiem (atmiņu, displeju, tastatūru, printeri u. c. ārējām ierīcēm), atbrīvojot no šī uzdevuma izpildes centrālo procesoru. Personālajos datoros kontrolleri parasti ir to standarta komponentos (displejos, tastatūrā, diskdziņos u. c.) vai izvēršanas platēs iebūvētas mikroshēmas. Tiem jānodrošina datu apmaiņa starp datora kopnēm.

i **Kopne / maģistrāle** (*bus*)

Elektrisko signālu vadītāju (vadu) komplekts, pa kuru notiek informācijas apmaiņa starp dažādiem datora funkcionālajiem blokiem (procesoriem, atmiņu, pieslēgvietām, ārējo iekārtu vadības ierīcēm u.c.). Personālajos datoros parasti izmanto standartizētās kopnes, ko veido trīs dažādu kopņu apvienojums. Pa tām tiek pārsūtīti dati (datu kopne), informācija par datu atrašanās vietu (adrešu kopne) un vadības informācija (vadības kopne). Kopnes galvenokārt raksturo ar bitu skaitu, ko vienlaicīgi var pārsūtīt pa kopni (izšķir 8, 16, 32 utt. bitu kopnes), un ar maksimālo datu ātrumu, ko norāda MHz.

i **Mātesplate** (*mother board*)

Mikrodatora montāžas plate, kas veido mikrodatora pamatstruktūru. Tajā ir centrālais procesors, operatīvā atmiņa, virknes un paralēlās pieslēgvietas dažādu ārējo iekārtu (piemēram, displeja, tastatūras un disku) vadībai. Ja mikroshēma, kas vada displeju, virknes un paralēlās pieslēgvietas, peli un diskdziņus, neatrodas uz mātesplates, tad tos vada autonomas vadības iekārtas, kuras pievieno mātesplates izvērsuma platei.

i **Mikroarhitektūra** ir datoru arhitektūras iekšējās realizācijas (aparātūras līmenī) apraksts (bieži lieto arī sinonīmu *datora organizācija*).

Dažādiem datoriem var būt viena instrukciju kopas arhitektūra, līdz ar to tie var izpildīt vienas un tās pašas programmas, pat ar dažādām mikroarhitektūrām. Šīs dažādās mikroarhitektūras ir tas, kas ļauj procesoru jaunajām paaudzēm sasniegt augstāku ražību, salīdzinot ar iepriekšējām paaudzēm.

Reālo elektronisko shēmu izvietojumu, aparātūras konstrukciju un citas fiziskās detaļas sauc par kādas mikroarhitektūras *īstenošanu* (*implementation*). Divas mašīnas var būt ar vienu mikroarhitektūru, bet ar atšķirīgu aparātūras īstenošanu.

Daudzi skaitļotāji no 20.gs. 50. - 70.gadiem izmantoja mikroprogrammēšanu, lai īstenotu to kontroles loģiku, kura dekodēja programmas instrukcijas un izpildīja tās. Mikroprogrammas vārdu biti bija elektriskie signāli, kuri kontrolēja ierīces, kuras faktiski veica skaitļošanu. Termins "mikroarhitektūra" tika lietots, lai aprakstītu šīs ierīces, kuras vadīja mikroprogrammas vārdi.

i **Mikroprocesors**

Ar mikroprocesoru saprot ar programmu vadāmu mikroshēmu, kas paredzēta digitālās informācijas apstrādei un kas pēdējā laikā tiek būvēta, tāpat kā pirmais mikroprocesors, tikai uz vienas mikroshēmas (*chip*). Mikroprocesors veic divas svarīgas lietas: programmējamu datu apstrādi un šī procesa vadību, tādējādi kļūstot par datora galveno sastāvdaļu. Mikroprocesors sastāv no vadības bloka (kontrollera), ALU (*Arithmetic - Logical Unit*, aritmētiski loģiskā bloka) un atmiņas. Visi bloki savstarpēji saistīti ar kopnēm. Iekšējā atmiņa sastāv no programmu (instrukciju) un datu operatīvās RAM (*Random Access Memory*) brīvpieejas un ROM (*Read Only Memory*) energoneatkarīgas, pastāvīgas lasāmatmiņas. Iekšējā atmiņa nosaka ātru instrukciju un datu pārvades izpildi. Lai palielinātu atmiņas apjomu, kopnes tiek pievienotas ārējai atmiņai (RAM, ROM, EPROM – *Electrically Programmable Read Only Memory* – elektriski programmējama ROM). Ja mikroprocesorā ir *peldošā punkta* procesors FPU (*floating point unit*), tas vēl satur DMA (*Direct Memory Access*) bloku, kešatmiņu un komplicētāku kopņu sistēmu.

Mikroprocesorus var iedalīt divās lielās grupās – plaša lietojuma, universālajos (GBP – *General Purpose Processors*) un konkrētam uzdevumam specializētos procesoros, ko pēdējā laikā dēvē par DSP (*Digital Signal Processing*). Pirmais DSP procesors bija 1980. gadā japāņu firmas NEC izstrādātais NEC 7720, kas bija paredzēts telekomunikāciju lietojumos.

i **Mikroshēmojums** (*chipset*)

Mikroshēmu grupa, kuras var kopīgi izmantot kādas funkcijas īstenošanai, tādēļ tās apvienotas un tiek traktētas kā vienots bloks.

i **Neimana/Hārvarda arhitektūra**

Neimana arhitektūra ir 1946. gadā ungāru matemātiķa Dž. fon Neimana un viņa kolēģu G. Goldsmīta un A. Berksa formulētā un mūsdienās datortehnikā vispārpieņemtā bloku darbības principu un to mijiedarbes kārtība. Hārvarda arhitektūru savukārt iedibināja Hārvarda universitātes līdzstrādnieku grupa, kas ieteica mikroprocesoru modeli nevis ar kopējām kopnēm, bet datiem un instrukcijām nodalītām, atsevišķām kopnēm.

i **Paralēlisms**

Pēdējā laikā uz mikroshēmas izvieto vairākus mikroprocesorus, kas *sajūgti* paralēlai darbībai. Tā ir CMP (*Chip Multiprocessor*) arhitektūra. Starpprocesoru informācijas apmaiņu uz vienas mikroshēmas var realizēt daudz ātrāk nekā ar blokiem ārpus mikroshēmas. CMP iedibināja firma IBM 2000. gadā. CMP izmanto gan GPP, gan DSP procesoros. Taču pastāv būtiskas atšķirības tajā, kā CMP šajos mikroprocesoros tiek realizētas.

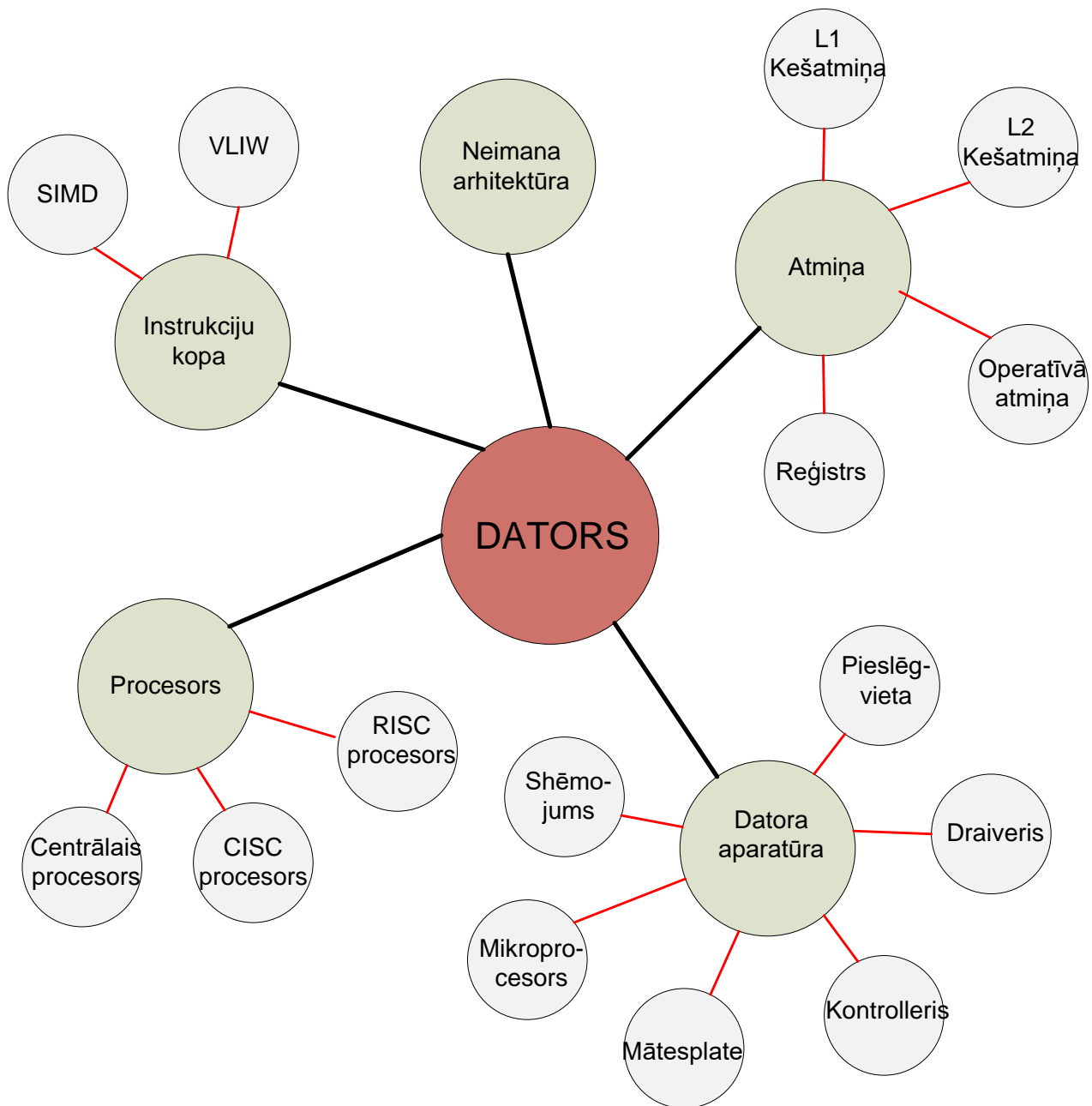
<p>i Pieslēgvieta /ports (<i>port</i>) Fizikāls savienojums, ar kura starpniecību sinhronizē un vada datu plūsmu starp centrālo procesoru un tādām ārējām iekārtām kā, piemēram, modemiem un printeriem.</p>
<p>i Reģistrs (<i>register</i>) Maza apjoma atmiņa, kas paredzēta datu īslaicīgai glabāšanai. Reģistra ietilpība parasti ir viens bits, baits vai vārds.</p>
<p>i RISC – procesori</p> <ul style="list-style-type: none"> ○ Neimana/Hārvarda arhitektūra ○ Instrukciju skaits <100 ○ Instrukciju formāti <4 ○ Aparatūrā ierakstīta instrukcija (neizmanto mikroinstrukcijas) ○ Vairums instrukciju apstrādā vienā darba taktī ○ Optimizēti kompilatori augsta līmeņa valodām ○ Lietojums: darbstacijas un serveri
<p>i Rokspiešanas princips (<i>handshaking</i>) Vairāku datu apstrādes ierīču darbības sinhronizācijas procedūra, kuras rezultātā tiek iegūts apstiprinājums, ka starp šīm iekārtām nodibināti vai pārtraukti sakari.</p>
<p>i SIMD Cits ceļš uz paralēlismu ir SIMD (<i>Single Instruction Multiple Data</i>) metode, kas procesoram pieļauj izpildīt to pašu operāciju, lietojot vienu instrukciju daudzkārtēju, neatkarīgu datu kopu apstrādē. Mikroprocesori ar SIMD atbalstu var apstrādāt datus, piemēram, 64 bitu reģistrā kā četrkārt mazāku datu, t.i., 16 bitu, vārdu. Turklāt tiek veiktas tās pašas operācijas, bet izejā tiek ģenerēti neatkarīgi lielumi. SIMD arhitektūra kļuva populāra pagājušā gadsimta 90.gados vektororientētiem algoritmiem (piemēram, <i>MPEG2</i>, <i>MPEG4</i>), lai paaugstinātu centrālā procesora veiktspēju multivides un videosignālu apstrādes lietojumos. SIMD bija tas, kas nodrošināja GPP procesoru (<i>Intel Pentium III</i>, <i>Motorola PowerPC G4</i>) līdzināšanos vai pārkumu ātrdarbības ziņā pār DSP procesoriem. Kaut arī sākotnēji paredzēta tikai GPP procesoriem, SIMD vēlāk plaši izmantota jaunākajās DSP izstrādēs.</p>
<p>i VLIW un superskalārā arhitektūra DSP procesoros izmanto ļoti gara instrukcijas vārda arhitektūru – VLIW (<i>Very Long Instruction Word</i>). Ar VLIW apzīmē instrukciju, ko var sagrupēt paralēlai darbībai. Piemēram, firmas <i>Texas Instruments</i> MP <i>TMS 320C6XXX</i> lieto VLIW, kas sastāv no astoņiem 32 bitu vārdiem. Kopā sanāk 256 bitu instrukcija. GPP procesoros paralēlismam izmanto t.s. superskalāro (<i>superscalar</i>) arhitektūru. Abas arhitektūras atšķiras galvenokārt ar to, kā instrukcijas paralēlai izpildei tiek grupētas.</p>

NODAĻAS KOPSAVILKUMS

- Datora arhitektūra ir datorsistēmas konceptuālais modelis, tā darbības struktūra un resursi, kas pieejami programmētājam. Arhitektūra nosaka datora darbības principus, informatīvās saites un pamat mezglu savstarpējo saistību.
- Datora arhitektūra ir daudzlīmeņu organizācija.
- Datoru arhitektūras projektēšanai ir iteratīvs raksturs.
- Šaurākā nozīmē ar arhitektūru saprot instrukciju kopas arhitektūru, plašākā - arhitektūra ietver sevī sistēmas organizāciju.

- Divas pamata instrukciju kopas arhitektūras mūsdienās ir CISC un RISC arhitektūras.
- Neimana datoru arhitektūras principi ir šādi:
 - binārās kodēšanas princips;
 - programmējamās vadības princips
 - atmiņas vienvēidības princips.
 - adresēšanas princips.

1.6.attēlā parādīts pirmajā nodaļā minēto svarīgāko jēdzienu koks.



1.6. att. Pirmajā nodaļā minēto jēdzienu koks

Uzdevumi un jautājumi patstāvīgajam darbam

1. Raksturot Neimana arhitektūras principus.
2. Salīdzināt datoru arhitektūras agrīno un mūsdienu priekšstatus.
3. Vai ir lietderīgi noteikt centrālā procesora arhitektūru ātrāk nekā datorsistēmas kopējo arhitektūru?
4. Kurš no kandidātiem vislabāk derētu par jaunas datorsistēmas arhitektūras izstrādātāju:
 - a) programmētājs;
 - b) kompilatoru izstrādātājs;
 - c) OS izstrādātājs;
 - d) inženieris, kas projektē centrālo procesoru.
5. Kas ir primārs: *hardware* vai *software*?

2. IEVADIZVADES ARHITEKTŪRA UN ATMIŅAS ORGANIZĀCIJA

Nodaļā analizēta datu apmaiņas būtības organizācija datorsistēmās. Tās pamatā ir ārējo ierīču informācijas ievadizvade, kā arī datu ierakstīšana un nolasīšana ar operatīvās atmiņas palīdzību.

Ja programmētājam jāuzraksta programma sistēmas līmenī (sistēmas dienestu vai draiveri), tad bez nopietnām aparatūras īpatnību zināšanām to izdarīt nav iespējams. Turklāt ievadizvades un operatīvās atmiņas funkcionēšanas principi ļauj labāk saprast datora ierīču darbu.

2.1. Ievadizvades arhitektūra

Kā tika noskaidrots iepriekšējā nodaļā, jebkuru datorsistēmu (neatkarīgi no arhitektūras) var uzdot kā vairāku apakšsistēmu kopumu:

- procesors (viens vai vairāki);
- operatīvā atmiņa;
- informācijas uzglabāšanas ierīces;
- lietotāja saskarnes ierīces.

Tāda tipa arhitektūrā visas datorsistēmas iekārtas savā starpā saistītas ar vienas vai vairāku kopņu (*bus*) palīdzību.



No elektronisko shēmu viedokļa kopne ir elektrisko signālu kopums, kas tiek pārraidīti un sinhronizēti noteiktā secībā.

Centrālais procesors apstrādā informāciju no dažādiem avotiem (atmiņa, cietie diski, ārējās ierīces utt.). Visas šīs ierīces atšķiras viena no otras un prasa dažādus datu apmaiņas ātrumus, dažādas sinhronizējošo signālu virknes, kas stipri sarežģī datora konfigurāciju. No šī viedokļa viena vai vairāku universālu ar aparatūru realizējamu saskarņu (kā, piemēram, PCI vai AGP kopnes) klātbūtne būtiski vienkāršo datorsistēmas komponentu mijiedarbību, ļaujot izstrādātājiem ātri ieviest jaunus tehniskus risinājumus.

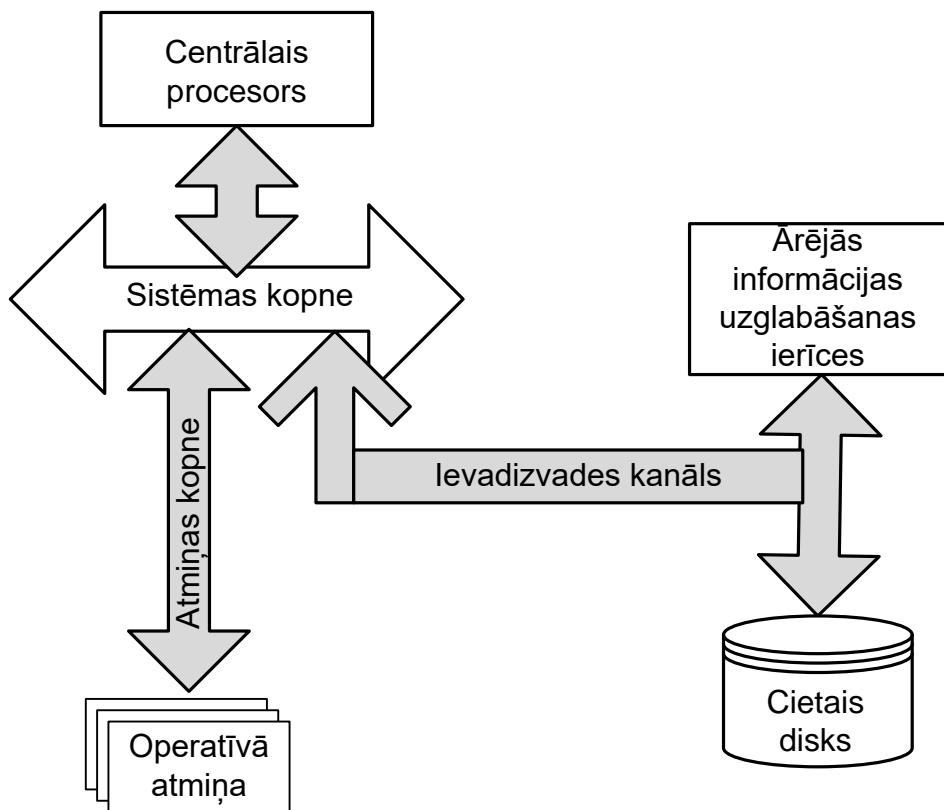


Šajā kontekstā datu apmaiņas organizācijai starp „procesoru – atmiņu” un „procesoru – ievadizvades ierīcēm” ir savas specifiskas īpatnības, jo tā ir speciālu aparatūras līdzekļu prerogātīva, kas ieguva nosaukumu *mikroshēmojums (chipset)*.

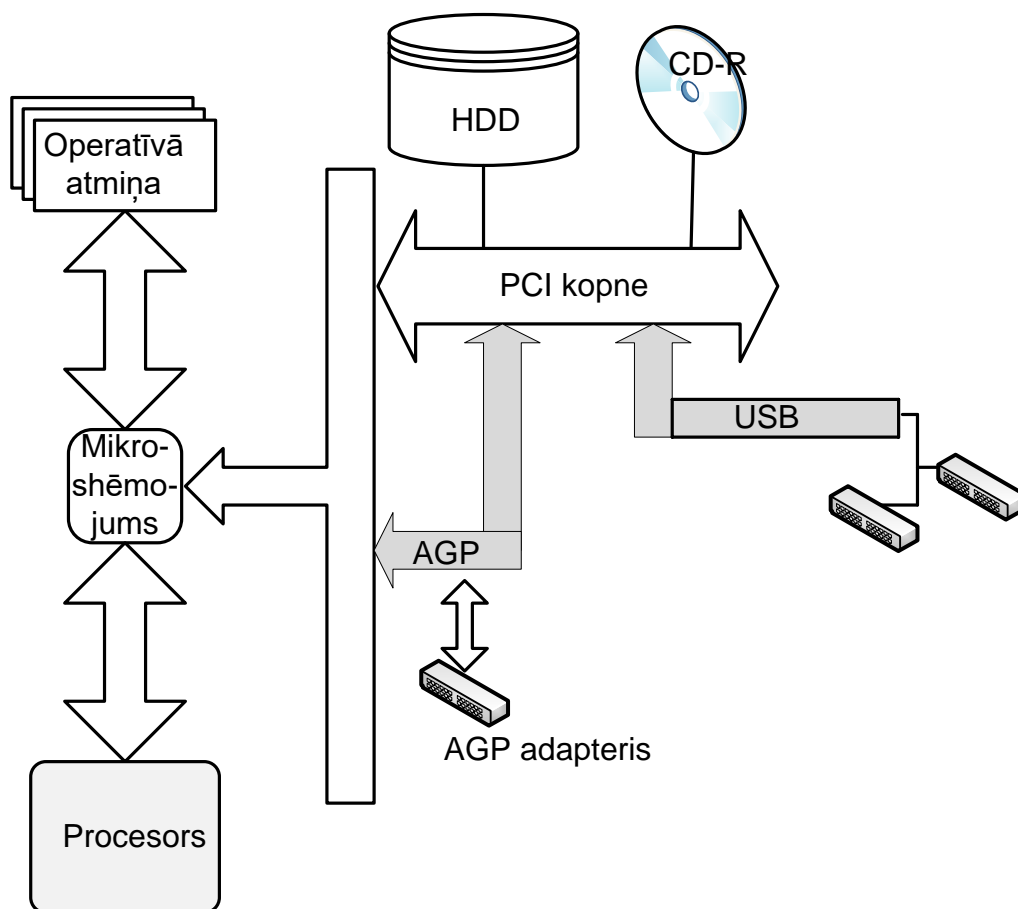
Klasiskajā sistēmas x86 arhitektūrā, kas attēlota 2.1. attēlā, procesors vada atmiņu un ievadizvades ierīces (*input/output - I/O*) pa tiešo, izmantojot saskarnes signālus, ko pieņemts saukt par sistēmas kopni.

Dator tehnoloģiju ēras sākumā tā arī notika: praktiskā ziņā atmiņas un I/O ierīču vadībai bija nepieciešams tikai uzstādīt pieņemšanas-nodošanas bufera mikroshēmas, kas nodrošināja nepieciešamo vadības signālu līmeni. Turpmāk tika izstrādāti kopnes kontrolleru mikroshēmas, kas mijiedarbojās ar procesora sistēmas kopnes signāliem un nodrošināja ērtu un universālu saskarni ar atmiņu un I/O ierīcēm.

Datora perifērijas ierīcēm attīstoties un paliekot arvien sarežģītākām, apmaiņas operāciju standartizācijas nolūkos tika izstrādāta aparatūras saskarnes specifikācija, kas ļāva vienam interfeisam pieslēgt daudzas ierīces. Patlaban plaši izplatītas kopnes ir PCI (*Peripheral Component Interconnect*), AGP (*Accelerated Graphics Port*), USB. Strauji tiek ieviests PCI-E (*PCI Express*) standarts [4]. Lai nodrošinātu ierīču darbu, kas pieslēgtas kopnēm, tika izstrādāti arī mikroshēmojumi. Vispārīgā mikroshēmojuma pieslēgšanas shēma datorsistēmās attēlota 2.2 attēlā.



2.1.att. Datu apmaiņas shēma datorā

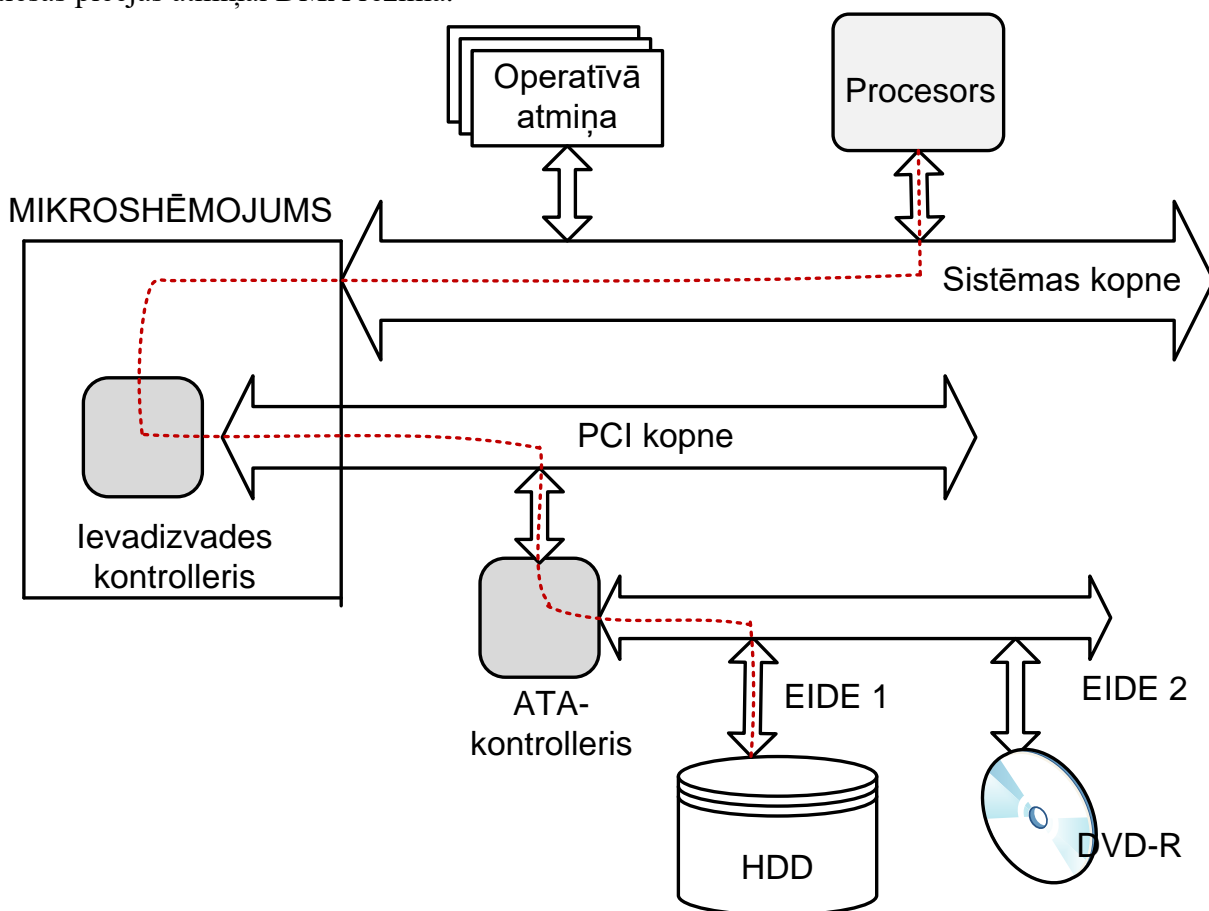


2.2.att. Mikroshēmojuma pieslēgšana datorsistēmai


⚠️ No praktiskā viedokļa mikroshēmojums sastāv no funkcionāli saistītām mikroshēmām, kuru sastāvā ietilpst I/O kontrolleri, AGP, ATA (*AT Attachment*), USB, audio u.c. Mikroshēmojuma kontrolleri izpilda dažādas funkcijas, bet fiziski var atrasties uz viena vai vairākiem kristāliem.


Mikroshēmojuma kontrolleri var mijiedarboties gan ar datorsistēmas iekārtām, gan arī savā starpā, sinhronizējot datu apmaiņu. Viens no tādiem mijiedarbības piemēriem parādīts 2.3. attēlā. Šeit vienkāršotā veidā dota shēma, pēc kuras tiek realizēta viena no iespējamajām operācijām datu apmaiņai ar cieto disku. Ar punktēto līniju parādīti etapi ierīču (kas ietilpst mikroshēmojumā) mijiedarbībai ar datorsistēmas aparātūru. Datu apmaiņā piedalās gan saskarnes ATA kontrolleris, kas ietilpst mikroshēmojuma sastāvā, gan I/O kontrolleris. Turklāt ATA izpilda nolaišanās-ierakstīšanas operācijas atbilstoši PCI kopnes specifikācijai, bet I/O kontrolleris sinhronizē datu apmaiņu ar procesora kopni (to parasti sauc par sistēmas kopni) [4].

Dotajā gadījumā ATA kontrolleris veic divvirzienu saskarnes funkcijas: no vienas puses, tas veic datu apmaiņu ar cieto disku saskaņā ar ATA standartu un, no otras puses, nodrošina datu nodošanu un saņemšanu pa PCI kopni. Savukārt I/O kontrolleris nodrošina saskarni starp PCI kopni un sistēmas kopni, t.i., ar procesora signālu līnijām. Īstenībā šis process ir daudz sarežģītāks, jo visos etapos nepieciešama pieejas kopnei sinhronizācija, jo nav izslēgts, ka citas ierīces vienlaicīgi grib tikt pie sistēmas kopnes resursiem. Turklāt datu apmaiņas laikā citas ierīces, ieskaitot procesoru, var izraisīt pārtraukumu vai pieprasīt PCI kopni lietošanai monopola stāvoklī darbam tiešās pieejas atmiņai DMA režīmā.




2.3.att. Mikroshēmojuma un datora aparātūras mijiedarbības piemērs

 Procesors mijiedarbojas ar dažādām ierīcēm ar mikroshēmojumu palīdzību – noteiktā veidā organizētām loģiskām ierīcēm (kontrolleriem). Katrs no kontrolleriem vada datu apmaiņu saskaņā ar kopnes specifikāciju, kurai tas ir pieslēgts. Mikroshēmojums veic vēl vienu svarīgu funkciju – datu plūsmu sinhronizāciju pēc apmaiņas ātruma, jo dažādām ierīcēm ir dažāds ātrums, kas prasa saskaņošanu.

 Kopne ir saskarne, ar kura palīdzību tā vai cita ierīču grupa mijiedarbojas ar procesoru (caur vienu no mikroshēmojuma kontrolleriem). Mūsdienu datorsistēmās ir trīs standarta kopnes: PCI, AGP un USB. Turklāt ir izstrādātas un ieviestas arī citas ātrdarbīgas kopnes, piemēram, PCI Express. Katra no minētajām kopnēm veic tikai tām raksturīgas funkcijas.

Sistēmas kopne nodrošina procesora saskarni ar pārējām ierīcēm, ieskaitot atmiņu.

 AGP kopne nodrošina grafiskā kontrollera saskarni ar informācijas attēlošanas ierīcēm. Tā ir ātrdarbīga kopne, speciāli konstruēta darbam ar augstražīgiem grafiskajiem kontrolleriem.

 PCI kopne nodrošina:

- ārējo ierīču saskarni (tās parasti sauc par PCI kartēm vai adapteriem);
- kopnei pieslēgto ierīču darba sinhronizāciju;
- asinhrono datu apstrādi, izmantojot pārtraukumu mehānismu IRQ (*Interrupt ReQuest*) un tiešās pieejas atmiņai metodi DMA;
- automātisku ierīču atpazīšanu un uzstādīšanu, kuras atbilst specifikācijai PnP (*plug-and-play*).

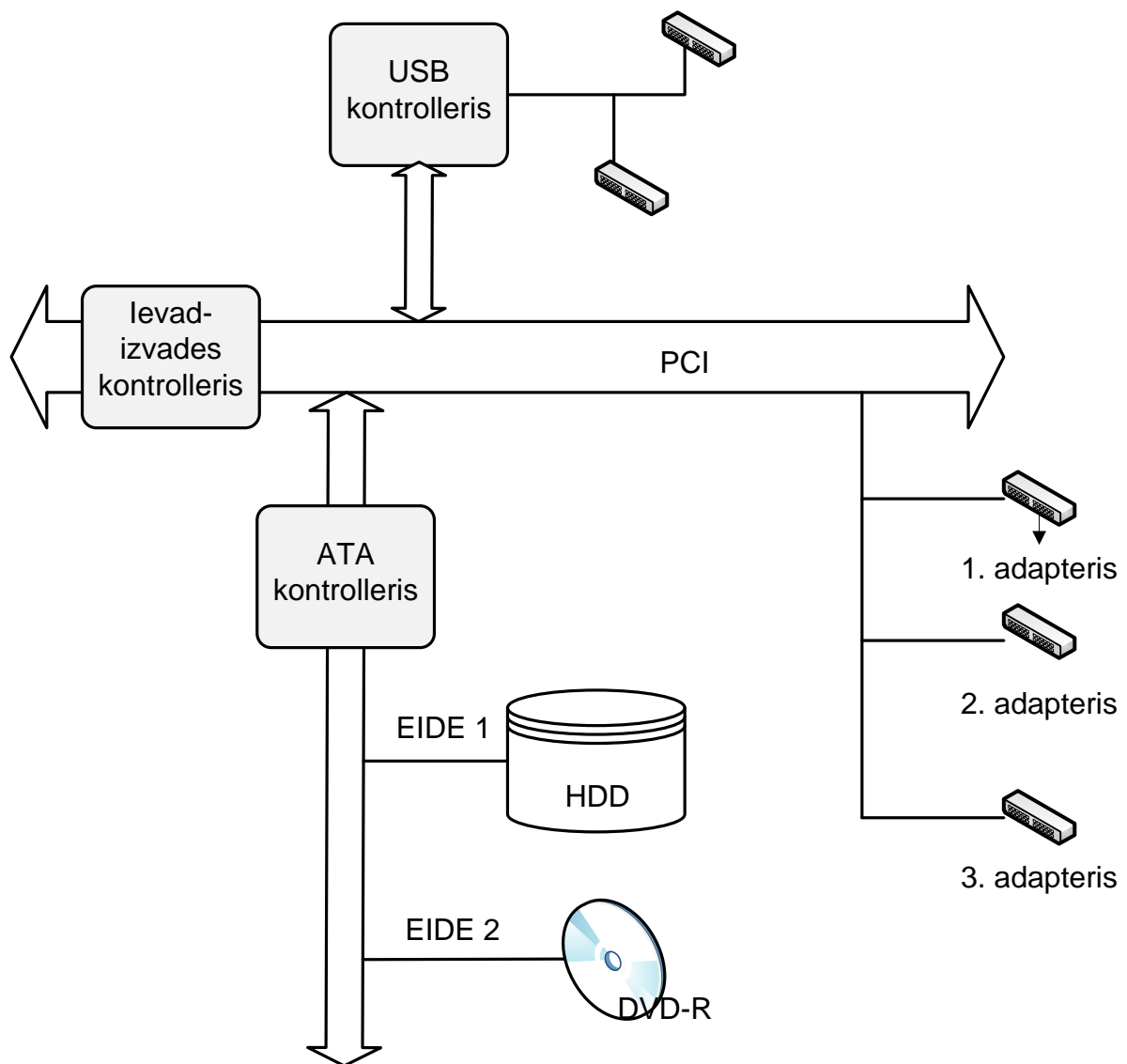
PCI kopnei ir ļoti nozīmīga loma datu apmaiņas procesā starp procesoru un ārējām ierīcēm, tāpēc tās darbība tiks apskatīta detalizētāki.

2.2. PCI kopnes realizācijas piemērs

PCI kopne paredzēta darbam ar 32 bitu datiem, bet praksē strādā kā 64 bitu ierīce. Jāatzīmē, ka PCI ir no procesora neatkarīga, t.i., to var izmantot atšķirīgās no x86 platformās [3,9].

Pa PCI kopni tiek pārraidīti buferizēti dati – procesors ievieto datus buferī, pēc tam sāk izpildīt citus uzdevumus. Kopnes saskarne nodod šos datus citām ierīcēm, kas pieslēgtas kopnei, un nodrošina visus nepieciešamos sinhronizācijas signālus. PCI ierīces (bieži sauc par PCI adapteriem) tāpat nodod datus, ievietojot tos buferī, pēc kā kopnes saskarne nodrošina turpmāko datu pārraidi procesoram. Parasti PCI kopne nodod 32 bitus vienas takts laikā.

2.4.attēlā parādīta iespējamā konfigurācija ierīču pieslēgšanai PCI kopnei.



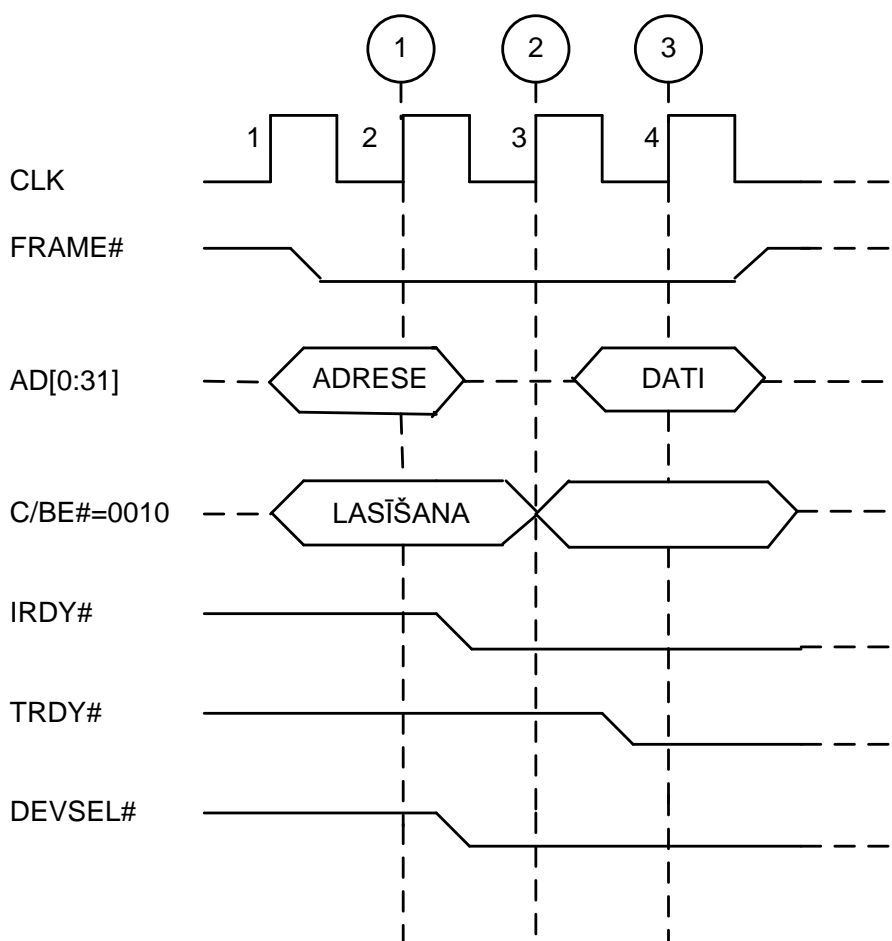
2.4.att. Ierīču pieslēgšanas PCI kopnei variants

Vispārīgi sakot, PCI kopne nav lokāla kopne, jo tā satur, kā minimums, divas funkcionālās daļas:

- iekšējo kopni, ko izmanto EIDE ievadizvades kanālu pieslēgšanai, ar kuru palīdzību tiek realizēta datu apmaiņa ar cieto disku;
- ārējo kopni, kas fiziski realizēta ar paplašināšanas slotu (*expansion slot*) palīdzību ierīču adapteru pieslēgšanai.

Datu apmaiņu kopnē realizē I/O kontrolleris, kas ietilpst visu mikroshēmojumu sastāvā. Turklāt tas veido signālu virkni saskaņā ar PCI specifikāciju, jo, piekļūstot pie I/O ierīcēm, procesors veido signālus atbilstoši sistēmas kopnes specifikācijai. Lai pārvērstu šos signālus PCI standartā, kalpo I/O kontrollera loģika. Kopnes kontrolleris arī ir datu apmaiņas iniciators.

Lai atvieglotu procesu izpratni datu apmaiņas laikā, 2.5.attēlā ir dota PCI kopnes vienkāršota laika diagramma, kas tomēr precīzi raksturo datu apmaiņas mehānismu [4,9].



2.5.att. Datu nolasīšanas PCI kopnē komandas izpildes laika diagramma

Dotais attēls demonstrē to, kādā veidā tiek izpildīta viena no tipiskajām operācijām – datu nolasīšana no ierīces. Jāatzīmē, ka datu apmaiņas princips, kas tiek izmantots PCI kopnē, ir pamats, uz kura balstās datu pārraide arī citās kopnēs (AGP, PCI Express), neskatoties uz to atšķirībām aparatūras realizācijas ziņā un saskarnes signālu specifikācijā.

Kopnes arhitektūrā atspoguļots viens no pamatprincipiem, kas kopīgs visām datorsistēmām, – rokspiešanas princips (*handshaking*), kas nozīmē to, ka datu apmaiņa sākas tikai tad, kad iekārtas ir tam gatavas.

Ja kāda no ierīcēm nav gatava, I/O operācija var tikt atlikta līdz ierīces gatavībai vai atcelta. Turklāt visās apmaiņas stadijās operācija tiek sinhronizēta ar takts frekvences sinhroimpulsiem. Datu apmaiņas sākumu iniciē viena no datorsistēmas ierīcēm – visbiežāk tas ir procesors. I/O operācijas bieži sauc par transakcijām, kuras savukārt var sadalīt atsevišķās izpildes fāzēs.

Laika diagrammas analīzei nepieciešams dot tajā izmantoto signālu aprakstu (pilnu signālu aprakstu var atrast PCI dokumentācijā):

- CLK – takts frekvences sinhroimpulsi;
- FRAME# – zems signāla līmenis atļauj veikt nolasīšanas-ierakstīšanas operāciju kopnē. Šis signāls ir apmaiņas iniciators un apliecina kopnes cikla sākumu. Iniciators nosaka ierīces adresi līnijās AD[0:31], kurš var tikt izmantots I/O ierīču loģiskajās shēmās operācijas izpildes turpināšanai dotajā ciklā. Šajā brīdī signālam FRAME# jābūt zēmam līmenim. Ja signāllīnijā tiek iestatīts loģiskais 1, tas liecina par to, ka operācija ir pabeigta.
- AD[0:31] – signāli šajās līnijās tiek multipleksēti. Datu apmaiņas sākumā šajās līnijās iestata 32 bitu adresi, kura tiek fiksēta pēc pirmā sinhroimpulsa krituma un atļaujās līnijā FRAME#. Turpmāk šajās līnijās fiksē 32 bitu datus, kas tiek

nolasīti vai ierakstīti saskaņā ar I/O ierīces loģiku. Datu nolasīšana vai ierakstīšana tiek veikta tikai gadījumā, ja signāliem līnijās IRDY# un TRDY# ir zems līmenis.

- IRDY# – iniciators iestata zemā līmenī, kas liecina par datu apstrādes iespēju, kuri fiksēti līnijās AD[0:31]. Kad signāls IRDY# ir zemā līmenī, iniciators gaida signāla TRDY# noteikšanu zemā līmenī, lai varētu sākt transakciju. Turklāt I/O ierīce var pārtraukt transakciju – iestatot signālu STOP# zemā līmenī – nekādas operācijas turpmāk netiek izpildītas.
- TRDY# – I/O iekārta iestata zemā līmenī, norādot uz datu gatavību līnijā AD[0:31]. Iekārta notur signālu TRDY# zemā līmenī tik ilgi, kamēr signāls IRDY# atrodas loģiskās 0 stāvoklī, tādējādi nodrošinot transakcijas izpildi.
- DEVSEL# – I/O ierīce iestata loģiskās 0 stāvokli, adrese līnijā AD[0:31] sakrīt ar ierīces adresi. Signāls DEVSEL# nevar būt novests loģiskā 1 stāvoklī līdz datu apstrādes beigām. Ierīce var noņemt signālu DEVSEL#, ja kaut kādu iemeslu dēļ nepieciešams pārtraukt operāciju (šajā gadījumā veidojas signāls STOP#).
- C/BE[3:0]# – šīs signāllīnijas tiek multipleksētas atkarībā no tā, kāda datu apmaiņas fāze izpildās dotajā momentā. Adreses iestatīšanas laikā šajās līnijās atspoguļojas komanda, kurai jāizpildās. Piemēram, datu nolasīšanas komandai no ierīces (I/O Read) ir kods 0010, datu ierakstīšanas komandai ierīcē (I/O Write) – 0011 utt. Datu fāzē 4 biti šajās līnijās atļauj operāciju ar 32 bitu datu atsevišķiem baitiem. Piemēram, bits C/BE[3] atļauj vecākā baita AD[31:24] apstrādi. Signālus līnijās C/BE[3:0] iestata iniciators, un tie tiek noturēti aktīvā stāvoklī visās transakcijas fāzēs.

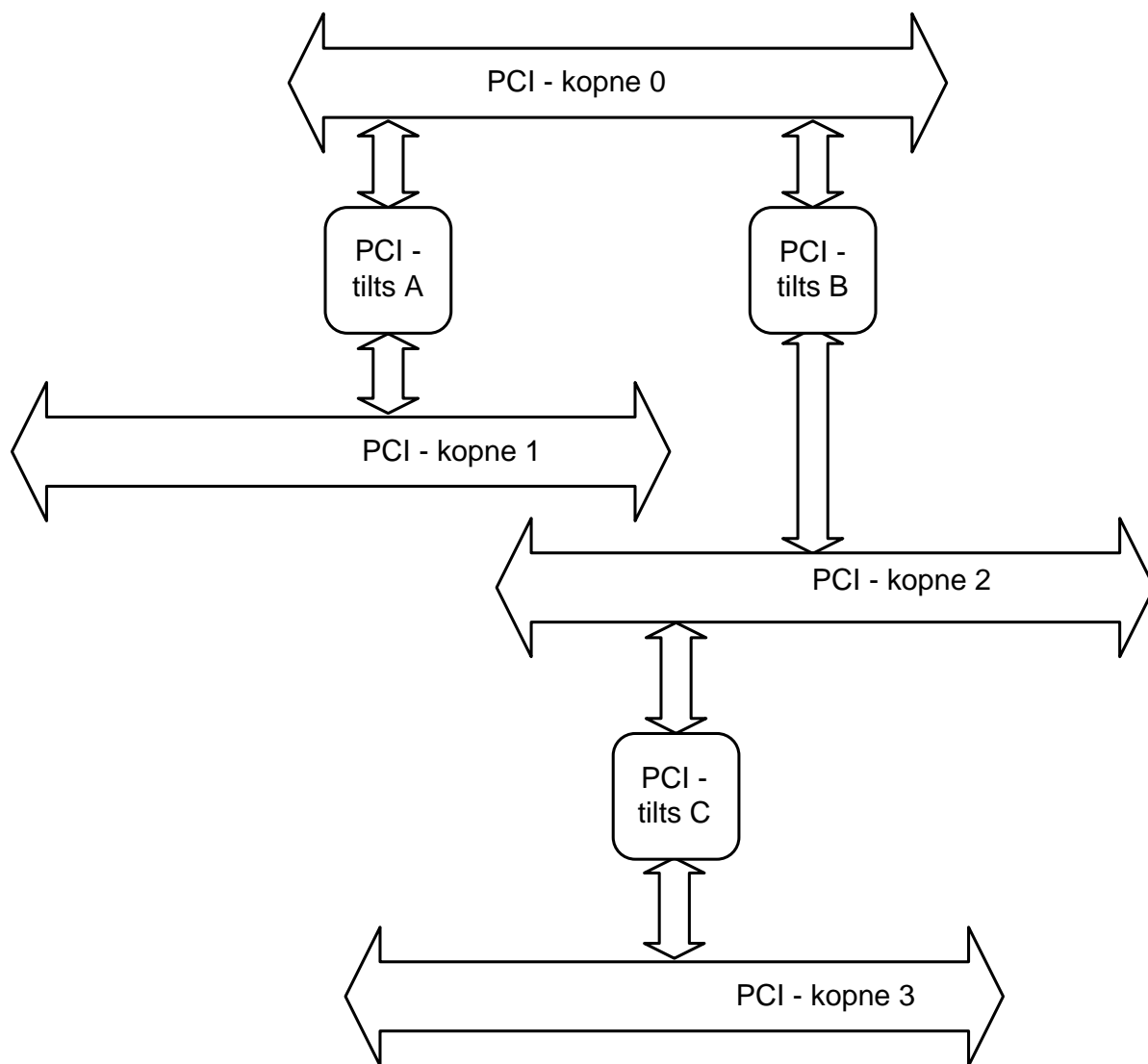
Datu nolasīšanas operācija PCI kopnē

1. Apmaiņas iniciators veido signāla FRAME# zemu loģisko līmeni, ar to atļaujot operāciju.
2. Iniciators uzdod ierīces adresi, pie kuras notiek vēršanās (t.s. saņēmējs) un komandas C/BE bitos [3:0] kodu (dotajā gadījumā 0010, kas atbilst nolasīšanas operācijai) – atzīme 1 laika diagrammā 2.5.attēlā. Ja ierīce – saņēmējs - atpazīst adresi, tā iestata zemu signāla līmeni līnijās DEVSEL#.
3. Iniciators iestata zemu signāla IRDY# (*Initiator ready*) līmeni, norādot uz gatavību saņemt datus. Tā kā saņēmējs nav iestādījis datu gatavības signālu TRDY# (*Target ready*) zemā līmenī, tad datu nolasīšana netiek veikta (atzīme 2 laika diagrammā).
4. Saņēmējs iestata signālu TRDY# zemā līmenī, apstiprinot datu gatavību, pēc kā ceturtajā sinhroimpulsā izpildās datu nolasīšana AD līnijās (atzīme 3 laika diagrammā).

Turpmāk var izpildīt kārtējās nolasīšanas operācijas, bet operāciju pārtraukšanai iniciatoram jāiestata augsts signāla FRAME# līmenis. Pēc līdzīgas shēmas notiek arī datu ierakstīšana ierīcē.



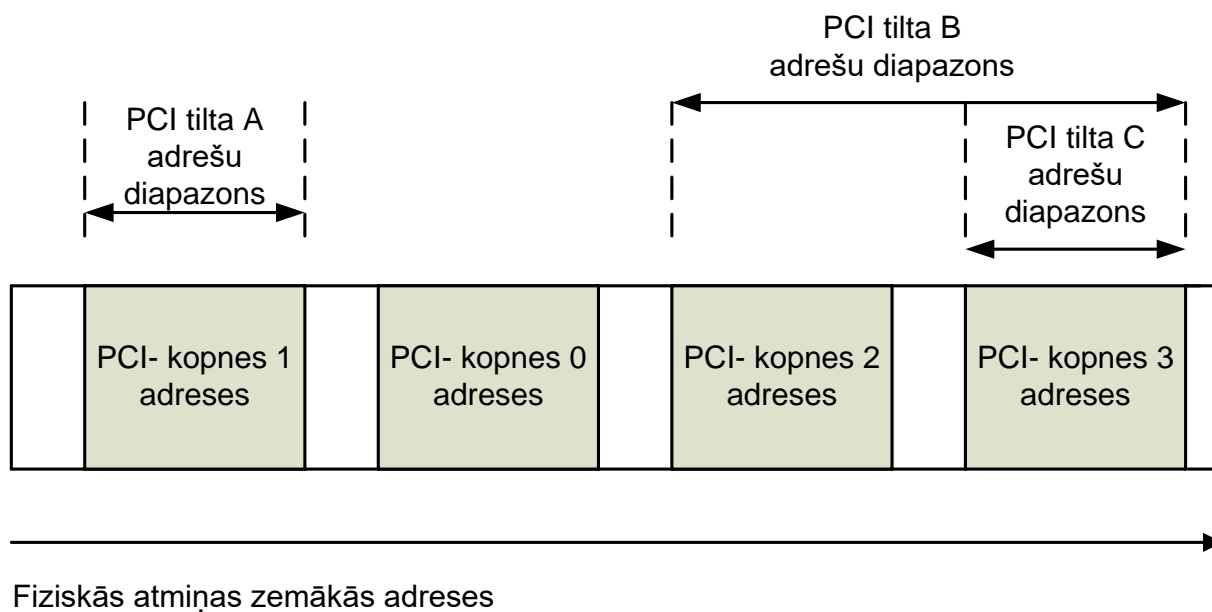
Viens no klasiskās PCI kopnes trūkumiem ir ierobežotais fizisko paplašināšanas slotu skaits, pie kurām var pieslēgt ierīces – to ir tikai 4. To nosaka ierobežojumi, kas saistīti ar kopnes neterminētajiem galiem. Šīs problēmas risināšanai tika izveidots speciāls PCI kopnes paplašināšanas mehānisms – tā saucamais PCI-PCI tilts (*PCI-to-PCI bridge*), ar kura palīdzību var palielināt PCI kopnei vienlaicīgi pieslēdzamo ierīču skaitu (sk. 2.6. attēlu).



2.6.att. PCI tiltu funkcionēšanas shēma

PCI kopnes paplašināšanas koncepcija ir diezgan vienkārša. Minimālajā konfigurācijā sistēmai, kā minimums, ir viens PCI tilts, sarežģītākās konfigurācijās – vairāki tilti. Vairāku PCI kopņu gadījumā pieeja pie ierīcēm izpildās, izmantojot papildus adresāciju. Katram PCI tiltam ir viens vai divi reģistri, kas satur bāzes adresi, pieļaujamās adresu telpas apjomu un adreses tipu (ievadizvades pieslēgvietas vai atmiņa). Reģistru saturs faktiski norāda uz logu I/O adresu telpā (*I/O address space*) vai arī uz logu atmiņā. Konfigurācijai 2.6. attēlā adresu telpa katrai no kopnēm var tikt uzdots veidā, kā tas ir parādīts 2.7. attēlā.

Informāciju par ierīču pieslēgšanu pie tās vai citas kopnes, kā arī šo ierīču izmantojamās resursus var apskatīt *Control Panel / Administrative Tools / Computer Management / Device Manager*.



2.7. att. PCI kopnes adresācija

Papildus informācija par kopņu PCI un PCI Express specifikāciju: <http://www.pcisig.com>.

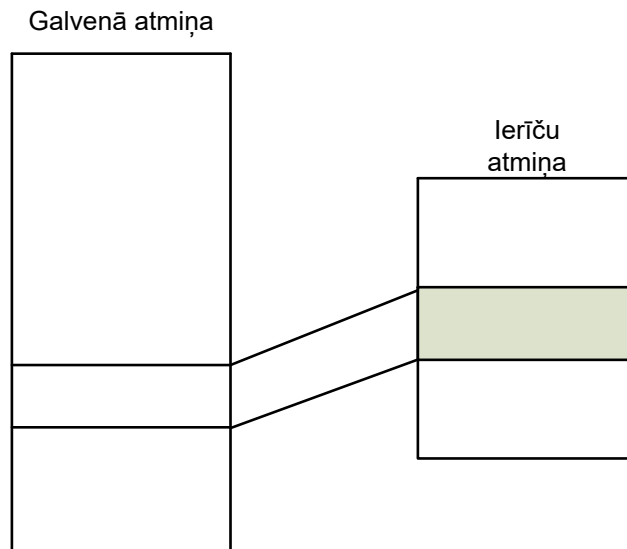
2.3. Ievadizvades ierīces

x86 procesori nodala pieeju pie atmiņas un I/O ierīcēm multipleksējot adresu līnijas zemākos 16 bitus. IA-32 arhitektūrā reģistru adresu telpu veido 65535 iespējamās adreses – tas nozīmē, ka teorētiski datorsistēmā var būt tieši tāds reģistru daudzums.

⚠ Galvenā atšķirība starp I/O ierīcēm un atmiņas ierīcēm slēpjas tajā apstākļī, ka atmiņas ierīces pārsvarā izmanto I/O pieslēgvietas (*port*), kas realizētas reģistru (*register*) veidā. Reģistru var salīdzināt ar fiziskās atmiņas šūnu, taču pieeja reģistram tiek realizēta daudz ātrāk aparātūras realizācijas īpatnību dēļ. Turklāt adresācijas un datu pārraides caur reģistriem loģika realizēta daudz vienkāršāk nekā operatīvajā atmiņā, kas arī ietekmē ātrdarbību.

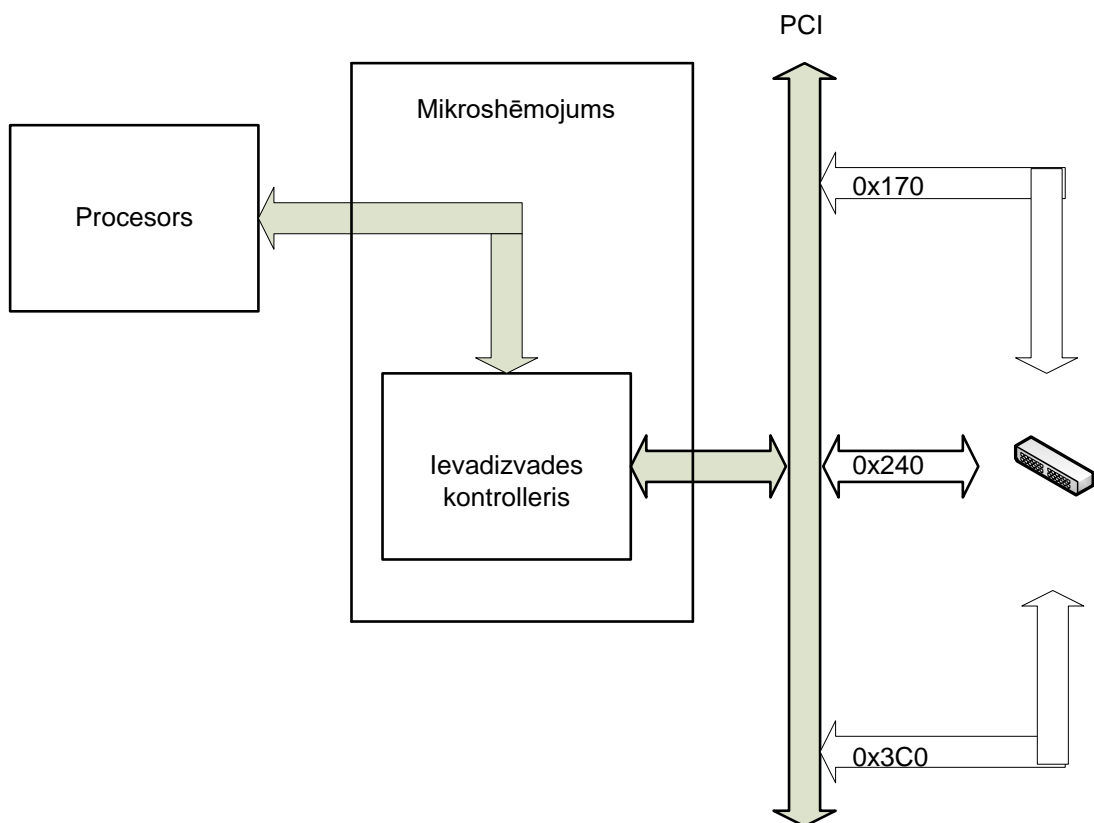
Pieeja pie ierīces reģistriem tiek realizēta ar procesora komandu *in* un *out* palīdzību [6], turklāt reģistra adrese tiek uzrādīta pašā komandā. Bieži tiek izmantotas ierīces, piekļuve kurām notiek caur atmiņas adresi, t.i., tiek realizēta ierīces reģistru attēlošana atmiņā (*memory-mapped registers*). Tādos gadījumos piekļuvei pie ierīces reģistriem tiek izmantota komanda *mov* un tās modifikācijas, t.i., pieeja tādiem reģistriem tiek veikta tāpat kā pie atmiņas šūnām.

Daudzos gadījumos I/O ierīcei izdalītās atmiņas telpas adresu skaits ir nepietiekams efektīvai ierīces darbībai. Turklāt, ja ierīce savam darbam aizņem lielāko adresu telpas daļu, būs neiespējami izdalīt resursus citām ierīcēm. Šī iemesla dēļ izstrādātāji izmanto uz pašas ierīces izvietotu noteiktu operatīvās atmiņas apjomu (tīkla kartes, grafiskie adapteri utt.). Pieeja tādu ierīču atmiņai notiek pilnīgi tādā pašā veidā, it kā tā atrastos kopējā fiziskajā atmiņā. Šiem nolūkiem tiek izmantota šāda metode – atmiņā tiek izdalīts kopējās atmiņas apgabals vai logs, caur kuru notiek ierīces atmiņas attēlošana. Tādu apgabalu sauc par koplietošanas atmiņu (*shared*). Tādā gadījumā daļa ierīces atmiņas tiek attēlota galvenajā atmiņā (sk. 2.8. att.).



2.8. att. Ierīces atmiņas izmantošana

I/O ierīces programmu modulis satur noteiktu reģistru skaitu un atmiņas adresu diapazonu (ja tā tiek izmantota). I/O modulis, kas attēlots 2.9. attēlā, izmanto trīs reģistrus ar adresēm 0x170, 0x240, 0x3C0.



2.9. att. I/O ierīču adresācija ar reģistru palīdzību



2.1. piemērs

Informācijas baita ierakstīšanai reģistrā 0x170 var izmantot šādas procesora komandas:

```
mov AL, data
mov DX, 0x170
out DX, AL
```

Šajā gadījumā reģistrā AL tiek ievietota datu baita vērtība, bet reģistrā DX – reģistra adrese. Datu nolasišanai no reģistra var izpildīt komandu virkni:

```
mov Dx, 0x170
in AL, DX
```

Pēc šo komandu izpildes reģistrā AL atradīsies baita vērtība no pieslēgvietas, kura adrese atrodas reģistrā DX.



Windows operētājsistēmās tādā veidā piekļūt pie aparatūras resursiem no lietotāja programmas nav iespējams, jo tikai lietojumprogrammas, kas darbojas kodola režīmā, pa tiešo var strādāt ar datora aparatūru. Tādas lietojumprogrammas ir visi kodola režīma draiveri (*driver*), kā arī atsevišķi sistēmas dienesti.

2.4. Fiziskās atmiņas organizācija

Atmiņas organizācija un funkcionēšana kardināli ietekmē datorsistēmas veiktspēju un pielietojumu izpildes laiku, taču atmiņas ātrdarbība būtiski atpaliek no procesora ātrdarbības. Tas nozīmē to, ka aizture operācijās ar atmiņu palielina komandu izpildes laiku un samazina sistēmas darbību kopumā, lai cik ātrs nebūtu procesors.

Procesoru izstrādātāji cīnās ar atmiņas aizturi dažādos veidos, piemēram, izmanto datu kešdarbi (*caching*), kas noteiktā veidā realizē datu un komandu izlasi no atmiņas. Taču ne mazāk būtiska loma atmiņas izmantošanas optimizācijā ir programnodrošinājuma izstrādātājiem, jo pastāv daudz metožu, kas ļauj uzlabot lietojumprogrammu veiktspēju [4,9].

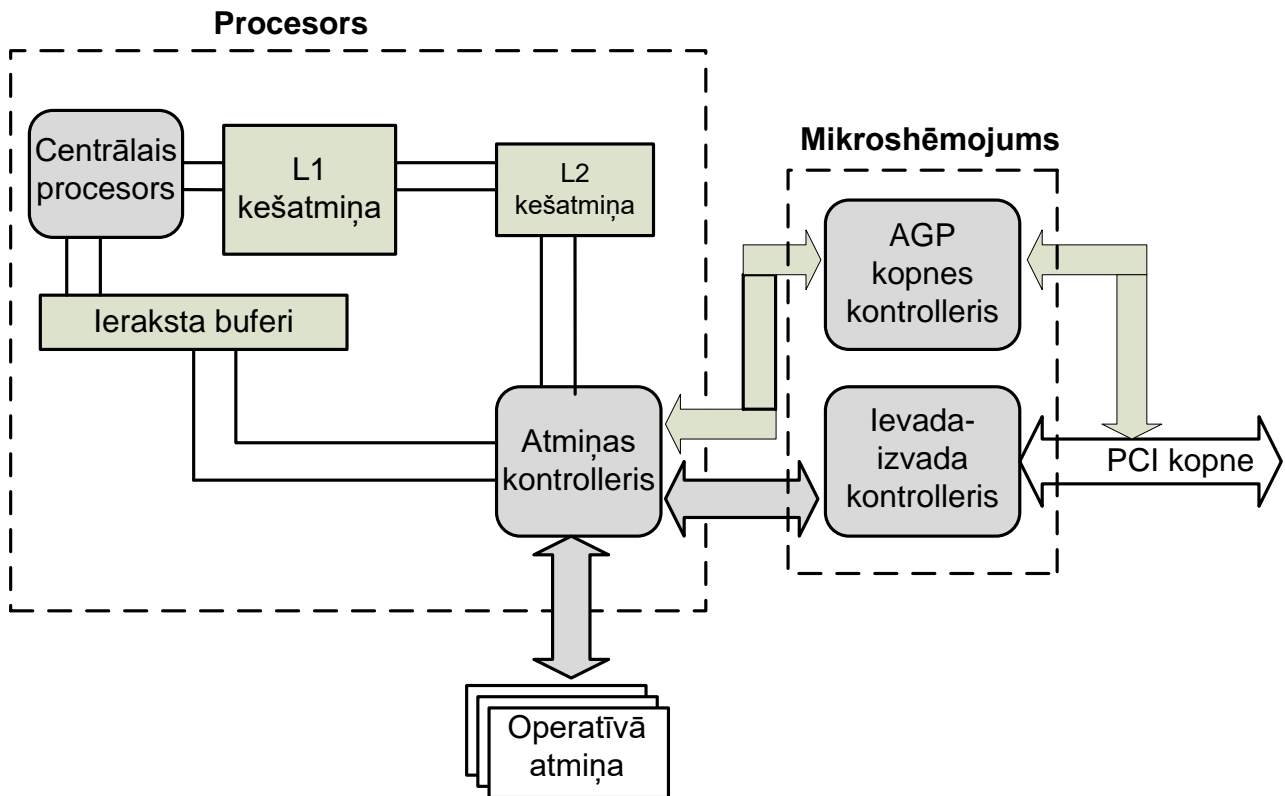
Atmiņas vadībai procesorā iekļauts speciāls atmiņas vadības modulis – atmiņas kontrolleris (*Memory controller*). Šādā kontekstā datu apmaiņas shēma datorsistēmā var tikt realizēta 2.10. attēlā redzamajā veidā.

Dotā shēma ataino datu apmaiņu datorsistēmā, kas izmanto *Intel Pentium 4* procesoru. Visas operācijas datu apmaiņā ar ierīcēm no procesora puses veic atmiņas kontrolleris.

Programmā izmantojamās atmiņas apjomu ierobežo procesora adrešu telpas maksimālais apjoms. *Intel Pentium 4* procesoriem, piemēram, maksimālās adrešu telpas apjoms (ieskaitot fizisko jeb operatīvo un virtuālo atmiņu) var sasniegt 64 Gb.

Lietojumprogrammas izmanto tikai virtuālo atmiņu, turklāt tās pat „neredz”, ka izdalītais atmiņas adrešu diapazons var pārsniegt reālo fiziskās atmiņas apjomu. Ja operētājsistēma konstatē strauju brīvās atmiņas trūkumu, procesors ģenerē izņēmuma stāvokli *page fault*, pēc kura atmiņas apgabals, ar ko strādā lietojumprogramma, tiek ielādēts tā saucamajā lappušetmiņā (*page memory*), tādējādi atbrīvojot fizisko atmiņu. Lappušetmiņa ir atmiņa, kas sadalīta vienādos apgabalos – lappusēs. Tādu sadalījumu izmanto atmiņas aizsardzībai vai dinamiskai pāradresēšanai virtuālās atmiņas pārvaldības procesā. Datu apmaiņa lappušetmiņā notiek veselām lappusēm. Diemžēl lappušetmiņas ielāde un tās atbrīvošana aizņem relatīvi ilgu laiku un ir atkarīga no cietā diska (uz kura tiek realizēta lappušetmiņu izvietošana) ātrdarbības. Tas izraisa datorsistēmas veiktspējas samazināšanos.

Datorsistēmu veiktspējas palielināšanai izstrādāts kešatmiņas realizācijas mehānisms. Tā ir ātrdarbīga atmiņa, kas iebūvēta procesorā.



2.10. att. Datu apmaiņas organizācijas shēma

⚠ Galvenais kešatmiņas uzdevums – kompensēt relatīvi lēnās operatīvās atmiņas darbu attiecībā pret procesoru. *Intel Pentium 4* procesoram ir divi kešatmiņas veidi – pirmā līmeņa kešatmiņa (*L1 cache*) un otrā līmeņa kešatmiņa (*L2 cache*). Jāatzīmē, ka *Intel Pentium 4* procesoriem ir arī pirmā līmeņa komandu kešatmiņa (*Trace cache*), kuras darbības specifika atšķiras no datu kešdarbes.

L1 kešatmiņai ir mazs apjoms (8 kb), taču liela ātrdarbība (tomēr mazāka nekā procesoram). L2 kešatmiņu reizēm sauc par universālo kešatmiņu, tā kā tajā var atrasties gan dati, gan komandas. Otrā līmeņa kešatmiņas apjoms ir 256 Kb (vai lielāks), un tā strādā 3x lēnāk nekā L1.

Datu apmaiņā ar operatīvo atmiņu visbūtiskāko lomu spēlē tieši L1 kešatmiņa, jo praktiski visi strādājošo lietojumprogrammu dati iziet caur to, kā arī ieraksta buferi (*write buffers*).

Datu apmaiņas mehānisms ar fizisko atmiņu

Ja lietojumprogrammai nepieciešams vērsties pie atmiņas apgabala, procesors vispirms mēģina atrast nepieciešamos datus kešatmiņā. Ja dati ir atrasti, tiek fiksēta tā saucamā „trāpīšana” kešā (*cache hit*), pēc kā dati no kešatmiņas tiek apstrādāti. Ja dati kešatmiņā netiek atrasti (*cache miss*), notiek vērsšanās pie operatīvās atmiņas. Šajā gadījumā procesors nolasa datus no atmiņas, turklāt nevis baitu pa baitam, bet blokos pa 64 baitiem. Tādējādi dati izvietojas kešatmiņā 64 baitu rindas veidā. Ja, piemēram, programmai nepieciešams nolasīt 71. baitu, tad šajā gadījumā kešatmiņā tiks ielādēti baiti no 64 līdz 127.

Kešatmiņas funkcionēšanas pamatā ir divi fundamentāli principi:

- datu lokalizācija telpā (*spatial locality*) un
- datu lokalizācija laikā (*temporal locality*).

Lokalizācija telpā paredz to, ka atmiņā virknes veidā izvietotie dati ar lielu varbūtību var tikt izmantoti turpmākajā programmas izpildē. Datu lokalizācijas laikā princips nosaka to, ka, iespējams, atkal notiks piekļuve kešatmiņas datiem, tāpēc tie glabājas kešatmiņā noteiktu laiku.

Būtiska loma atmiņas izmantošanas efektivitātes paaugstināšanā ir datu izlīdzināšana. Daudzas procesora SIMD komandas pieprasa datu izlīdzināšanu pirms operācijas izpildes.

Izlidzināšanas direktīvu izmantošana lietojumprogrammās ir viens no vienkāršākajiem paņēmieniem lietojumprogrammas optimizēšanai.

Atmiņas izmantošanas optimizācija balstās uz diviem principiem:

- izmantot pēc iespējas mazāk atmiņas operāciju izpildīšanai;
- veikt operācijas ar datiem atmiņā pēc iespējas ātrāk.

Kad tika apskatītas operācijas ar datiem kešatmiņā, tika uzskatīts, ka tie jau tur atrodas. Taču bieži gadās situācijas, kad lietojumprogramma noteiktu laiku neizmanto atmiņu. Tādos gadījumos gaidīšanas laika samazināšanai izmanto datu pirmsienesi kešatmiņā (par to vairāk būs aprakstīts 7. nodaļā). Šim nolūkam izmanto šādas procesora instrukcijas:

- *prefetchnta* – ielādē datus kešatmiņā (tikai nolasīšanai);
- *prefetch0* – ielādē datus visās kešatmiņas (pieejami nolasīšanai un ierakstīšanai);
- *prefetch1* – ielādē datus L2 un L3 kešatmiņā (bet ne L1 kešatmiņā);
- *prefetch2* – ielādē datus tikai L3 kešatmiņā.

Ar retiem izņēmumiem visās nolasīšanas un ierakstīšanas operācijās tiek izmantota L1 kešatmiņa, bet tas prasa papildus laiku. Vairākos gadījumos ierakstīšanas operāciju kešdarbe var izraisīt veiktspējas samazināšanos, piemēram, operācijās ar PCI adapteru reģistriem un ierīču draiveriem. Šī iemesla dēļ tāda tipa operācijās netiek veikta kešdarbe, un dati tur tiek ierakstīti nekavējoties pēc to apstrādes – neizmantojot kešatmiņu. Tādiem atmiņas apgabaliem ir zema ātrdarbība, jo katrai ierakstīšanas operācijai nepieciešams atsevišķs kopnes cikls.

Gadījumos, kad kešatmiņā nav vajadzīgo datu, tiek izmantotas speciālas straumējuma (*stream*) procesora instrukcijas:

- *movnti* – ieraksta 32 bitu veselas skaitļu vērtības;
- *movntq* - ieraksta 64 bitu veselas skaitļu vērtības;
- *movntdq* - ieraksta 128 bitu veselas skaitļu vērtības;
- *movntpd* - ieraksta 128 bitu mainīgo ar peldošo punktu vērtības (satur 2 dubultprecizitātes mainīgos);
- *movntps* - ieraksta 128 bitu mainīgo ar peldošo punktu vērtības (4 mainīgie);



2.2. piemērs. Ātra datu kopēšana, neizmantojot kešatmiņu

```
// 2-2 Atra datu kopeshana
#include "stdafx.h"
#include <stdio.h>
int main(void)
{
    __declspec (align(16)) float s1[4] = {3.43, -23.98, 7.45, 1.112};
    __declspec (align(16)) float d1[4];
    __asm
    {
        movupd XMM0, s1
        movntps d1, XMM0
    }
    for (int i1=0; i1<4; i1++)
    {
        printf("%6.3f\n", d1[i1]*1.7);
    }
    getchar();
    return 0;
}
```

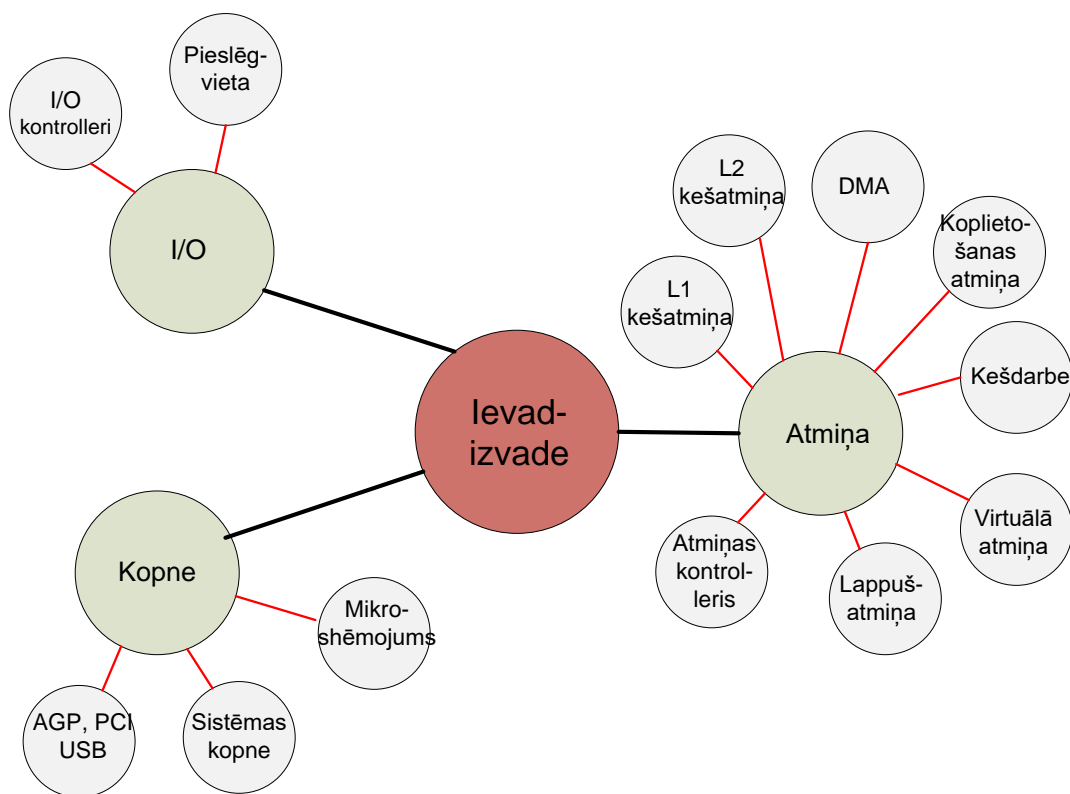
Rezultāts

5,831 -40,776 12,665 1,890

NODAĻAS KOPSAVILKUMS

- Datorsistēmu var uzdot kā vairāku apakšsistēmu kopumu: procesors, operatīvā atmiņa, informācijas uzglabāšanas ierīces un lietotāja saskarnes ierīces.
- Datorsistēmas ierīces savā starpā saistītas ar vienu vai vairākām kopnēm.
- Lai nodrošinātu ierīču darbu, kas pieslēgtas kopnēm, tika izstrādāti mikroshēmojumi.
- Procesors mijiedarbojas ar dažādām ierīcēm ar mikroshēmojumu palīdzību - noteiktā veidā organizētām loģiskām ierīcēm (kontrolleriem).
- Mūsdienu datorsistēmās ir trīs standarta kopnes: PCI, AGP un USB.
- Sistēmas kopne nodrošina procesora saskarni ar pārējām ierīcēm, ieskaitot atmiņu.
- Atmiņas ierīces pārsvarā izmanto I/O pieslēgvietas, kas realizētas reģistru veidā.
- Atmiņas organizācija un funkcionēšana kardināli ietekmē datorsistēmas veiktspēju.
- Galvenais kešatmiņas uzdevums – kompensēt relatīvi lēnās operatīvās atmiņas darbu attiecībā pret procesoru.
- *Intel Pentium 4* procesoram ir divi kešatmiņas veidi – pirmā līmeņa kešatmiņa un otrā līmeņa kešatmiņa.

2.11.attēlā parādīts otrajā nodaļā minēto svarīgāko jēdzienu koks.



2.11. att. Otrajā nodaļā minēto jēdzienu koks



Uzdevumi un jautājumi patstāvīgam darbam

1. Aprakstīt kāda datora mikroshēmojuma konfigurāciju.
2. Kāds procesora signāls aktivizē PCI kopnes līniju #FRAME?
3. Datorā Nr.1 katras komandas izpilde aizņem 10 ns, bet datorā Nr. 2 – 5 ns. Vai var apgalvot, ka dators Nr. 2 strādā ātrāk?
4. Datora atmiņas apjoms var būt 268 435 456 baiti. Kāpēc izstrādātāji izvēlējās tādu skaitli, nevis, piemēram, 250 000 000?

5. Datorā ir kopne ar takts ciklu 25 ns. Viena cikla laikā dators var nolasīt vai ierakstīt atmiņā 32 bitu vārdu. Sistēmā ir disks, kurš izmanto kopni un pārsūta informāciju ar ātrumu 40 Mb/s. Centrālais procesors izsauc no atmiņas un izpilda vienu 32 bitu komandu katrās 25 sekundēs. Cik lielā mērā disks palēnina procesora darbu?
6. Datu pārraides ātrums starp centrālo procesoru un atmiņu ir lielāks nekā datu pārraides ātrums starp procesoru un ievadizvades ierīcēm. Kādā veidā šī neatbilstība izraisa datora veiktspējas samazināšanu? Kā to pārvarēt?
7. Pieņem, ka datorā ir pirmā līmeņa kešatmiņa ar pieejas laiku 5 ns un otrā līmeņa kešatmiņa ar pieejas laiku 10 ns. Pieejas laiks pie operatīvās atmiņas ir 50 ns. Ja 20% no pieejas atmiņai veic pirmā līmeņa kešatmiņa, bet 60% - otrā līmeņa kešatmiņa, tad kāds ir vidējais pieejas laiks?
8. Ja programmai jānolasa 145. baitu, kādai datu virknei jābūt ielādētai kešatmiņā?

3. ATMIŅAS LOĢISKĀ ORGANIZĀCIJA UN VIRTUĀLĀ ATMIŅA

Viens no svarīgākajiem jautājumiem datoru arhitektūrā ir atmiņas vadība. Šajā nodaļā ir analizēta atmiņas arhitektūra, kas atspoguļo OS un atmiņas mijiedarbību, kā arī atklāj lietojumprogrammu atmiņas optimizācijas nozīmi, jo bieži vien problēmas rada neefektīva atmiņas izmantošana.

Nodaļā detalizēti ir raksturots arī virtuālās atmiņas mehānisms un tās pārvaldības aspekti.

3.1. Virtuālās atmiņas struktūra

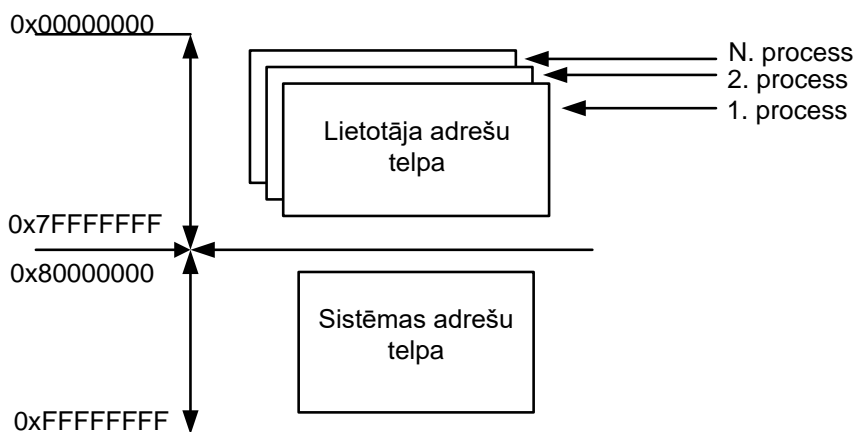
Mūsdienās visas operētājsistēmas izmanto virtuālās atmiņas koncepciju.

Šīs koncepcijas pamatā ir priekšstats par nepārtrauktu adresu telpu, ar kuru var darboties OS un izpildāmie procesi neatkarīgi no tā, kādā veidā realizēta fiziskā atmiņa. Tādā gadījumā visa atmiņa tiek uzskatīta par virtuālo adresu telpu, izņemot nelielus apgabalus, kuros notiek fizisko adresu pārveidošana loģiskajās adresēs [4,5,7,9].

Kā zināms, 32 bitu pielietojumiem teorētiski var izdalīt virtuālo adresu telpu līdz 4 Gb. Turklāt atmiņas adresu rādītājs varētu pieņemt vērtības no diapazona 0x00000000 līdz 0xFFFFFFFF, kura vērtība ir 4 294 967 296.

Patiesībā virtuālā adresu telpa 32 bitu pielietojumiem ierobežota ar 2 Gb apjomu, kas saistīts ar adresu telpas sadalīšanu divos apgabalos: lietotāju adresu telpā un sistēmas adresu telpā.

Lietotāju adresu telpa tiek izmantota resursu izdalīšanai strādājošajām lietojumprogrammām, bet sistēmas apgabals paredzēts pašas OS vajadzībām (kodols, ierīču draiveri, sistēmas dienesti utt.). Turklāt lietotāja procesi nevar tiešā veidā vērsties pie atmiņas resursiem, kas izdalīti sistēmai. Atmiņas sadalījums apgabalos ilustrēts 3.1. attēlā.



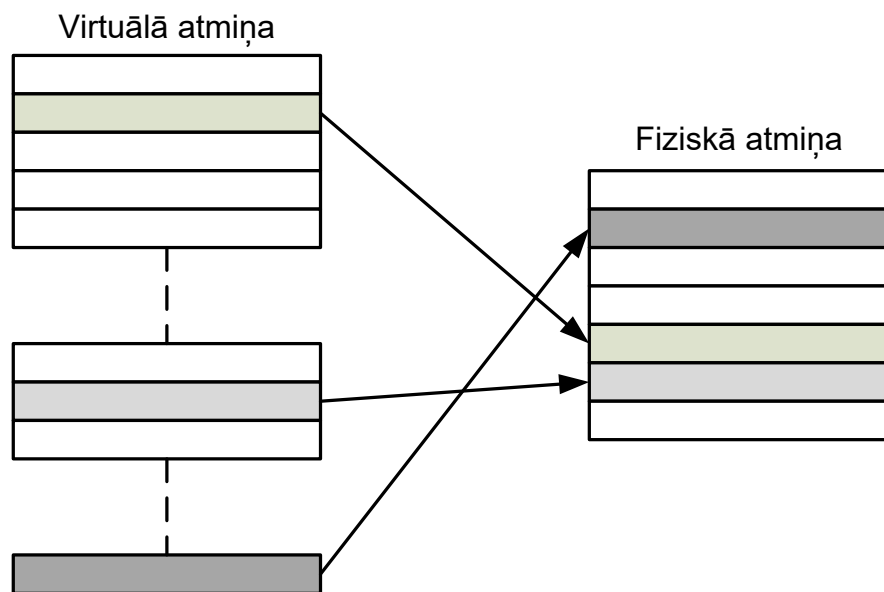
3.1. att. Virtuālās adresu telpas sadalījums divos apgabalos

Atsevišķos gadījumos var pārvarēt 2 Gb barjeru, pie OS ielādes uzdodot speciālu opciju, kas ļauj palielināt virtuālās adresu telpas apjomu līdz 3 Gb (32 bitu OS versijās) vai līdz 4 Gb (64 bitu OS versijās).

⚠ Virtuālās atmiņas vadību veic atmiņas pārvaldnieks, kas nodrošina divu svarīgu uzdevumu izpildi.

- Pirmais uzdevums ir translācija jeb, citiem vārdiem sakot, virtuālās adresu telpas attēlošana (*mapping*) fiziskajā atmiņā. Turklāt jebkurai plūsmai, kas izpildās dotā procesa kontekstā, jāsaņem korektas norādes uz atbilstošajām fiziskās atmiņas

adresēm. Adrešu translācija tiek veikta ar fiksēta apjoma blokiem jeb lappusēm. Tā, piemēram, OS *Windows* lappuses apjoms pēc noklusējuma ir 4 Kb (to var izmainīt). Virtuālo adrešu translācijas shēma fiziskajās adresēs parādīta 3.2. attēlā.



3.2. att. Fiziskā un virtuālā atmiņa

- Otrs uzdevums, ko veic atmiņas pārvaldnieks, – nodrošināt operatīvās atmiņas un cietā diska lapošanu (*paging*). Termins „lappušu apmaiņa” tiek lietots tā iemesla dēļ, ka visas manipulācijas ar virtuālo atmiņu tiek veiktas ar fiksēta apjoma blokiem (lappusēm).

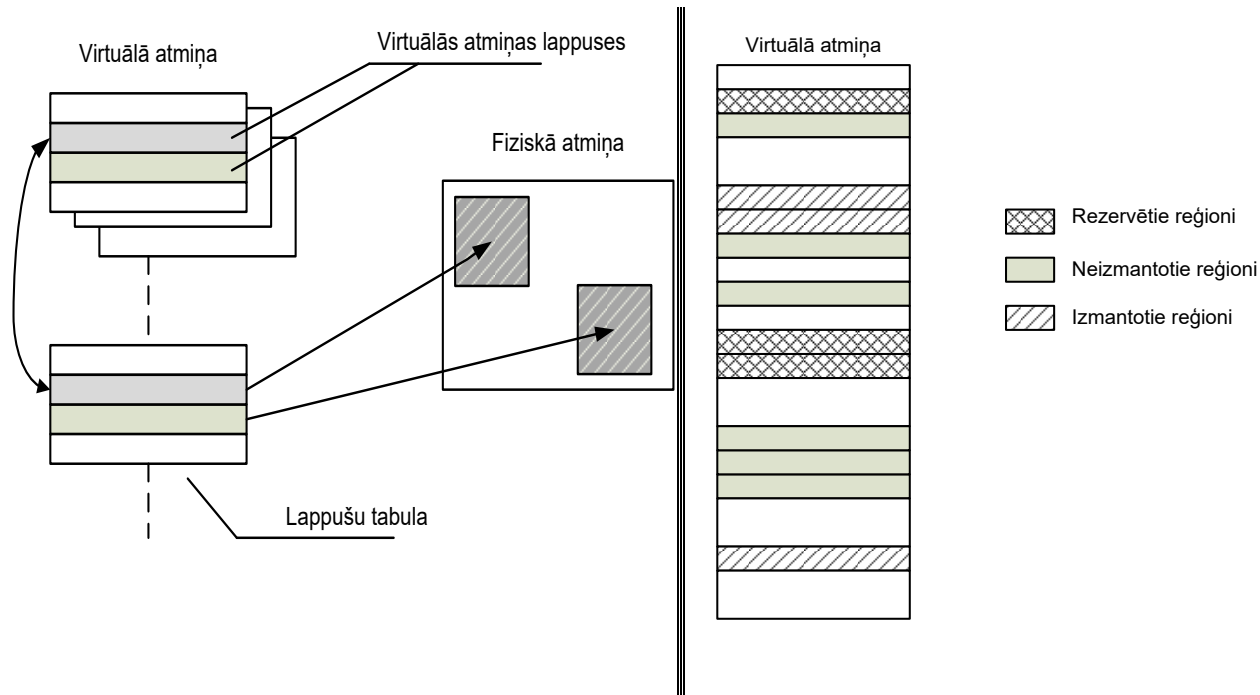


Lappušu apmaiņas jeb lapošanas būtība ir tāda, ka noteiktā laika periodā neizmantojamās operatīvās atmiņas lapas, ar kurām programma pašlaik nestrādā, tiek pārlādētas no operatīvās atmiņas uz cieto disku un nomainītas ar citām lapām, kas dotajā brīdī ir nepieciešamas. Tādā veidā tiek ekonomēta operatīvā atmiņa, turklāt tas ļauj izpildīt vairākus uzdevumus vienlaicīgi.

Lappušu apmaiņa ir efektīva gadījumos, kad notiek operatīvās atmiņas pārlāde, piemēram, ja programma pūlas saņemt vairāk atmiņas nekā ir sistēmā. Virtuālās atmiņas lappušu glabāšanai uz diska tiek izveidota tā saucamā lappuses datne jeb pārvešanas datne (*swap file, paging file*). Lappuses datnes parametriem ir būtiska loma lappušu apmaiņā, piemēram, situācijā, kad lietojumprogrammai darbam ir nepieciešams 600 Mb atmiņas, bet datora operatīvās atmiņas apjoms ir 512 Mb.

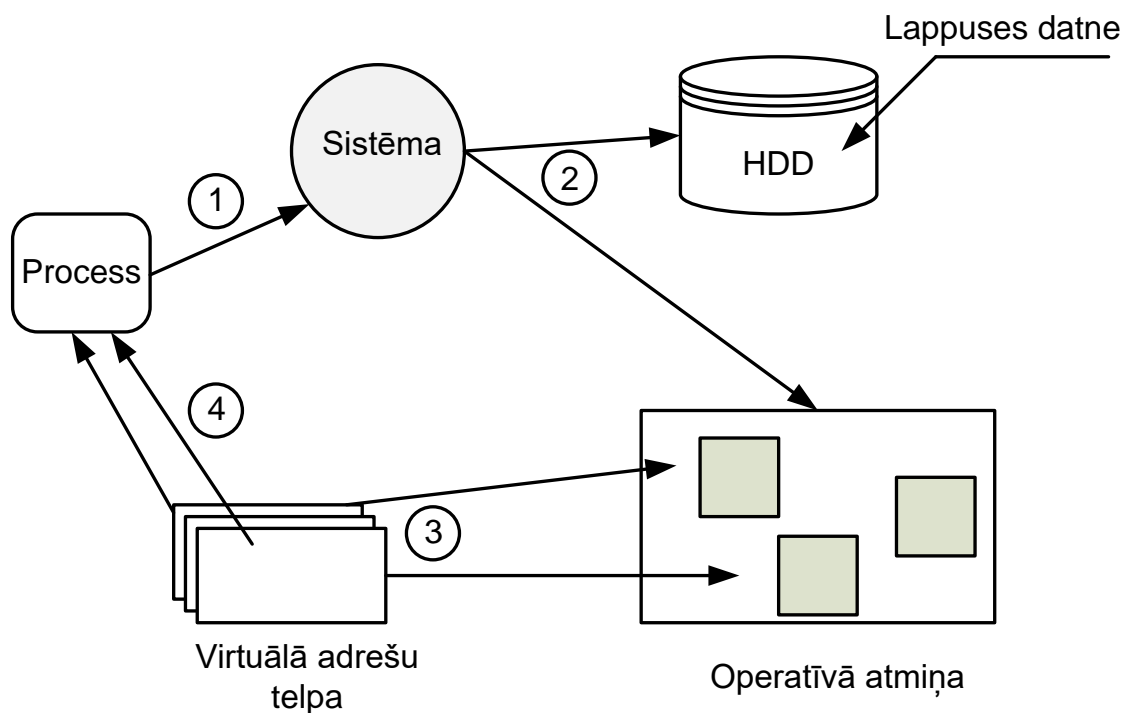
Tādas iespējas nodrošināšanai arī kalpo lappušu apmaiņas mehānisms. Lappušu apmaiņu vada OS. Informācija par lietojumprogrammai izdalītajām virtuālās atmiņas lappusēm glabājas speciālā lappušu tabulā (*page table*) ierakstu veidā. 3.3. attēlā parādītā shēma demonstrē vēl vienu svarīgu lappušu apmaiņas mehānisma aspektu – secīgi izvietotām virtuālās atmiņas lappusēm atbilst tādā pat veidā izvietoti ieraksti lappušu tabulā, lai gan tajā pašā laikā izdalītās fiziskās atmiņas adreses var būt patvaļīgas.

Virtuālo adrešu telpu var iztēloties kā vairāku tipu apgabalu jeb reģionu apvienojumu. Turklāt dažādie reģioni var izvietoties dažādi, t.i., virtuālo adrešu telpa ir fragmentāra (skat. 3.4. attēlu).



Atmiņas rezervēšanas gadījumā sistēma izlīdzina atmiņas reģiona sākumu atbilstoši granularitātes (*granularity*) prasībām. Tā ir sistēmas modularitātes pakāpe (jo augstāka sistēmas granularitāte, jo šī sistēma ir elastīgāka un vairāk piemērojama konkrēta lietotāja vajadzībām). *Intel x86* platformai reģiona adrese izlīdzinās pēc 64 Kb robežas.

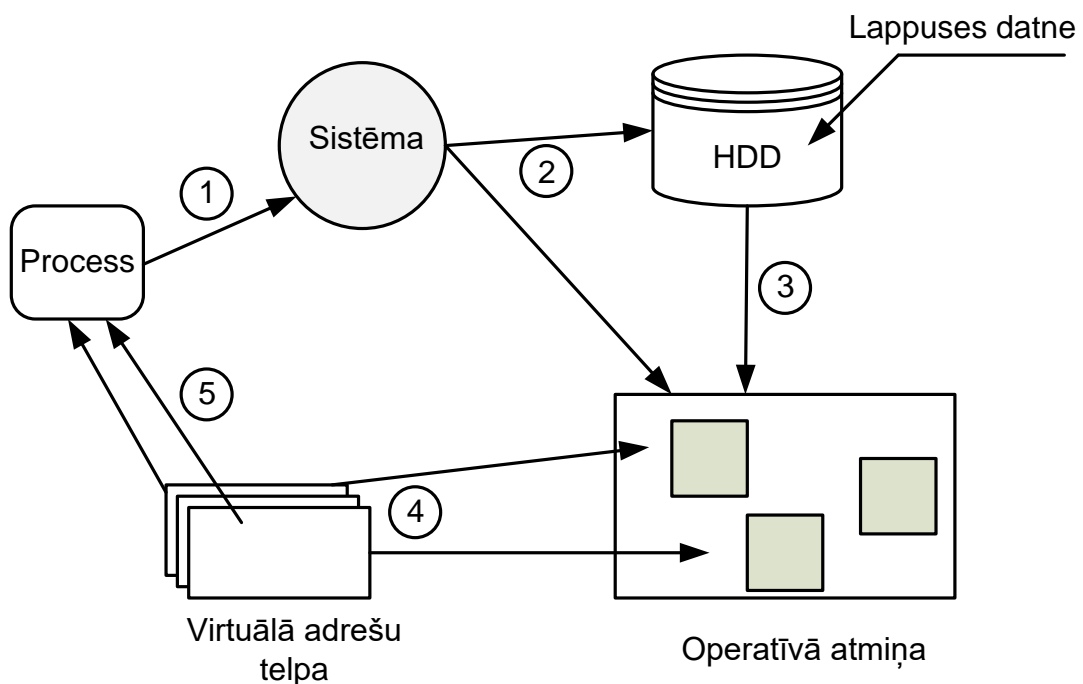
⚠ Gadījumā, kad process pūlas iegūt pieeju datu apgabalam no savas adresu telpas, ir iespējami vairāki notikumi scenāriji [9]. Pirmā varianta shēma parādīta 3.5. attēlā.



Tiek pieņemts, ka procesam nepieciešamie dati atrodas operatīvajā atmiņā. Šajā gadījumā sistēma datu apgabala virtuālo adresi pārveido fiziskajā, un sākas datu apstrāde. Pieejas shēma satur 4 soļus:

- ① Notiek vēršanās pie datu apgabala, kas atrodas procesa virtuālajā adresu telpā, un tāpēc tiek veidots pieprasījums adresu pārvaldniekam un citiem sistēmas dienestiem.
- ② Sistēma nosaka, kur atrodas pieprasītie dati – operatīvajā atmiņā vai lappuses datnē.
- ③ Tā kā tika pieņemts, ka dati atrodas operatīvajā atmiņā, tad procesors veic virtuālās adreses attēlošanu fiziskajā atmiņā.
- ④ Tiek iegūti pieprasītie dati.

Otrajā variantā pieņem, ka procesam nepieciešamie dati atrodas lappuses datnē. Šajā gadījumā pieprasījuma shēma būs sarežģītāka (skat. 3.6. attēlu).



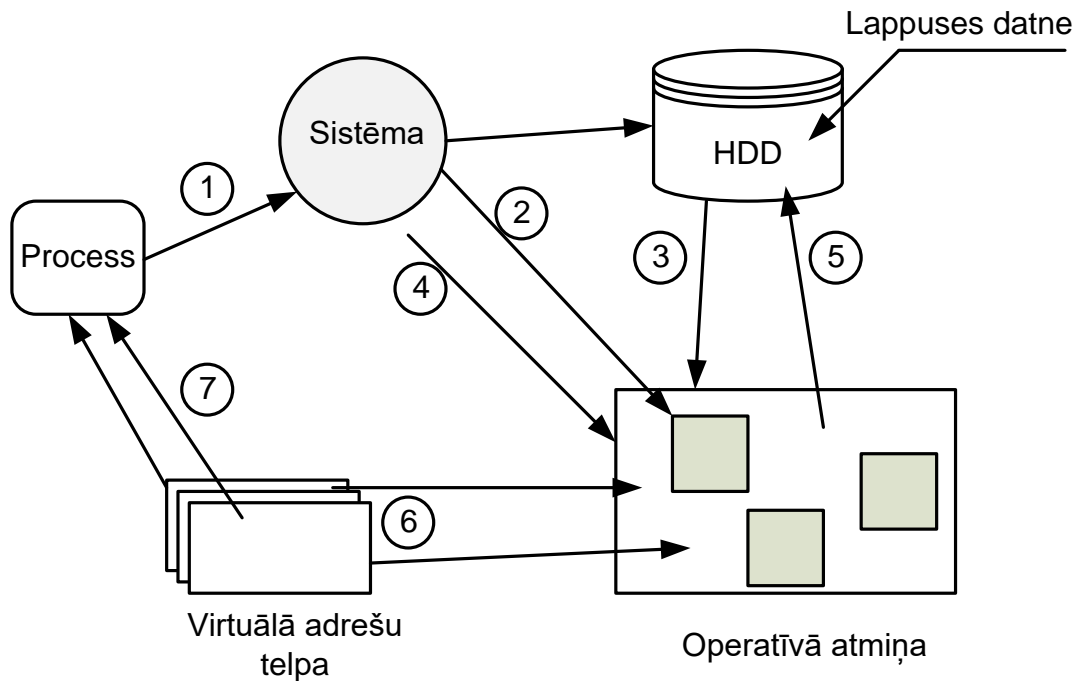
3.6.att. Piekļūšana datiem, kas atrodas lappuses datnē

Pieejas shēma satur 5 soļus:

- ① Notiek vēršanās pie datu apgabala, kas atrodas procesa virtuālajā adresu telpā.
- ② Pieprasījums tiek nodots sistēmai, kura nosaka, kur atrodas pieprasītie dati – operatīvajā atmiņā vai lappuses datnē. Tā kā datu fiziskajā atmiņā nav, procesors ģenerē stāvokli „lappuses kļūda” (*page fault*), kas liek OS meklēt brīvo lappusi operatīvajā atmiņā.
- ③ Atrodot brīvo lappusi, sistēma ielādē tajā datus no lappuses datnes.
- ④ Procesors veic virtuālās adreses attēlošanu operatīvajā atmiņā, kur tika ielādēti dati.
- ⑤ Process iegūst pieeju pieprasītajiem datiem un sākas to apstrāde.

Ja operatīvajā atmiņā nav brīvu lappušu, tad pieprasījuma shēma kļūst vēl sarežģītāka, kas demonstrēts 3.7. attēlā. Shēmā ilustrēts gadījums, kad pieprasāmā lappuse atrodas lappuses datnē un operatīvajā atmiņā trūkst brīvu lappušu.

- ① Notiek vēršanās pie datu apgabala, kas atrodas procesa virtuālajā adresu telpā.
- ② Sistēma nosaka, kur atrodas pieprasītie dati – operatīvajā atmiņā vai lappuses datnē. Tā kā datu fiziskajā atmiņā nav, procesors ģenerē stāvokli „lappuses kļūda”.
- ③ Stāvokļa „lappuses kļūda” gadījumā sistēma sāk meklēt brīvo lappusi operatīvajā atmiņā. Pieņem, ka tāda lappuse nav atrasta.



3.7.att. Pieklūšana datiem, kad nav brīvas operatīvās atmiņas

④ Sistēma sāk fiziskās atmiņas lappušu pārskatu to lappušu noteikšanai, kuras var ielādēt lappuses datnē, lai atbrīvotu atmiņu izpildāmā procesa datiem.

⑤ Sistēma atbrīvo fiziskās atmiņas lappusi, pārrakstot tās saturu lappuses datnē. Turklāt pieprasāmie dati no lappuses datnes tiek pārrakstīti atbrīvojamajai fiziskās atmiņas lappusei.

⑥ Sistēma veic fiziskās atmiņas brīvā apgabala attēlošanu procesa virtuālajā adrešu telpā.

⑦ Sākas datu apstrāde.

Jāatzīmē arī situācija, kad pieprasīto datu nav ne operatīvajā atmiņā, ne lappuses datnē. Šādā situācijā procesors ģenerē stāvokli „pieejas tiesību pārkāpums” (*access violation*), pēc kā tiek izsaukta dotās situācijas apstrādāšanas procedūra (*handler*).

3.2. Atmiņas pārvaldnieks


Atmiņas pārvaldnieks ir realizēts kā OS izpildāmās (*executive*) apakšsistēmas daļa, bet tā programmas modulis atrodas datnē *Ntoskrnl.exe*. Atmiņas pārvaldnieks nodrošina [7,9]:

- virtuālās atmiņas izdalīšanu un atbrīvošanu;
- adrešu telpas izdalīšanu vairāku vienlaicīgu procesu izpildei (*shared memory*);
- datņu attēlošanu atmiņā;
- virtuālās atmiņas izlādēšanu cietajā diskā;
- informācijas par virtuālās atmiņas lappusēm iegūšanu;
- virtuālās atmiņas lappušu aizsardzības atribūtu iestādīšanu.

Atmiņas pārvaldnieku var uzskatīt par vairāku programmu komponentu kopumu:

- sistēmas dienesti, kas veic virtuālās atmiņas izdalīšanu, atbrīvošanu un vadību. Pieklūšana šiem dienestiem tiek realizēta ar *Windows API* saskarnes palīdzību vai ar attiecīgās funkcijas no procesora kodola izsaukšanas palīdzību;
- izņēmuma situāciju un virtuālās atmiņas attēlošanas kļūdu apstrādātājs;
- komponenti, kas atbild par virtuālās atmiņas vadības procesu.

Līdzīgi pārējiem OS sistēmas dienestiem I/O pārvaldnieka servisi ļauj izsaucošajai programmai manipulēt ar procesa virtuālo adresu telpu.

 Daudzi atmiņas pārvaldnieka servisi kļūst pieejami *Windows* API funkciju izsaušanas rezultātā.

API saskarne ietver triju grupu funkcijas, kas ļauj vadīt lietojumprogrammu atmiņu:

- lappušu vadības funkcijas (kopējais nosaukums *VirtualXXX*, kur *XXX* – funkcijas nozīmes sufikss);
- datņu attēlošanas atmiņā vadības funkcijas (*CreateFile-Mapping*, *MapViewOfFile*);
- „kaudzes” (*heap*) vadības funkcijas (kopējais nosaukums *HeapXXX*).

Atmiņas pārvaldnieks nodrošina arī papildus servisu izpildi, piemēram, fiziskās atmiņas izdalīšanu un atbrīvošanu, atmiņas lappušu bloķēšanu DMA operāciju izpildes laikā. Šiem nolūkiem atmiņas pārvaldnieks izmanto funkcijas, kuras sākas ar prefiksu *Mm*. Turklāt atmiņas pārvaldnieks nodrošina atmiņas izdalīšanu no sistēmas „kaudzes”, kas nepieciešama sistēmas dienestu un ierīču draiveru darbam – tiek izmantotas funkcijas, kas sākas ar prefiksu *Ex*. Piemēram, funkcija *ExAllocatePool* nodrošina atmiņas apgabala izdalīšanu no sistēmas pūla (*pool* – dinamiski sadalāmu vienveidīgu objektu, piemēram, vienāda lieluma atmiņas apgabalu, elementāru procesoru, līdzīgu perifērijas ierīču u. c., kopums). Tādas funkcijas pieejamas tikai lietojumprogrammas darbā kodola režīmā, tāpēc tās pārsvarā izmanto sistēmas dienesti un ierīču draiveri.


Par atmiņas pārvaldnieka funkcijām var uzskatīt arī datņu sistēmas objektu attēlošanu atmiņā (*memory mapped files*), kā arī lielu un izretinātu (*sparse*) atmiņas apgabalu optimizāciju. Atmiņas pārvaldnieks ļauj izdalīt procesam vairāk atmiņas nekā to atļauj virtuālā adresu telpa, pateicoties AWE (*Address Windowing Extensions*) mehānismam. Piemēram, izmantojot AWE, adresu telpa 32 bitu sistēmās var pārsniegt 4 Gb.

Sistēmas ielādes brīdī atmiņas pārvaldnieks izveido divus virtuālās atmiņas adresu pūlus: nerezidentu (*paged pool*) un rezidentu (*non-paged pool*).


Rezidentais pūls satur noteiktu virtuālās atmiņas adresu diapazonu, kas garantēti atradīsies fiziskajā atmiņā sistēmas darbības laikā. Dati, kas atrodas rezidentajā pūlā, pieejami jebkurā laika momentā, turklāt lappušu apmaiņa šajā gadījumā nav nepieciešama. Rezidento pūlu pārsvarā izmanto ierīču draiveri pārtraukumu apstrādē, kuriem lappušu apmaiņa nav pieejama.

Nerezidentais pūls satur virtuālās atmiņas adresu diapazonu, kurā ir atļauta lappušu apmaiņa. To izmanto draiveri, kas neizmanto pārtraukumu apstrādi.

Abi pūli izvietoti virtuālās adresu telpas sistēmas daļā, bet tajos izmantojamās atmiņas izdalīšanu un atbrīvošanu veic funkcijas *ExAllocatePool* un *ExFreePool*.

 Vienprocesoru sistēmās tiek izveidoti 3 nerezidentie pūli, bet daudzprocesoru sistēmās – 5.

Vairāki nerezidentie pūli ļauj samazināt bloķēšanas gadījumu skaitu atmiņas izdalīšanas un atbrīvošanas funkciju darbības rezultātā. Rezidento un nerezidento pūlu sākotnējais apjoms atkarīgs no datorsistēmas fiziskās atmiņas apjoma.

 Svarīgs aspekts darbā ar virtuālo atmiņu ir atmiņas aizsardzība.

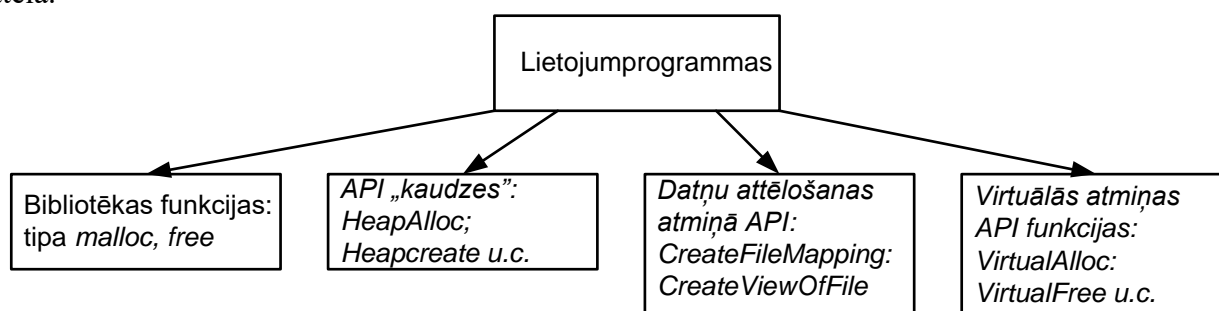
Katram procesam OS vidē tiek izdalīta privāta (*private*) adresu telpa. Tas nozīmē to, ka datu plūsmā šī procesa ietvaros ir pieejama tikai tā atmiņa, kas izdalīta šim procesam. Jaunākās OS *Windows* nodrošina atmiņas aizsardzību četros veidos:

- 1) visas sistēmas datu struktūras un atmiņas pūli pieejami tikai kodola režīmā – lietojumprogrammu datu plūsmām nav pieejas šīm atmiņas lappusēm. Ja tomēr no lietojumprogrammas puses notiek mēģinājums piekļūt šīm lappusēm, tiek ģenerēta kļūda, ko atmiņas pārvaldnieks interpretē kā piekļuves pārkāpšanas kļūdu (*Access violation*). Salīdzinājumam - OS *Windows 98* ļauj lietojumprogrammām piekļūt

atsevišķām sistēmas adrešu telpas lappusēm, kas daudzos gadījumos var izraisīt sistēmas krahu;

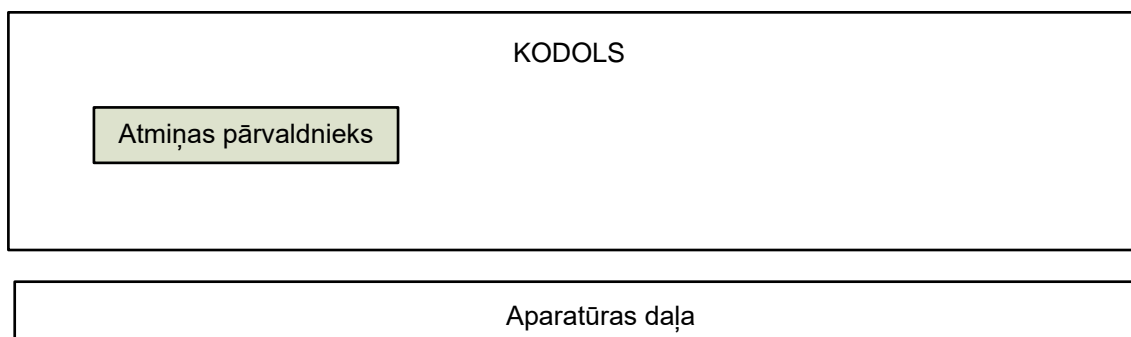
- 2) katram procesam ir sava adrešu telpa, kas aizsargāta pret citu procesu piekļuves tajā. Atsevišķos gadījumos process var speciāli atļaut citam procesam izmantot savu adrešu telpu, piemēram, ar funkciju *ReadProcessMemory* vai *WriteProcessMemory* palīdzību;
- 3) visi procesori nodrošina attiecīgu aizsardzību aparatūras līmenī. Piemēram, lappuses, kuros atrodas programmas kods, tiek atzīmētas kā „tikai lasīšanai” (*read only*) un nevar tikt modificētas no lietojumprogrammu puses;
- 4) tiek izmantots pieejas kontroles saraksts ACL (*Access-Control List*), kurš tiek pārbaudīts katru reizi, kad process pūlas piekļūt resursiem. Turklāt piekļuvi atmiņai saņem iegūst tikai tie procesi, kam ir iestatīti attiecīgie atribūti.

OS atmiņas vadības mehānisms, kas pieejams lietojumprogrammām, ir parādīts 3.8. attēlā.



Lietotāja režīms

Kodola režīms



3.8.att. Lietotāja saskarne



3.1. piemērs. Virtuālās atmiņas apjoma noteikšana

Piemērā tiek izmantota funkcija *GetSystemInfo*.

```

// Virtualas atminas apjoms
#include "stdafx.h"
#include <stdio.h>
#include <windows.h>

int _tmain(int argc, _TCHAR* argv[])
{
    SYSTEM_INFO SysInfo;
    GetSystemInfo(&SysInfo);
    printf("Info par datoru: \n");
    printf("Procesoru skaits: %u\n",
        SysInfo.dwNumberOfProcessors);
}
  
```

```

printf("Lappuses apjoms: %u\n", SysInfo.dwPageSize);
printf("Procesora tips: %u\n", SysInfo.dwProcessorType);
getchar();
return 0;
}

```

Rezultāts

Procesoru skaits: 1
 Lappuses apjoms: 4096
 Procesora tips: 586



3.2. piemērs. Informācijas iegūšana par pieejamās fiziskās atmiņas apjomu

Piemērā tiek izmantota funkcija *GlobalMemoryStatus*.

```

// Info par pieejamo fizisko atminu
#include "stdafx.h"
#include <stdio.h>
#include <windows.h>

int _tmain(int argc, _TCHAR* argv[])
{
    MEMORYSTATUS memStat;
    GlobalMemoryStatus(&memStat);
    printf("Pieejama fiziskaa atmina: %d Kb\n", memStat.dwAvailPhys / 1024);
    getchar();
    return 0;
}

```

Rezultāts

Konkrētajā datorā tika iegūts skaitlis: 208 300 Kb.



3.3. piemērs. Funkcijas *VirtualProtect* iespējas

```

// VirtualProtect izmantosana
#include "stdafx.h"
#include <windows.h>
#define REGSIZE 1024

int _tmain(int argc, _TCHAR* argv[])
{
    DWORD oldProtect;
    char *s1="Rinda Nr. 1 tiks kopeeta uz Virtualo atminu ";
    char *s2="Rinda Nr. 2 tiks kopeeta uz Virtualo atminu ";

    char *p=(char*)VirtualAlloc(NULL, REGSIZE, MEM_COMMIT, PAGE_READWRITE);
    memcpy(p, s1, strlen(s1));
    printf("%s\n", p);

    VirtualProtect(p, REGSIZE, PAGE_READONLY, &oldProtect);
    memcpy(p, s2, strlen(s2));
    printf("%s\n", p);

    VirtualFree(p, 0, MEM_RELEASE);
    getchar();
    return 0;
}

```

Komentāri

Funkcijai *VirtualProtect* ir šāda sintakse:
 BOOL VirtualProtect (LPVOID lpAddress,
 SIZE_T dwSize,
 DWORD flNewProtect,
 PDWORD lpflOldProtect);

- lpAddress – virtuālās atmiņas reģiona adrese, kurai jāmaina pieejas atribūti;
- dwSize – reģiona apjoms baitos;
- flNewProtect – iestatāmais pieejas atribūts;
- lpfloldProtect – norāde uz mainīgo, kur glabāsies iepriekšējie pieejas atribūti. Ja šis parametrs ir NULL vai norāda uz neeksistējošu mainīgo, funkcija paziņo par kļūdu.

Programma kopē datus no viena atmiņas apgabala citā un attēlo kopējamus datus displejā. Par sākotnējiem datiem kalpo divas rindas – s1 un s2. Programma izdala 1024 baitu lielu virtuālās atmiņas apgabalu ar piekļuves tiesībām gan lasīšanai, gan ierakstīšanai – uz to norāda mainīgais p:

```
char *p=(char*)VirtualAlloc(NULL, REGSIZE, MEM_COMMIT,
                             PAGE_READWRITE);
```

Tālāk rindas s1 simboli tiek iekopēti atmiņas apgabalā, uz kuru norāda p:

```
memcpy(p, s1, strlen(s1));
```

Displeja ekrānā redzams atmiņas apgabala saturs: printf("%s\n", p);

Tad tiek mainīts mainīgajā p dotais piekļuves atmiņai atribūts – tā kļūst pieejama tikai lasīšanai: VirtualProtect(p, REGSIZE, PAGE_READONLY, &oldProtect);

Funkcijas memcpy(p, s2, strlen(s2)); izpilde būs neveiksmīga, jo ierakstīt datus atmiņā, uz ko norāda mainīgais p, nav atļauts. Tiks izdota vesela virkne paziņojumu par kļūdu. Nolasīt datus no šī paša apgabala tomēr ir atļauts. Lai par to pārliecinātos – jāatzīmē programmas rinda kā komentārs (// memcpy(p, s2, strlen(s2));) un atkārtoti jāpalaiž programma izpildei.

3.3. Virtuālā atmiņa un lietojumprogrammu veiktspēja

Atmiņas izmantošanas optimizācija ietver sevī daudz jautājumu, uz kuriem bieži vien nav viennozīmīgas atbildes. Tomēr pastāv vesela virkne paņēmieni, ar kuru palīdzību var panākt pielietojumu veiktspējas palielināšanu – attiecīgā veidā nokonfigurējot to vai citu atmiņas vadības apakšsistēmas komponentu.



Viens no iespējamajiem veidiem atmiņas izmantošanas optimizēšanā ir procesa lappušu rezidentā pūla izmantošanā (*working set*). Procesā adresu rezidentais pūls ir operatīvajā atmiņā izvietotās lappuses, pie kurām process var piekļūt. Šīs lappuses pastāvīgi atrodas operatīvajā atmiņā un pieejamas procesam bez lappušu pārslēgšanas (*page fault*). Procesā rezidentajam pūlam ir noteikts minimālais un maksimālais apjoms (lappusēs), kas arī nosaka lappušu apmaiņas efektīvu izmantošanu.

OS rezidento pūlu apjomus procesiem nosaka pēc FIFO principa – „pirmais atnāci – pirmais tiek apkalpots”. Piemēram, ja lietojumprogramma nosaka (sistēmā ar 640 Mb operatīvo atmiņu) minimālo rezidentā pūla apjomu 400 Mb, tad nākamajai lietojumprogrammai, kas pieprasīs 400 Mb operatīvās atmiņas, tiks atteikts resursu izdalīšanās.



Pēc noklusējuma jebkurš process saņem minimālajam rezidentajam pūlam 50 lappuses, maksimālajam – 345 lappuses.

Programmētājam ir iespējas palielināt lietojumprogrammas veiktspēju, ja pareizi izvēlēsies rezidentā pūla apjomus. Rezidentā pūla minimālo un maksimālo apjomu var uzdot ar funkcijas *SetProcessWorkingSetSize* palīdzību [9].

Lappuses kļūdas gadījumā tiek pārbaudīta procesa rezidentā pūla maksimālā apjoma vērtība un brīvās atmiņas apjoms sistēmā. Ja apstākļi pieļauj, atmiņas pārvaldnieks procesam palielina rezidentā pūla apjomu. Ja brīvās atmiņas ir maz, atmiņas pārvaldnieks iniciē lappušu apmaiņu, nepievienojot jaunas lappuses rezidentajā pūlā.

OS pēc iespējas vairāk pūlas atbrīvo operatīvo atmiņu uz lappušu ierakstīšanas lappuses datnē rēķina. Taču ar to arī ir problēmas – jo intensīvāk parādās tādas lappuses, jo vairāk operatīvās atmiņas nepieciešams to apstrādei. Gadījumos, kad fiziskās atmiņas apjoms sāk samazināties,

atmiņas pārvaldnieka komponente – rezidento pūlu pārvaldnieks (*working set manager*) – iniciē automātisku rezidento pūlu samazināšanu, kas dod iespēju palielināt pieejamās operatīvās atmiņas apjomu.

Rezidento pūlu pārvaldnieks analizē brīvās atmiņas apjomu un lemj, kāds rezidentais pūls ir jāsamazina. Šim nolūkam tiek noteikti rezidentie pūli, kas pārsnieguši minimālās vērtības, un tiek izdzēstas liekās lappuses. Rezidento pūlu pārvaldnieks nosaka biežumu, pēc kura tiek pārbaudīti rezidentie pūli, kā arī veic prognozi par turpmāko procesu apstrādes secību. Piemēram, procesi, kas strādā ar lielu lappušu skaitu, pie kurām sen nav bijis pieejas, tiks pārbaudīti pirmām kārtām. Procesi, kas prasa lielus atmiņas resursus, neveic skaitļošanas darbības vai atrodas gaidīšanas stadijā, tiks pārbaudīti agrāk nekā procesi ar mazākiem resursiem, bet intensīvāku datu apmaiņu. Protams, ka process, kas izpildās fona režīmā, tiks pārbaudīts pēdējais.

Ja rezidento pūlu pārvaldnieks atrod procesus, kuri izmanto lielāku apjomu nekā minimālais rezidentā pūla apjoms, tas meklē lappuses, kuras var izdzēst no pūla. Tā turpinās tik ilgi, kamēr pūlu pārvaldnieks nesaņems nepieciešamo sistēmas fiziskās brīvās atmiņas apjomu.

Vienprocesoru sistēmās rezidento pūlu pārvaldnieks cenšas izdzēst no pūla lappuses, pie kurām sen nav bijis pieejas. Šajā nolūkā tiek pārbaudīts pieejas bits tabulā PTE (*Process Table Entry*), un, ja bits ir „nomests” (kas nozīmē, ka kopš pēdējās rezidentā pūla pārbaudes lappusei nebija pieejas), tad šī lappuse tiek izmesta no rezidentā pūla.

Rezidentajam pūlam apjomu var mainīt arī lietojumprogrammā [9].



3.4. piemērs. Rezidentā pūla minimālā un maksimālā apjoma mainīšana

```
// Rezidentā pūla adrešu apjoma maiņa
#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
#include <psapi.h>
    DWORD processId;
    DWORD minSet, maxSet;

int main(void)
{
    HANDLE hProcess;
    PROCESS_MEMORY_COUNTERS pmc;
    processId=GetCurrentProcessId();
    hProcess = OpenProcess(PROCESS_QUERY_INFORMATION |
        PROCESS_VM_READ | PROCESS_SET_QUOTA,
        FALSE,
        processId);

    if (hProcess == NULL)
        return 0;
    if (GetProcessMemoryInfo(hProcess, &pmc, sizeof(pmc)))
printf("\n\tRezidentaa puula apjoms: %d Lapas\n", pmc.WorkingSetSize / 4096);

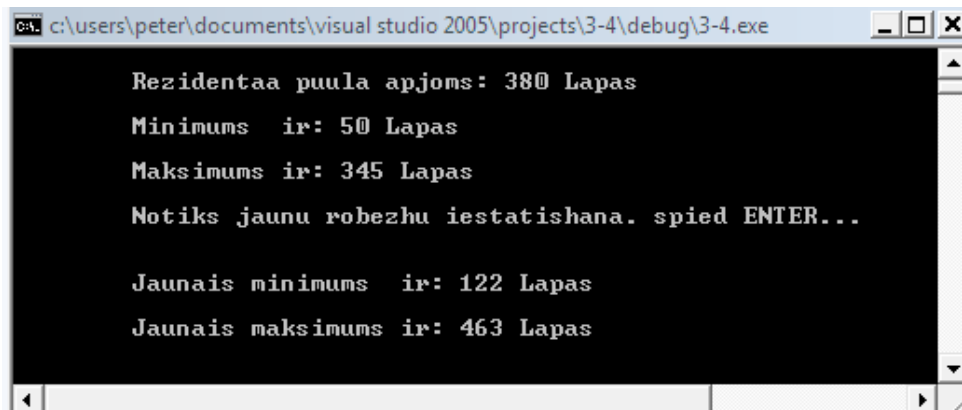
    GetProcessWorkingSetSize(hProcess, (PSIZE_T)&minSet, (PSIZE_T)&maxSet);
        printf("\n\tMinimums ir: %d Lapas\n", minSet / 4096);
        printf("\n\tMaksimums ir: %d Lapas\n", maxSet / 4096);
        printf("\n\tNotiks jaunu robezhu iestatishana. spied ENTER...\n");
    getchar();

    SetProcessWorkingSetSize(hProcess, 500000, 1900000);
    GetProcessWorkingSetSize(hProcess, (PSIZE_T)&minSet, (PSIZE_T)&maxSet);
        printf("\n\tJaunais minimums ir: %d Lapas\n", minSet / 4096);
        printf("\n\tJaunais maksimums ir: %d Lapas\n", maxSet / 4096);
    getchar();

    CloseHandle(hProcess);
    return 0;
}
```

! Projektam ir jāpievieno bibliotēka *psapi.lib*.

Programmas darba rezultāts parādīts attēlā:



```
c:\users\peter\documents\visual studio 2005\projects\3-4\debug\3-4.exe

Rezidentaa puula apjoms: 380 Lapas
Minimums ir: 50 Lapas
Maksimums ir: 345 Lapas
Notiks jaunu robezhu iestatishana. spied ENTER...

Jaunais minimums ir: 122 Lapas
Jaunais maksimums ir: 463 Lapas
```

Komentāri

Lai varētu manipulēt ar procesa virtuālo atmiņu, vispirms ir jāiegūst procesa identifikators un deskriptors (*descriptor*) jeb procesa atslēgas vārds (šajā gadījumā *hProcess*). Ar funkcijas `GetProcessMemoryInfo` palīdzību tiek iegūta informācija par procesa pašreizējo stāvokli.

Lietojumprogrammas veikspējas uzlabošanas nolūkā tiek palielināta procesa rezidentā pūla minimālā un maksimālā vērtība, izmantojot funkciju

```
SetProcessWorkingSetSize(hProcess, 500000, 1900000,
```

ar kuras palīdzību rezidentā pūla minimālais izmērs tiek uzdots 500000 baitu apjomā, bet maksimālais – 1,9 Mb.

Kā redzams no programmas darbības rezultātiem, atmiņas pārvaldnieks ir dinamiski pamainījis pūla robežvērtības. Taču tas negarantē to, ka pieprasītais fiziskās atmiņas apjoms tiks rezervēts šim procesam lietojumprogrammas izpildes laikā. Situācijās, kad process būs gaidīšanas režīmā vai pieejamās atmiņas apjoms strauji samazinās (*low memory*), atmiņas pārvaldnieks var mainīt rezidentā pūla apjomu, jo atmiņas pārvaldnieka darbības galvenā prioritāte ir OS normāla funkcionēšana.

Ja katram procesam ir savs adrešu rezidentais pūls, tad arī OS ir savs (vienīgais) rezidentais pūls. Šajā pūlā ietilpst:

- sistēmas kešatmiņas lappuses (*system cache pages*);
- nerezidentais (lappušu) adrešu pūls (*paged pools*);
- nerezidentais kods un dati, kas atrodas datnē *Ntoskrnl.exe*;
- ierīču draiveru nerezidentais kods un dati.

Sistēmas rezidentā pūla minimālais un maksimālais apjoms tiek iestatīts OS inicializācijas laikā un ir atkarīgs no datorsistēmas fiziskās atmiņas apjoma (izskaitļotās vērtības tiek ievestas sistēmas mainīgajos).

! Būtiska loma virtuālās atmiņas izmantošanā un sistēmas darbā kopumā ir lappušu datnei – jo biežāk sistēmai nākas kopēt atmiņas lappuses datnē vai nolasīt tās atpakaļ atmiņā, jo lielāka diska iekārtas noslodze (tādējādi lēnāk strādā OS). Vienkāršākā izeja no šādas situācijas ir papildus atmiņas moduļu pievienošana, kas ļaus samazināt pieejas diskam pieprasījumu skaitu.

OS sākotnējās ielādes laikā sesiju pārvaldnieks (*Session Manager*) nolasa no sistēmas reģistra lappušu datnes parametrus:

```
(HLM\SYSTEM\CurrentControlSet\Control\SessionManager\MemoryManagement\PagingFiles).
```

Šeit uzdoti lappušu datņu nosaukumi un izmēri (OS *Windows* var operēt ar maksimums 16 lappušu datnēm).

Ar x86 savietojamās sistēmās katrai lappušu datnei maksimālais apjoms var būt 4095 Mb. Lappušu datni pastāvīgi izmanto OS (nolasīšanai un ierakstīšanai), tāpēc to nevar defragmentēt ar standarta OS līdzekļiem.

Tā kā lappušu datne satur procesu un kodola virtuālās atmiņas datus, tad drošības nolūkos var izdzēst datnes saturu datorsistēmas izslēgšanas brīdī. Šim nolūkam sistēmas reģistra parametram *HLM\SYSTEM\CurrentControlSet\Control\SessionManager\MemoryManagement\ClearPageFileAtShutdown* iestāda vērtību 1.

Dažādas OS dažādi manipulē ar lappušu datni [9]. Piemēram, ja *Windows 2000* lappušu datnes nav, tad sistēma pēc noklusējuma izveido datni 20 Mb apjomā. OS *Windows XP* un *Windows Server 2003* izveido lappušu datni uz pieejamās fiziskās atmiņas analīzes pamata (tās apjoms parādīts 3.1. tabulā).

3.1. tabula

Lappuses datnes apjoms (pēc noklusējuma)

Fiziskās atmiņas apjoms (RAM)	Minimālais apjoms	Maksimālais apjoms
< 1 Gb	1.5 x RAM	3 x RAM
>1 Gb	1 x RAM	3 x RAM

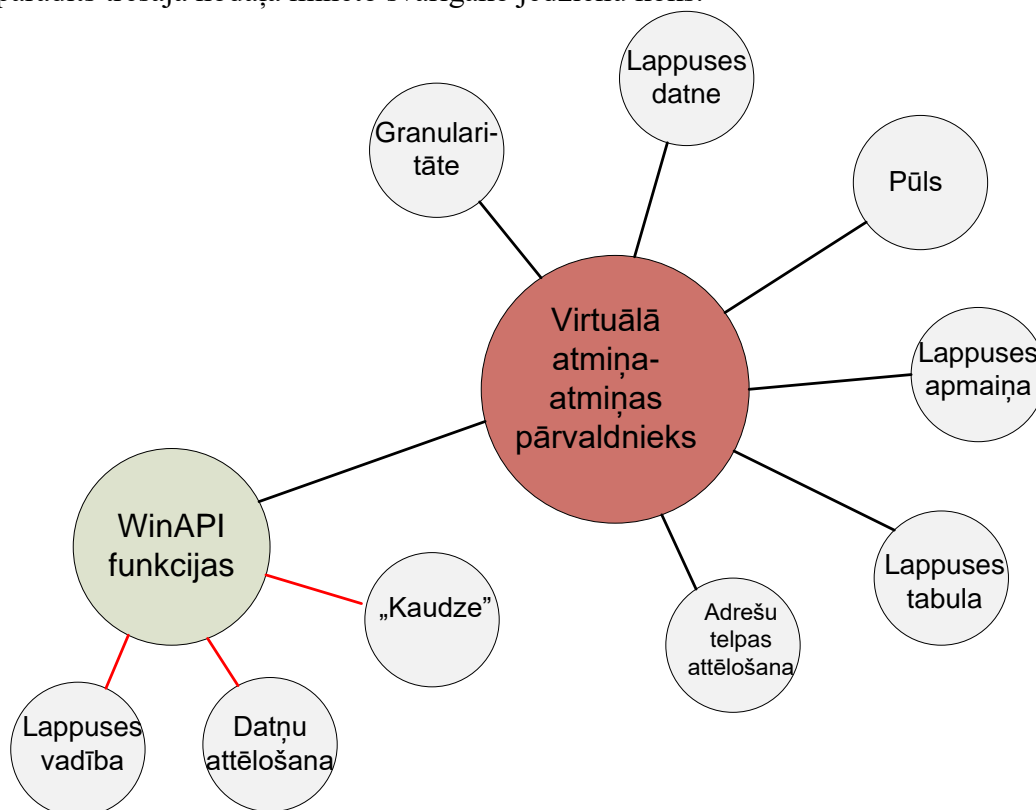
Ja lappušu datnes apjoms ir pārāk liels, tas nekādi neiespaido sistēmas veiktspēju. Ja tās apjoms pārāk mazs – var tikt izdots paziņojums „*System running low on virtual memory*”. Tādos gadījumos ir jāpārbauda – vai kāda no programmām neizmanto ļoti lielu atmiņas apjomu. Iespējams arī cits cēlonis – kāds no kodola draiveriem izmanto pārāk daudz atmiņas no sistēmas nerezidentā pūļa.

NODAĻAS KOPSAVILKUMS

- Operētājsistēmas izmanto virtuālās atmiņas koncepciju.
- Virtuālā atmiņa - nepārtraukta adrešu telpa, ar kuru var darboties OS un izpildāmie procesi neatkarīgi no tā, kādā veidā realizēta fiziskā atmiņa.
- Virtuālās atmiņas vadību veic atmiņas pārvaldnieks.
- Atmiņas pārvaldnieks nodrošina:
 - virtuālās atmiņas izdalīšanu un atbrīvošanu;
 - adrešu telpas izdalīšanu vairāku vienlaicīgu procesu izpildei;
 - datņu attēlošanu atmiņā;
 - virtuālās atmiņas izlādēšanu cietajā diskā;
 - informācijas iegūšanu par virtuālās atmiņas lappusēm;
 - virtuālās atmiņas lappušu aizsardzības atribūtu iestatīšanu.
- Lappušu apmaiņas būtība ir tāda, ka noteiktā laika periodā neizmantojamās operatīvās atmiņas lapas, ar kurām programma pašlaik nestrādā, tiek pārlādētas no operatīvās atmiņas uz cieto disku un nomainītas ar citām lapām, kas pašreizējā brīdī ir nepieciešamas.
- Informācija par lietojumprogrammai izdalītajām virtuālās atmiņas lappusēm glabājas speciālā lappušu tabulā ierakstu veidā.
- Eksistē vairāki scenāriji, kad procesam ir nepieciešamība piekļūt atmiņas apgabalam no savas adrešu telpas.
- Sistēmas ielādes brīdī atmiņas pārvaldnieks izveido divus virtuālās atmiņas adrešu pūļus: nerezidento un rezidento.
- Vienprocesoru sistēmās tiek izveidoti 3 nerezidentie pūļi, bet daudzprocesoru sistēmās – 5.
- Pastāv virkne paņēmieni, ar kuru palīdzību var panākt pielietojumu veiktspējas palielināšanu - attiecīgā veidā nokonfigurējot to vai citu atmiņas vadības apakšsistēmas komponentu.
- Pēc noklusējuma jebkurš process saņem minimālajam rezidentajam pūļam 50 lappuses, maksimālajam – 345 lappuses.

- Arī operētājsistēmai ir savs (vienīgais) rezidentais pūls, kurā ietilpst:
 - sistēmas kešatmiņas lappuses;
 - nerezidentais (lappušu) adrešu pūls;
 - nerezidentais kods un dati, kas atrodas datnē *Ntoskrnl.exe*;
 - ierīču draiveru nerezidentais kods un dati.

3.9. attēlā parādīts trešajā nodaļā minēto svarīgāko jēdzienu koks.



3.9.att. Trešās nodaļas svarīgāko jēdzienu koks

Uzdevumi un jautājumi patstāvīgam darbam

1. Kādās situācijās iespējams paziņojums par virtuālās atmiņas trūkumu?
2. Kā pievienot papildus virtuālo atmiņu?
3. Vai reālā laika OS ir lietderīga virtuālās atmiņas izmantošana?
4. Ja virtuālās lappuses izvietotas patvaļīgā secībā, vai tas jūtami ietekmē lietojumprogrammas veiktspēju?
5. Dažās arhitektūrās atmiņas pārvaldnieks virtuālajām lappusēm iestata mainīgu lappušu apjomu. Vai tas ir lietderīgi?
6. Kādos gadījumos var atteikties no lappušu datnes?
7. Modelēt situāciju, kad lietojumprogrammai nepieciešamie dati nav atrodami ne operatīvajā atmiņā, ne lappuses datnē.
8. Kā lietojumprogrammai iegūt pieeju citā procesā izmantojamajai atmiņai?
9. Kādos gadījumos iespējams attīrīt vai defragmentēt lappušu datni?
10. Noskaidrot kāda datora fiziskās un virtuālās atmiņas apjomu.

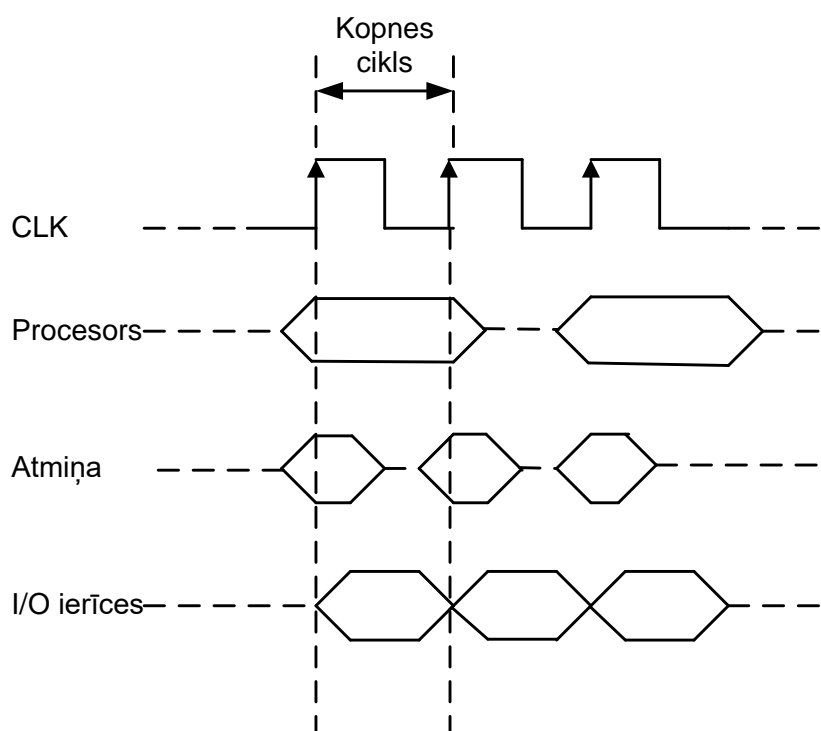
4. INTEL PROCESORU ORGANIZĀCIJA UN IESPĒJAS

Nodaļā ir apskatītas galvenokārt Intel Pentium procesoru arhitektūras īpatnības [9,19], jo tieši šīs klases procesori visbiežāk tiek izmantoti IA-32 datorsistēmās. Tomēr tas ļaus bez grūtībām izprast arī citu izstrādātāju procesoru (piemēram, AMD) funkcionēšanu [16]. Tas izskaidrojams ar to, ka visiem mūsdienu procesoriem ir līdzīga arhitektūra un tie izmanto kopīgu pieeju datu apstrādē.

Nodaļā tiks apskatīta arī procesoru arhitektūra no programmētāja viedokļa – komandu un datu plūsmu organizācija, saskarne ar sistēmas kopni un datu sinhronizācija.

4.1. Mūsdienu procesoru funkcionēšanas pamati

Visu datorsistēmas iekārtu darbs tiek sinhronizēts ar takts frekvences impulsiem, ko rada procesora un mātesplates elektroniskās shēmas [9]. Shematiski to var attēlot ar 4.1. attēlā redzamo shēmu.



4.1. att. Datorsistēmas sinhronizācija

Sinhronizējošo impulsu virkne attēlā apzīmēta ar CLK. Par takts impulsu avotu mūsdienu datorsistēmās kalpo atsevišķa ierīce.

Visas laika raksturlielumu atkarības datorsistēmās tiek mērītas kopnes ciklos.

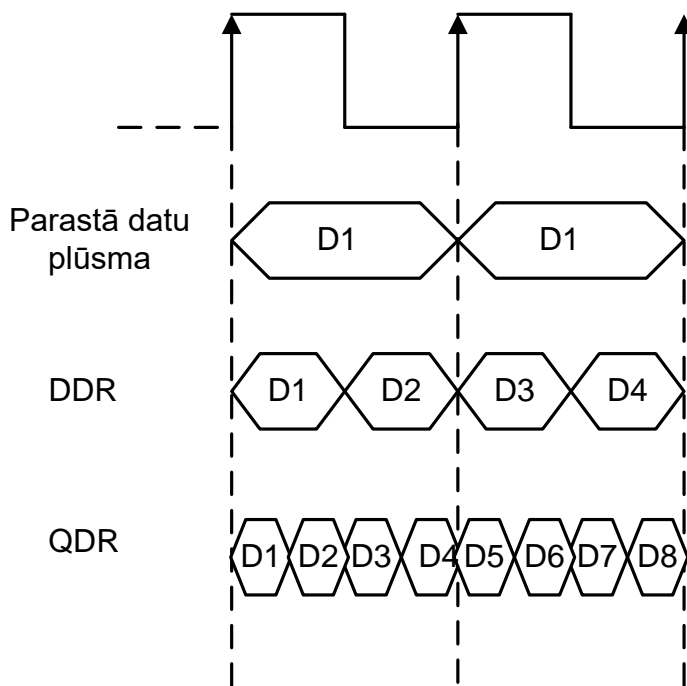
i Kopnes cikls – sinhroimpulsu atkārtošanās periods, datorsistēmās tas ir laika intervāls starp divām secīgām sinhroimpulsu frontēm (sk. 4.1. att.).

Otrs būtisks datora darba aspekts saistīts ar to, ka dažādām ierīcēm ir dažāda ātrdarbība. Visātrdarbīgākā iekārta datoros ir procesors, tad seko atmiņa un I/O ierīces. Ar aparātūras ātrdarbību saprot sinhronizācijas robežfrekvenci, ar kuru var strādāt ierīce. Piemēram, pie kopnes takts frekvences vērtības 200 MHz procesoram *Intel Pentium 4*, kurš strādā ar iekšējo takts frekvenci 2,4 GHz, jāreizina iekšējā frekvence ar noteiktu koeficientu – šajā gadījumā 12 ($200 \times 12 = 2400$).

⚠ Procesora iekšienē visas operācijas tiek izpildītas ar procesora takts frekvenci, t.i., visām ierīcēm procesorā jāstrādā sinhroni ar šo frekvenci. Piemēram, procesoram ar iekšējo darba frekvenci 2,4 GHz visām izlases, dekodēšanas un izpildes komandām jāizpildās sinhroni ar dotās frekvences takts impulsiem. Tajā pašā laikā visas ārējās (attiecībā pret procesoru) ierīces sinhronizēsies ar frekvenci 200 MHz. Tāda atšķirība procesora ātrdarbībā pret ārējām ierīcēm rada būtiskas problēmas, kas ietekmē datorsistēmas veiktspēju kopumā. Attēlā uzdotā shēma demonstrē ārējās (attiecībā pret procesoru) kopnes darbu.

Atšķirības dažādu ierīču ātrdarbībā veicina datu apstrādes aizturi, it īpaši relatīvi lēnās operatīvās atmiņas dēļ. Lai no tā izvairītos, tiek izmantotas vairākas pieejas.

Pirmā pieeja balstās uz iespējami lielāka datu apjoma pārraidi vienā kopnes ciklā. Piemēram, AMD procesori var dubultot pārraidāmo datu daudzumu (DDR – *Dual Data Rate*), bet Intel Pentium procesori – pārraidīt četrkārtotu datu daudzumu vienā kopnes ciklā (sk. 4.2. attēlu).



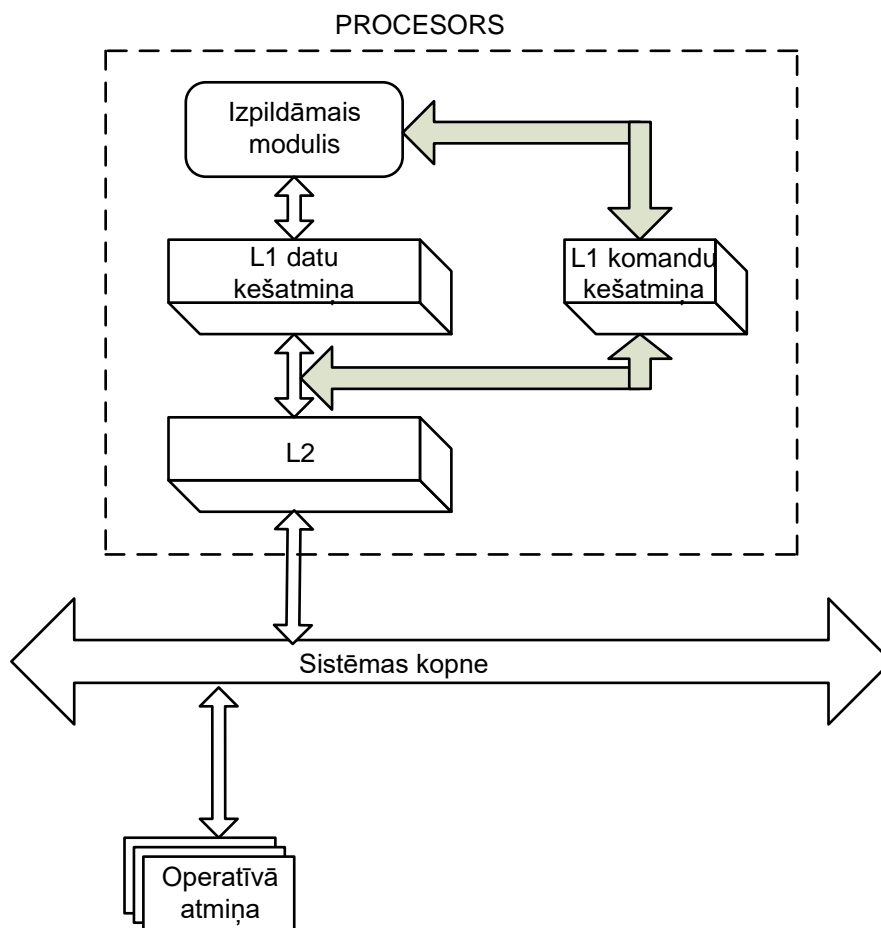
4.2. att. Datu apmaiņa ar atmiņu dažādiem procesoriem

Otrās pieejas pamatā - procesors veic datu kešdarbi. Kešatmiņa ir ātra operatīvā atmiņa, kas izvietota procesora kristālā un paredzēta komandu un datu laicīgai uzglabāšanai apmaiņas procesos ar operatīvo atmiņu.

Visiem mūsdienu procesoriem ir komandas/datu kešatmiņa, kas dod iespēju veikt iepriekšēju informācijas izlasi no operatīvās atmiņas, ar to samazinot procesora gaidīšanas laiku. Parasti uz procesora kristāla izvietojas triju veidu kešatmiņa (sk. 4.3. attēlu).

Komandu/datu kešatmiņa realizēta procesoros kā ātrdarbīga statiska atmiņa. Kešatmiņu pieņemts apzīmēt ar simbolu L, aiz kura norāda kešatmiņas līmeni (1,2,3). Kešatmiņas līmenis raksturo tās fizisko attālumu no procesora izpildāmajiem moduļiem un ātrdarbību. Piemēram, pirmā līmeņa komandu kešatmiņa tiek apzīmēta kā L1 un izvietota blakus procesora datu izlases un dekodēšanas iekārtām. Šī ir visātrdarbīgākā kešatmiņa, jo no tās notiek komandu izlase ar procesora iekšējo frekvenci. Līdzīga ātrdarbība piemīt L1 datu kešatmiņai, kurā uzglabājas ar komandām apstrādājamie dati.

Vispārējās nozīmes kešatmiņa L2 izvietota blakus sistēmas kopnes saskarnei un paredzēta datu apmaiņai ar datorsistēmas operatīvo atmiņu. L2 ātrdarbība ir daudz augstāka nekā operatīvajai atmiņai (kura ir dinamiska tipa), bet mazāka par L1 kešatmiņas ātrdarbību.



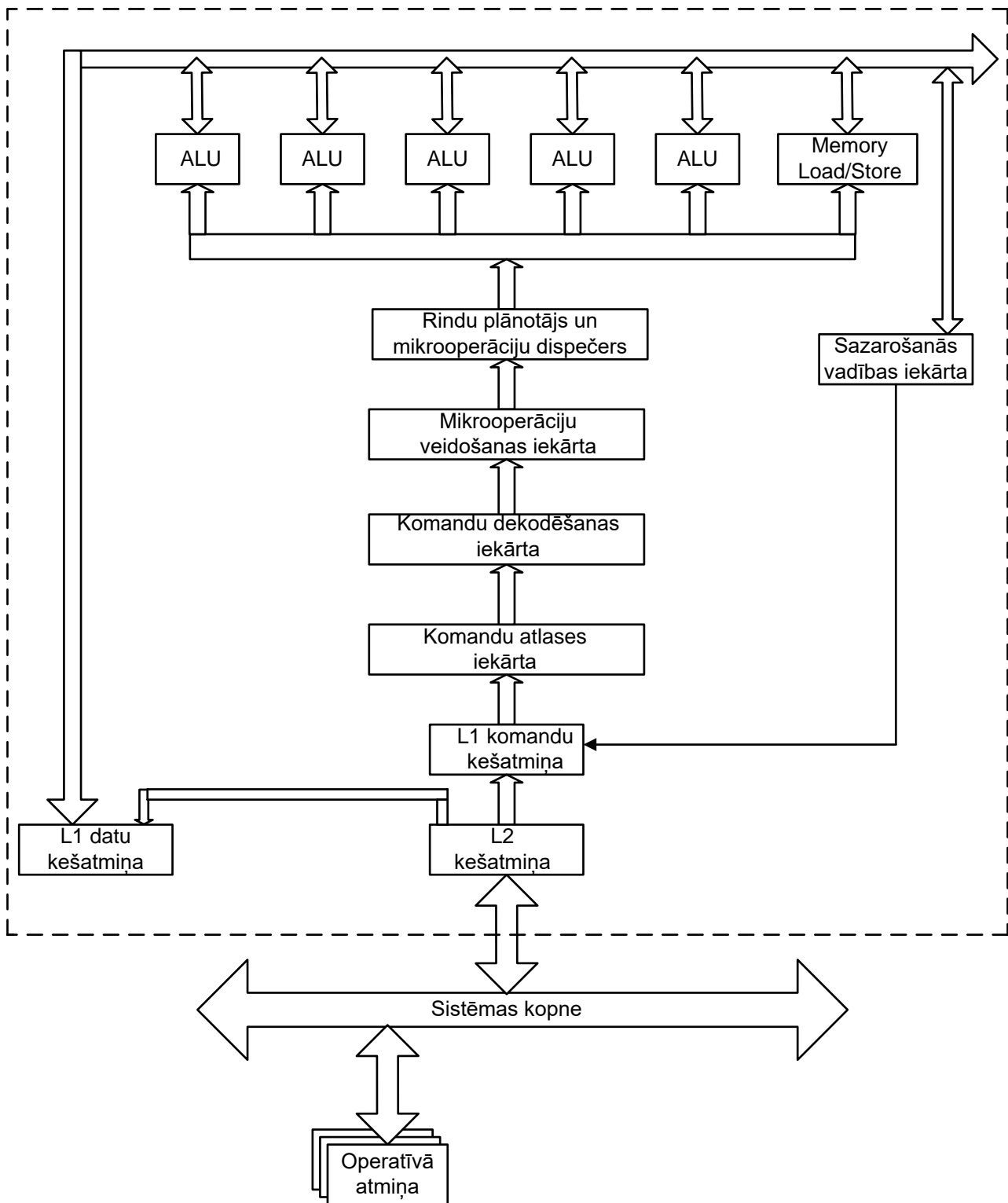
4.3. att. Atmiņas kešdarbe procesoros

Attēlā attēlotais izpildāmais modulis pēc būtības ir nosacīts nosaukums veselai grupai savstarpēji saistītu ierīču, kuras nodrošina komandu izpildes pilnu ciklu. Šajā grupā ietilpst instrukciju izlases un dekodēšanas ierīces, vairākas aritmētiski loģiskās ierīces (tai skaitā moduļi skaitļu ar peldošo punktu apstrādei), mikroinstrukciju rindas plānotājs, mikroinstrukciju plānotājs u.c.

4.4. attēlā tiek piedāvāta komandu, kas tiek nodotas procesoram, izpildes secība uz hipotētiska procesora bāzes (tajā ietilpst visi mūsdienu procesoram piemītošie komponenti).

Pēc komandu/datu atlasē atmiņā tie tiek iesūtīti vispārējās nozīmes L2 kešatmiņā. Pēc tam dati L2 kešatmiņā tiek sadalīti: procesora instrukcijas nonāk L1 komandu kešatmiņā, bet dati – L1 datu kešatmiņā. Tālāk sākas komandu izpildes process, kas visiem mūsdienu procesoriem notiek pēc tipiskas shēmas:

- no sākuma notiek pieeja L1 komandu kešatmiņai nākamās komandas/instrukcijas izvēlei jeb atlasei;
- ja instrukcija atrodas kešatmiņā, tad izpildās parastā darbību secība - atlase, instrukcijas dekodēšana un tās izpilde. Ja instrukcijas kešatmiņā nav, notiek pieeja L2 kešatmiņai;
- ja instrukcija atrasta, sākas kārtējās komandu grupas ielāde L1 komandu kešatmiņā, pēc kā dotā komanda tiek izņemta no L1 kešatmiņas, un sākas tās izpildes cikls. Ja, veicot pieeju L2, instrukcijas tur nav, tad procesors vēršas pie kopējās atmiņas jaunas lappuses ielādei.




4.4. att. Procesora komandu izpildes secība

- no L1 kešatmiņas paņemtā komanda tiek dekodēta, kā rezultātā veidojas mikroinstrukciju (μops) virkne. Mikrooperācijas ir komandas, kas veic relatīvi vienkāršas darbības. IA-32 platformas procesori operē ar CISC komandu kopu, bet pašā procesorā CISC komandas tiek pārveidotas RISC instrukcijās. Faktiski mikroinstrukcijas arī ir tādas RISC instrukcijas, kuras tiek izņemtas no ROM (tas satur visu procesora instrukciju mikrokodus);
- mikroinstrukciju virkne nonāk vienā no mikroinstrukciju rindām, ko sastāda mikrooperāciju plānotājs saskaņā ar mikrooperācijas tipu. Hipotētiskajam

procesoram 4.4. attēlā tiks izveidotas rindas ALU un matemātiskajam līdzprocesoram - FPU (*Floating Point Unit*). Jāatzīmē, ka līdz brīdim, kad operācijas izpilda plānotājs, operāciju secība atbilst izpildāmās programmas instrukciju secībai. Pēc mikrooperāciju plānotāja mikroinstrukciju izpildes secība var mainīties tā iemesla dēļ, ka daļa instrukciju var izpildīties paralēli, jo procesoros ir vairāki ALU un FPU moduļi.

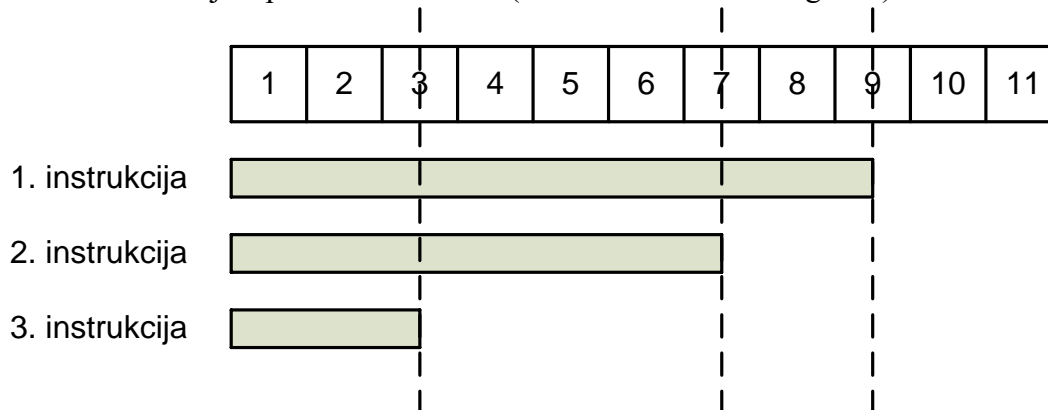
Mikrooperāciju izpildes laikā var tikt iestatīti dažādi stāvokļa karodziņi (*flags*), kurus izmantojot, programma var sazaroties. Katram procesoram ir savs pāreju un sazarošanās apstrādes algoritms – šādu situāciju gadījumā mainās kešatmiņas saturs, un operāciju secības izpilde sākas no jaunas adreses.

4.2. Komandu konveijerapstrāde

 Visi procesori nodrošina iespēju izpildīt komandas ar tā saucamā komandu konveijera (*pipeline*) palīdzību [4,9]. Tā būtība ir tāda, ka vienas procesora instrukcijas izpilde tiek sadalīta vairākos soļos jeb stadijās (*stage*). Šādu soļu skaitu un tajos izpildāmās operācijas nosaka procesora arhitektūra. Piemēram, *Intel Pentium 4* procesorā komandu konveijers satur 20 soļus. Katrā stadijā izpildās kāda no operācijām komandas apstrādē (izlase, dekodēšana, mikrooperācijas veidošana utt.), turklāt atsevišķas operācijas izpildei var būt nepieciešami vairāki soļi. Katrs konveijera solis paredz instrukcijas apstrādi ar noteiktas ierīces palīdzību (ar komandu izlases ierīci, komandu dekoderu, komandu dispečeru utt.).

Komandu konveijera izmantošana ļauj samazināt aiztures laiku starp instrukciju izpildi. Ja procesoru agrīnajos modeļos nākamā instrukcija sāka izpildīties tikai pēc iepriekšējās beigām, tad tagad vairākas instrukcijas var izpildīties vienā un tajā pašā laikā.


Komandu konveijers neļauj organizēt instrukciju paralēlu izpildi, jo dažādas komandas atrodas dažādās izpildes stadijās, taču ar tā palīdzību būtiski samazinās procesora dīkstāves laiks. Hipotētisks komandu konveijers parādīts 4.5. attēlā (reāli tas ir daudz sarežģītāks).

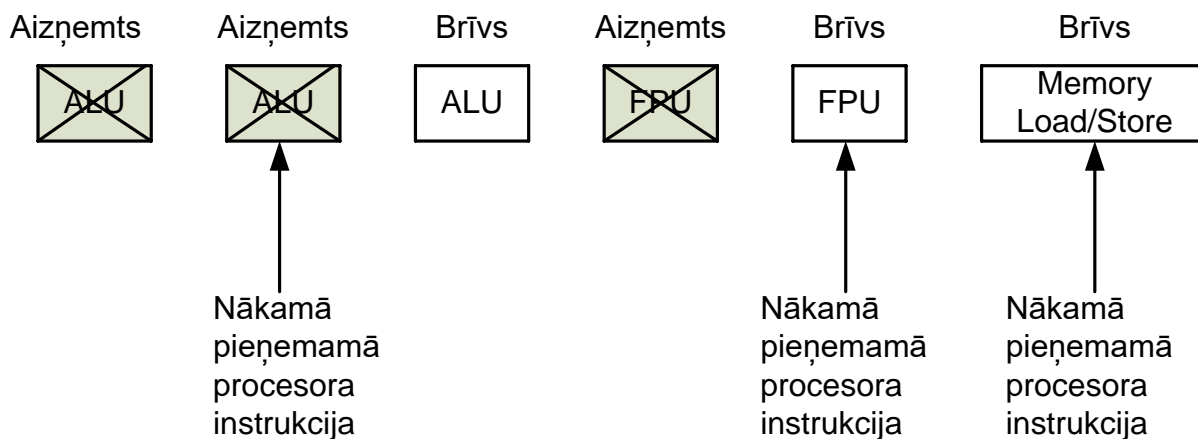


4.5. att. Komandu konveijerapstrādes piemērs

Šeit pirmā instrukcija atrodas devītajā izpildes solī, otrā instrukcija – 7 izpildes solī un trešā instrukcija – 3 solī.

4.3. Komandu paralēlā izpilde

 Visi mūsdienu procesori satur vairākas ALU un FPU un var saturēt citus moduļus mikroinstrukciju paralēlai apstrādei. Turklāt rodas iespāids, ka paralēli strādā vairāki procesori – tādu procesora arhitektūru sauc par superskalāru. Ar piemēra palīdzību var parādīt, kādā veidā izpildās instrukciju virkne 4.6. attēlā dotajā hipotētiskajā procesorā.



4.6. att. Komandu paralēlā izpilde

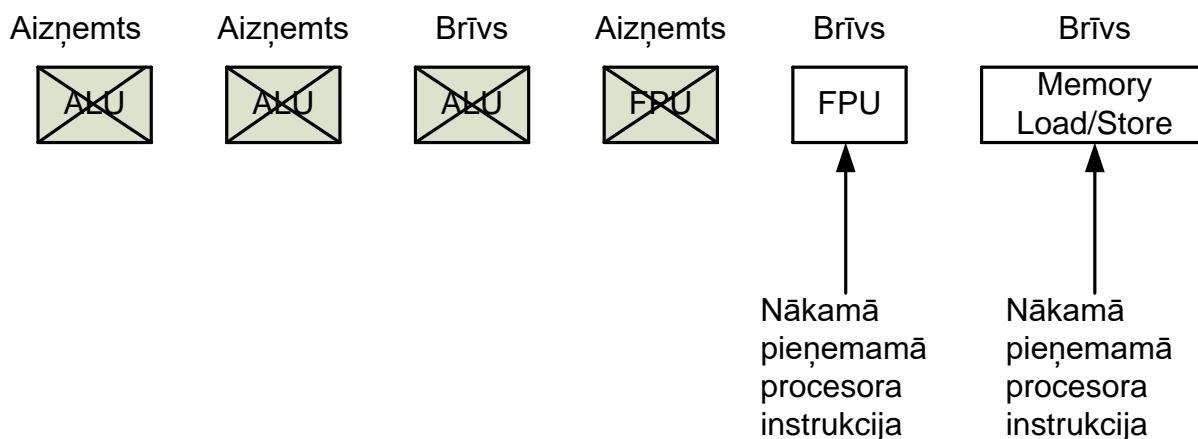
4.1. piemērs

Pieņem, ka ir dota šāda procesora instrukciju virkne (ērtības labad instrukcijas ir sanumurētas):

1. `add EAX, EDX`
2. `inc EBX`
3. `sub EAX, ECX`
4. `mov mem1, ESI`
5. `fmul`
6. `shr ECX, 2`

Pirms pirmās instrukcijas izpildes ir brīvi daži izpildāmie moduļi (sk. 4.6. att.). Starp tiem ir ALU, FPU un datu ielādes/izlādes modulis. Pirmajai komandai (`add EAX, EDX`) jāizpildās ALU modulī, tāpēc pēc dekodēšanas rindas plānotājs novieto mikroinstrukciju rindu atbilstošajā instrukcijas tipa rindā.

Tā kā dotajā momentā ir brīva ALU ierīce, mikroinstrukciju rindas dispečers nosūta komandas mikroinstrukcijas uz šo ALU izpildei. Pirms otrās komandas izpildes (`inc EBX`) ir 4.7.attēlā parādītais stāvoklis.



4.7. att. Izpildāmo moduļu stāvoklis pēc pirmās komandas izpildes

Instrukcija *inc EBX* izpildei jāievieto kādā no ALU ierīcēm. Dotajā momentā tās aizņemtas ar iepriekšējo komandu izpildi, tāpēc procesors pāriet pie 3. instrukcijas (*sub EAX, ECX*). To arī nav iespējams izpildīt, jo ALU joprojām ir aizņemti. Tad procesors vēršas pie 4. instrukcijas (*mov mem1, ESI*), kura var tikt izpildīta datu apmaiņas ar atmiņu modulī. Pārejot pie 5. instrukcijas, procesors konstatē, ka FPU modulis ir brīvs, tāpēc sākas dotās instrukcijas izpilde.

Tādam procesora komandu izpildes mehānismam ir vispārpieņemts apzīmējums OOO (*Out-of-Order engine*). Visi bez izņēmuma mūsdienu procesori izmanto OOO mehānismu, jo ar tā palīdzību tiek sasniegta komandu izpildes veikspēja un tiek izslēgtas nelietderīgas aiztures komandu virknes izpildē.

Līdz šim tika apskatīta procesora instrukciju izpildes kārtība, pieņemot, ka tās visas izpildās secīgi. Acīmredzams, ka pat visvienkāršākās programmas reti iztiek bez sazaršanās un pārejas operācijām uz citiem izpildāmā koda punktiem. Mūsdienu procesoros tādas situācijas tiek atrisinātas ar vēl viena mehānisma palīdzību – ar tā saucamo „spekulatīvo” komandu izpildes algoritmu (*Speculative Execution*). Tā darbība ir demonstrēta ar piemēra palīdzību.



4.2. piemērs

Pieņem, ka dota šāda komandu virkne:

```
add EAX, 0x10
cmp EAX, EDX
jb next
xor EAX, EAX
adc EAX, 0
next:
fdiv
sub EDX, mem1
```

Pieņem arī, ka procesoram ir brīvi izpildāmie moduļi (sk. 4.6. att.). Analizējot doto fragmentu un atceroties, ka tas izmanto OOO mehānismu, procesors nosūta instrukciju *fdiv* uz brīvo FPU izpildei. Tajā pašā laikā brīvajā ALU sākas instrukcijas *cmp EAX, EDX* izpilde.

Tādā veidā vienlaikus tiek apstrādi divi programmas sazarojumi. Ja izrādās, ka reģistra EAX saturs lielāks par EDX, tad procesors ignorē komandas *fdiv* izpildes rezultātu, turklāt nekāda aizture nenotiek, jo viens no FPU moduļiem bija brīvs un instrukcijas *fdiv* izpilde papildus laiku neprasītu. Tajā pašā laikā, ja reģistra EAX saturs būtu mazāks par EDX, tad nenāktos izpildīt instrukciju *fdiv*, jo tā jau izpildīta, bet varētu uzreiz pāriet pie instrukcijas *sub EDX, mem1*.

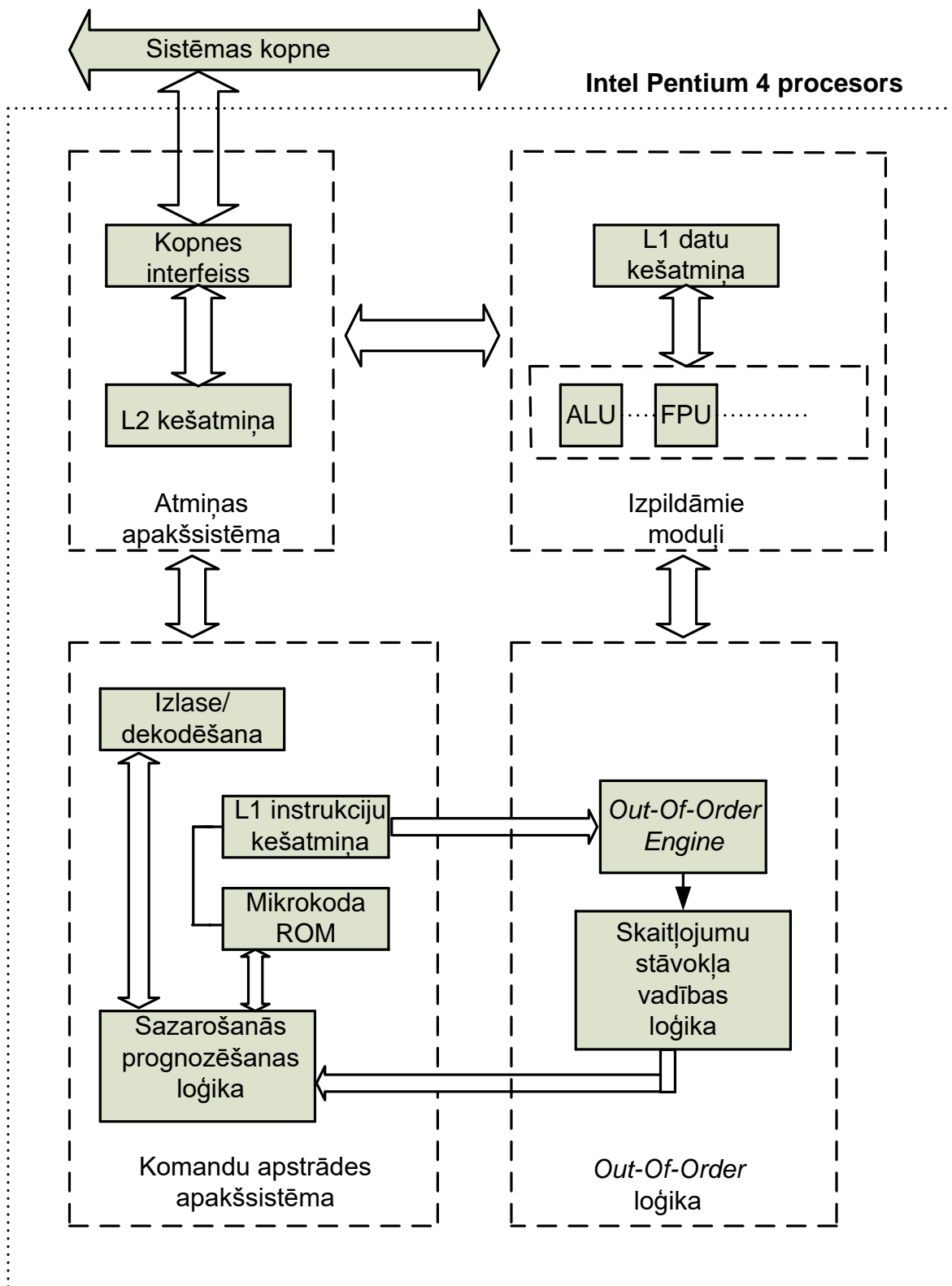
Šajā gadījumā dotais koda atzars izpildās ātrāk.

Apskatītie veikspējas optimizācijas mehānismi ir vispārīgi, tajā pašā laikā katram procesoram ir papildus mehānismi programmas koda izpildes ātruma palielināšanai.

4.4. Intel Pentium 4 arhitektūras īpatnības

Intel Pentium 4 procesoru arhitektūra balstās uz iepriekšējās apakšnodaļās aprakstītajiem principiem, un tā funkcionālā shēma parādīta 4.8. attēlā [9,19].

Skaidrākai procesoru darbības principu izpratnei šajā shēmā nav uzrādīti atsevišķi funkcionālie mezgli, kas nav pārāk būtiski un to funkcionēšanas principi tiks izklāstīti tālāk tekstā.



4.8. att. Intel Pentium 4 procesora funkcionālā shēma



Salīdzinot ar pārējiem, Intel Pentium 4 procesoriem ir šādas īpatnības:

- procesors pārraida datus pa kopni ar četrkārtīgu ātrdarbību (QDR). 4.1. tabulā parādīta kopnes takts frekvence un datu pārraides ātrums;

Datu pārraides ātruma atkarība no kopnes frekvences

Kopnes takts frekvence (MHz)	Veiktspēja (MHz)	Datu pārraides ātrums (Gb/s)
100	400	3,2
133	533	4,2
200	800	6,4
266	1066	8,5

- kopne datu apmaiņai starp L2 un L1 datu kešatmiņu ļauj vienlaicīgi izpildīt 256 bitu apmaiņu. Salīdzinot ar iepriekšējiem procesoru modeļiem (64 biti), tas ir 4 reizes vairāk. Tajā pašā laikā caurlaide starp L2 un L1 instrukciju kešatmiņu (kas tagad saucas par *Trace Cache*), ir palikusi iepriekšējā – 64 biti;
- L1 instrukciju kešatmiņa fiziski ir izvietota blakus komandu dekodēšanas iekārtai (agrāk - blakus instrukciju izlases iekārtai). *Trace Cache* var saturēt līdz 12000 mikroinstrukciju (katra mikroinstrukcija aizņem 100 bitus), t.i., tās apjoms ir 150 Kb;
- *Intel Pentium 4* procesoram ir 128 iekšējie reģistri, kas izvietoti modulī RAT (*Register Renaming Unit*). Šo reģistru nozīme tiks apskatīta vēlāk;
- procesorā atrodas 5 izpildāmie moduļi (ALU un FPU), kuri strādā paralēli, un divi moduļi datu ielādei/izlādei caur operatīvo atmiņu.

Kā tika minēts iepriekš, *Intel Pentium 4* procesoram faktiski ir 3 kešatmiņas. Vispārējās nozīmes L2 kešatmiņas apjoms var būt 256 Kb, 512 Kb, 1 Mb vai 2 Mb (modeļiem, kas izgatavoti pēc 90 nanometru tehnoloģijas). Būtiski arī tas, ka L1 komandu kešatmiņa tagad izvietota blakus dekodēšanas iekārtai, kas palielina procesora veiktspēju, jo atkārtotas jau izmantotu komandu ielādes vietā var izmantot instrukciju kešatmiņā atrodošos mikroinstrukciju bloku.

Tādas novitātes pamatā ir šāds pieņēmums: programmas ciklu izpildei, kam var būt nepieciešamas vairākas atkārtotās, jau dekodētās procesora instrukcijas tagad atrodas *Trace Cache* un var tikt izņemtas pēc nepieciešamības, tādējādi nav nepieciešams tās atkārtoti dekodēt.

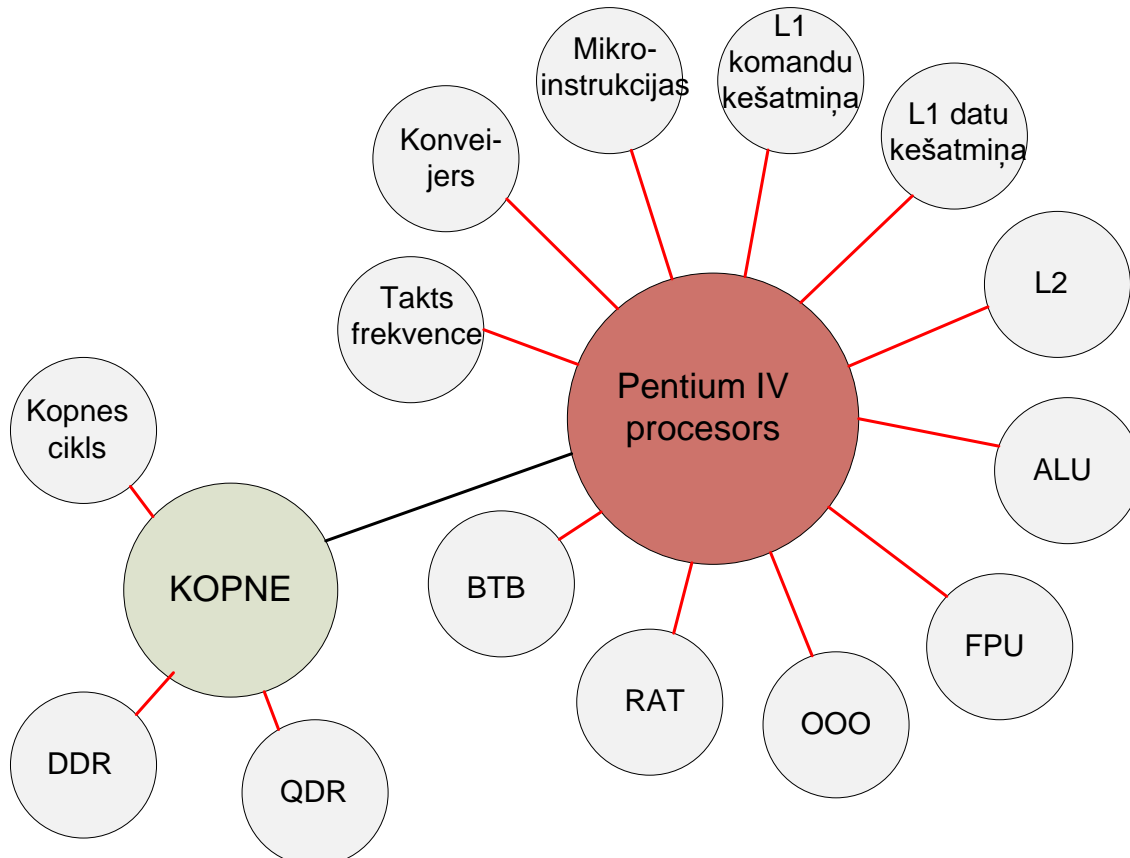
Turpmāk termina „L1 instrukciju kešatmiņa” vietā tiks izmantots termins „*Trace Cache*”. *Trace Cache* ir savs atmiņas buferis, kurā glabājas informācija par programmas sazaršanās punktiem. Buferis saucas BTB (*Branch Target Buffer*) un var saturēt līdz 512 sazaršanās punktiem. Tāds pats buferis ir arī komandu izlases iekārtai, bet tā BTB apjoms ir lielāks – tas var saturēt līdz 4096 sazaršanās punktiem.

NODAĻAS KOPSAVILKUMS

- Visu datorsistēmas iekārtu darbs tiek sinhronizēts ar takts frekvences impulsiem, ko rada procesora un mātesplates elektroniskās shēmas.
- Visas laika raksturlielumu atkarības datorsistēmās tiek mērītas kopnes ciklos.
- Procesora iekšienē visas operācijas tiek izpildītas ar procesora takts frekvenci.
- Pēc komandu/datu izlases no atmiņas tie tiek iesūtīti vispārējās nozīmes L2 kešatmiņā. Pēc tam dati L2 kešatmiņā tiek sadalīti: procesora instrukcijas nonāk L1 komandu kešatmiņā, bet dati – L1 datu kešatmiņā. Tad sākas komandu izpildes process.
- Visi procesori nodrošina iespēju izpildīt komandas ar komandu konveijera palīdzību – vienas procesora instrukcijas izpilde tiek sadalīta vairākos soļos.
- *Intel Pentium 4* procesorā komandu konveijers satur 20 soļus.
- Visi mūsdienu procesori satur vairākas ALU un FPU un var saturēt citus moduļus mikroinstrukciju paralēlai apstrādei.
- *Intel Pentium 4* procesoriem ir šādas būtiskas īpatnības:
 - procesors pārraida datus pa kopni ar četrkārtīgu ātrdarbību;

- kopne datu apmaiņai starp L2 un L1 ļauj vienlaicīgi izpildīt 256 bitu apmaiņu;
- L1 instrukciju kešatmiņa fiziski ir izvietota blakus komandu dekodēšanas iekārtai;
- procesoram ir 128 iekšējie reģistri;
- procesorā atrodas 5 izpildāmie moduļi (ALU un FPU) un 2i moduļi datu ielādei/izlādei caur operatīvo atmiņu.

4.9.attēlā parādīts ceturtajā nodaļā minēto svarīgāko jēdzienu koks.



4.9. att. Ceturtās nodaļas svarīgāko jēdzienu koks

Uzdevumi un jautājumi patstāvīgam darbam

1. Sarindot kešatmiņas tipus pēc to ātrdarbības.
2. Ja matemātiskais līdzprocesors (FPU) spēj apstrādāt arī veselos skaitļus, kāpēc ir nepieciešams ALU, kas veic darbības tikai ar veselajiem skaitļiem?
3. Kas notiktu hipotētiskajā gadījumā, ja procesoram, atmiņai un I/O ierīcēm būtu vienāda ātrdarbība?
4. Kāpēc procesorā nepieciešami vairāki ALU un FPU moduļi?
5. Vai būtu lietderīgi mikroinstrukciju bloku izvietot citā ierīcē, nevis ROM?
6. Vai komandu konveijers realizē instrukciju paralēlu izpildi?
7. Uz piemēra pamata modelēt komandu paralēlu izpildi.
8. Ja L1 instrukciju kešatmiņa satur 5000 mikroinstrukcijas, kāds ir tās apjoms kilobaitos?

5. PROCESORA PROGRAMMU MODELIS

Nodaļā ir analizēts x86 procesora programmu arhitektūras, komandu sistēmas un datu adresācijas modelis, jo bez šīm zināšanām nav iespējams izprast mūsdienu datoru funkcionēšanas principus. Tas attiecas gan uz programmētājiem, kas strādā ar augsta līmeņa programmēšanas valodām, gan uz programmu izstrādātājiem zema līmeņa asamblera valodā.

Nodaļā tiek izmantots programmas kods asamblera valodā, jo programmas funkcionēšanas analīze ļauj izprast procesora komandu darbības īpatnības.

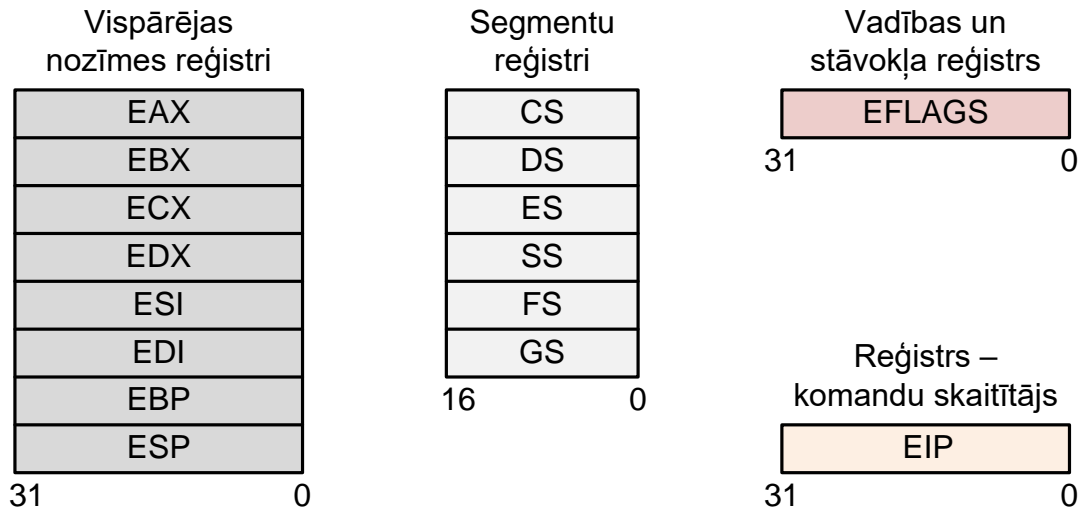
5.1. Procesora reģistri



x86 procesora bāzes programmas modelis – tie ir procesora resursi. Bāzes programmas modelis satur:

- 8 vispārējas nozīmes reģistrus (*general purpose registers*) datu un norāžu uzglabāšanai;
- segmentu reģistrus (*segment register*) 6 segmentu selektoru uzglabāšanai;
- vadības un stāvokļa reģistru EFLAGS (*flag register*), ar kura palīdzību notiek programmu izpildes stāvokļa un procesora stāvokļa vadība;
- reģistru-norādi EIP, kas satur nākamo izpildāmo procesora instrukciju;
- procesora komandu jeb instrukciju sistēmu;
- datu adresācijas veidus (režīmus), ko izmanto procesora komandas.

Bāzes reģistri ļauj veikt datu apstrādes pamatdarbības un tiek izmantoti programmu izstrādē (sk. 5.1. att.) [4,6,7,8,9].



5.1. att. Procesora bāzes reģistri

Bāzes reģistru komplektā funkcionāli tiek izdalītas atsevišķu reģistru grupas, kas pilda noteiktu uzdevumu. Piemēram, vispārējās nozīmes reģistru grupā iekļauti 8 reģistri, kurus var izmantot šādiem nolūkiem:

- kā aritmētisko un loģisko operāciju operandus;
- kā operandus adrešu izskaitļošanā;
- kā norādes uz atmiņas mainīgajiem.

Jebkuru no šiem reģistriem var izmantot iepriekš minētajās operācijās, taču atsevišķiem reģistriem ir sava specifika, piemēram, reģistrs ESP tiek izmantots steka rādītāja uzglabāšanai, tāpēc izmantot to citās operācijās nav vēlams.

Noteiktu operandu vērtību uzglabāšanai daudzās operācijās bieži tiek izmantoti reģistri ECX, ESI un EDI. Arī citi reģistri tiek izmantoti noteiktos nolūkos:

- EAX – kalpo par akumulatoru darbā ar operandiem un uzglabā operāciju rezultātus;
- EBX – izmanto kā norādi uz datiem, kas atrodas datu segmentā, kuru adresē reģistrs DS;
- ECX – izmanto kā ciklu skaitītāju;
- EDX – kalpo par norādi uz ievadizvades ierīču pieslēgvietām (komandas *in* un *out*);
- ESI – satur datu adresi, kas atrodas segmentā, kuru adresē reģistrs DS;
- EDI – adresē datus, kas atrodas segmentā, kura bāzes adrese uzdota reģistrā ES;
- ESP – satur steka norādi steka segmentā, kuru adresē reģistrs SS;
- EBP – satur norādi uz datiem, kas atrodas stekā, kuru adresē reģistrs SS.

Vispārējas nozīmes reģistru jaunākie 16 biti tiek adresēti tāpat kā procesora 80x86 16 bitu reģistri (AX, BX, CX, DX, BP, SI, DI, SP). 16 bitu reģistri AX, BX, CX, DX ļauj adresēt vecākos (AH, BH, CH, DH) un jaunākos (AL, BL, CL, DL) 8 bitu reģistrus (sk. 5.2. att.).

	31	16 15	8 7	0
EAX		AH	AL	
EBX		BH	BL	
ECX		CH	CL	
EDX		DH	DL	
ESI		SI		
EDI		DI		
EBP		BP		
ESP		SP		

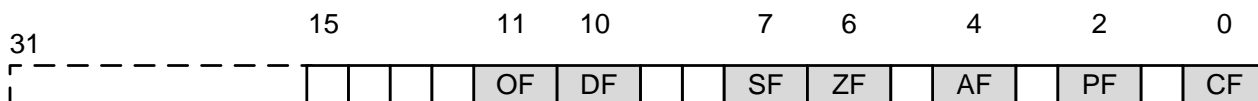
5.2. att. Vispārējas nozīmes reģistru izmantošana

Segmentu reģistri (CS, DS, SS, ES, FS, GS) satur 16 bitu segmentu selektorus.



Selektors ir speciāla norāde vai rādītājs, kas identificē doto segmentu atmiņā.

Reģistrs EFLAGS satur stāvokļa bitu grupu (bieži sauc par karodziņu vai karogu), vadības karodziņu un sistēmas karodziņu grupu. 5.3. attēlā parādīta karodziņu reģistra struktūra un svarīgāko karodziņu apzīmējumi.



5.3.att. Procesora vadības/stāvokļa reģistrs

Attēlā parādīti biežāk izmantojamie karodziņi:

- OF (*Overflow Flag*) – pārpildīšanās karodziņš. Fiksē pārpildīšanās situāciju, t.i., ja aritmētiskās operācijas rezultāts iziet ārpus pieļaujamā diapazona robežām;
- DF (*Direction Flag*) – virziena karodziņš. Izmanto rindu apstrādes komandas. Ja DF=0, tad rinda tiek apstrādāta tiešā virzienā – no mazākām adresēm uz lielākām. Ja DF=1, rindu apstrāde notiek pretējā virzienā;

- SF (*Sign Flag*) – zīmes karodziņš. Rāda operācijas rezultāta zīmi. Ja SF=1, tad ir negatīvs rezultāts;
- ZF (*Zero Flag*) – nulles karodziņš. ZF=1, ja operācijas rezultāts ir 0;
- AF (*Auxiliary Flag*) – papildus pārneses karodziņš. Izmanto operācijās ar sapakotiem bināri decimāliem skaitļiem;
- PF (*Parity Flag*) – paritātes karodziņš. PF=1, ja operācijas rezultātā bināro vieninieku skaits ir pārskaitlis;
- CF (*Carry Flag*) – pārneses karodziņš. Parāda, vai operācijās ir bijusi bitu pārņemšana.


Stāvokļa karodziņus izmanto nosacītās pārejas komandas *jcc*, kur *cc*- nosacījuma kods: *eq*, *le*, *lt*, *ne* utt., kā arī komandas *setcc*, *loopcc* un *cmovcc*.

Procesora stāvokļa karodziņus var ievietot stekā un izņemt no tā ar komandu *pushf*, *pushfd*, *popf* un *popfd* palīdzību. Tos var ielādēt reģistra AX vecākajā daļā un izņemt no tā ar komandu *lahf* un *sahf* palīdzību.

Reģistram EIP ir speciāla nozīme - tas satur nākamās izpildāmās komandas nobīdi. Ja programmā sastopamas komandas *jcc*, *call*, *ret* vai *iret*, tad EIP saturs var mainīties patvaļīgā veidā, turklāt nākamās komandas nobīde var kļūt gan pozitīva, gan negatīva. Tiešā veidā ar kaut kādas instrukcijas palīdzību reģistra saturu izmainīt nevar, vienīgi var nolasīt tā saturu – izpildot komandu *call* un apskatot steka saturu (tas norādīs uz nākamo komandu).


Reģistru EIP var modificēt netieši – ja stekā nomaina nākamās komandas adresi ar vēlamu, pirms tam izpildot komandu *call*.

Speciāla nozīme ir segmenta reģistriem CS, DS, ES, SS, FS un GS. Tie tiek izmantoti visās programmās, norādot uz koda segmenta, steka segmenta un datu segmenta bāzes adresēm.

 *Intel* procesoros terminam „segments” ir divējāda nozīme. Ar to var apzīmēt iepriekš noteikta apjoma fiziskās atmiņas apgabalu (32 bitu lietojumprogrammām tas teorētiski var sasniegt 4 Gb). Tāpat ar šo terminu var apzīmēt mainīga izmēra atmiņas apgabalu, kurā var atrasties programmas kods, dati vai steks.

Fiziskais segments var atrasties tikai adresēs, kas dalās ar 16. Katrs loģiskais programmas segments definē atmiņas apgabalu, kuru adresē segmenta selektors (atrodas segmenta reģistrā).

Nelielām programmām (mazākām par 64 Kb) programmas kods un dati var atrasties atsevišķos segmentos – tāpēc problēmas nerodas. Tajā pašā laikā lielām programmām, kas aizņem vairākus koda vai datu segmentus, nepieciešama pareiza to datu adresācija, kuri atrodas dažādos segmentos. Ja programmas kods atrodas vairākos segmentos, tad tiek sarežģīta pārejas un sazarošanās realizācija programmā, kā arī var rasties problēmas ar procedūru izsaukšanu.

 Izmantojot 32 bitu aizsargāto režīmu (tieši tādā režīmā arī strādā visas mūsdienu *Windows* lietojumprogrammas), tamlīdzīgas adresācijas problēmas nerodas, jo programmas koda un datu adresācijai tiek izmantota 32 bitu efektīvā adrese nepārtrauktajā atmiņas apgabalā.

5.2. Procesora komandu sistēma un datu adresācija

Intel procesoru komandu sistēma sastāv no tā saucamajām vispārējās nozīmes procesora instrukcijām (*general-purpose instructions*). Pēc funkcionālajām pazīmēm tās var iedalīt vairākās grupās:

- datu pārvietošanas komandas;
- veselo skaitļu aritmētiskās komandas;
- loģisko operāciju komandas;
- vadības nodošanas komandas;
- operāciju ar rindām komandas.

Atsevišķas komandas grūti ierindot kādā no uzskaitīto komandu grupām, piemēram, komandas darbam ar steku vai komandas darbam ar tabulu datiem.

Lielākā daļa komandu strādā ar operandiem atmiņā (ko var adresēt ar vienu no nākamajā apakšpunktā aprakstītajiem veidiem), kā arī ar vispārējas nozīmes un segmentu reģistriem. Visu procesora komandu detalizēts apraksts dots *Intel* dokumentācijā [19], kā arī literatūrā par asamblera programmēšanas valodu [2,6,8,9,12].

Jebkuras programmas izpildes gaitā procesors griežas pie atmiņas, kurā glabājas komandas un dati. Lai iegūtu pieeju datiem, kaut kādā veidā ir jānosaka to adrese atmiņā.



Operanda adreses noteikšanas metode tiek saukta par adresācijas režīmu jeb adresāciju.

Adresācijas metožu analīzei tiks izmantota *Intel* procesoru asamblervaloda.



Asamblers ir programma, kas pārvērš asamblervalodā rakstītās instrukcijas mašīnvalodas komandās.

Procesora instrukcijas satur dažādākās komandas, kas strādā gan ar operandiem, gan bez tiem. Operanda avota dati var atrasties reģistrā, atmiņā, pieslēgvietā vai tikt uzdoti tieši instrukcijā. Operands mērķis var atrasties operatīvajā atmiņā, reģistrā vai pieslēgvietā.

Datu adresācijas izpratnei tālāk tiks analizēts operanda adreses veidošanās mehānisms.



Operanda adrese tiek veidota pēc shēmas „segments : nobīde” un 32 bitu lietojumprogrammām adrese ir 16:32 bitu formā. Operanda nobīde tiek saukta par tā efektīvo jeb izpildāmo adresi (*Effective address*).

Segmenta selektors var tikt uzdots tiešā vai netiešā veidā. Parasti segmenta selektors tiek ielādēts segmenta reģistrā, bet pats reģistrs tiek izvēlēts atkarībā no izpildāmās operācijas tipa (sk. 5.1. tabulu).

5.1. tabula

Segmenta reģistra izvēles kritēriji

Operācijas tips	Segmenta reģistrs	Izmantojamais segments	Izmantošanas nosacījumi
Procesora komandas	CS	Programmas segments	Procesora komandu izsaukšanas gadījumā
Pieeja stekam	SS	Steka segments	Visas operācijas ar steku un atmiņu, kurās bāzes reģistri ir ESP un EBP
Lokālie dati	DS	Datu segments	Visas operācijas ar datiem, izņemot tās, kur tiek izmantots steks vai rinda-mērķis (darbā ar rindām)
Darbs ar rindām (rinda-mērķis)	ES	Datu segments, ko adresē ES	Rindu operācijās operands-mērķis

Procesors automātiski izvēlas segmentu atkarībā no tabulā aprakstītā kritērija. Ja, piemēram, ir jā saglabā reģistra EDX saturs atmiņā, ko adresē segmenta reģistrs ES un nobīde (reģistrā ESI), tad var izmantot komandu *mov ES:[ESI], EDX*.

Procesora līmenī uz segmenta nomaiņu norāda ar speciāla nomaiņas prefiksa palīdzību – tas ir viena baita skaitlis pirms komandas koda. Atsevišķos gadījumos prefiksa nomaiņa nav iespējama:

- programmas koda segmentam – visas komandas izmanto tikai segmenta reģistru CS;

- operācijās ar rindām mērķis tiek adresēts tikai ar reģistra ES palīdzību;
- operācijas ar steku vienmēr izmanto reģistru SS.

Atsevišķas procesora instrukcijas pieprasa segmenta reģistru tiešu inicializāciju. Tādos gadījumos segmenta selektors var tikt izņemts no 16 bitu reģistra vai no atmiņas mainīgā kā, piemēram, komandā *mov DS, BX*. Šeit segmenta selektors, kas atrodas reģistrā BX, tiek ievietots reģistrā DS. Atsevišķos gadījumos segmenta selektoru var uzdot ar 48 bitu rādītāja palīdzību (jaunākais dubultvārds satur 32 bitu nobīdi, bet vecākais vārds – 16 bitu segmenta selektoru).

Pārsvarā programmētāji manipulē ar operanda efektīvo adresi, t.i., ar to operanda pilnās adreses daļu, kura nosaka operanda nobīdi dotajā segmentā.

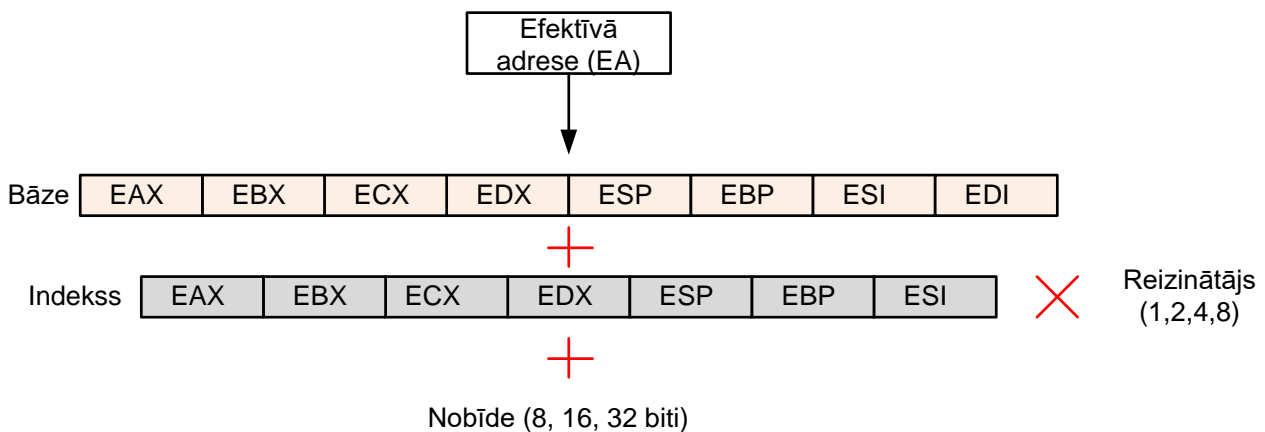


Lai izvairītos no jucekļa, turpmāk termina „nobīde” vietā tiks lietots korektāks formulējums ”efektīvā adrese”. Ar nobīdi tiks saprasta skaitliskā vērtība, kura tiek pieskaitīta adresei.

Operanda efektīvo adresi var uzdot vairākos veidos. Procesora instrukcijas piedāvā programmētājam plašu adresācijas veidu spektru. Vispārīgā gadījumā operanda efektīvā adrese var sastāvēt no vairākām sastāvdaļām:

- nobīdes (tā var būt 8, 16 vai 32 bitu vērtība);
- bāzes (vispārējas nozīmes reģistra saturs);
- indeksa (vispārējas nozīmes reģistra saturs);
- reizinātāja (2, 4 vai 8).

Vispārīgā gadījumā efektīvā adrese EA sastāv no nobīdes, bāzes un indeksa summas, turklāt šo summu var koriģēt ar reizinātāja palīdzību (sk. 5.4. att.):



5.4.att. Efektīvās adreses izskaitļošanas shēma



Formula EA izskaitļošanai ir šāda: $EA = \text{Bāze} + \text{Indekss} \times \text{Reizinātājs} + \text{Nobīde}$.

Pastāv atsevišķi ierobežojumi reģistru izmantošanā par bāzes vai indeksu reģistriem efektīvās adreses veidošanā:

- reģistru ESP nedrīkst izmantot kā indeksu reģistru;
- ja par bāzes reģistriem izmanto reģistrus ESP un EBP, tad par segmentu reģistru jāizmanto SS. Pārējos gadījumos pēc noklusējuma par segmentu reģistru izmanto DS.

Jāatzīmē, ka bāze, indekss un nobīde var tikt izmantoti dažādās kombinācijās, turklāt jebkurš komponents var izpalikt. Reizinātājs tiek izmantots tikai ar indeksu.



5.1. piemērs. Tiešā adresācija

```
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
    int i1=0;
    __asm
```

```

{
    mov EAX, i1
    sub EAX, 5
    mov i1, EAX
}
printf("i1= %d\n", i1);
getchar();
return 0;
}

```

Rezultāts

i1=-5

Komentāri

Šeit instrukcijas *mov EAX, i1*

mov i1, EAX operē ar mainīgo *i1*, kas atrodas atmiņā. Tiek izskaitļota mainīgā efektīvā adrese tieši instrukcijā *mov* un rezultāts tiek izvadīts displeja ekrānā.

Tiešās adresācijas režīmu pielieto darbā ar vienkāršiem mainīgajiem, turklāt operandam avotam un operandam mērķim jābūt ar vienādiem izmēriem (savādāk kompilācijas procesā tiks izvadīts kļūdas paziņojums). Piemērā mainīgais *i1* ir vesela tipa mainīgais, tāpēc kļūda nerodas.



5.2. piemērs. Sarežģītāks tiešās adresācijas paraugs

```

#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
    union my_union
    {
        char c1;
        int i1;
    } test_un;

    test_un.i1 = 0x4442;
    __asm
    {
        mov AL, byte ptr test_un.c1
        add AL, 5
        mov byte ptr test_un.c1, AL

        mov AL, byte ptr test_un.c1+1
        sub AL, 1
        mov byte ptr test_un.c1+1, AL
    }
    printf("c1=%c, c2=%c\n", test_un.c1, test_un.c1+1);
    printf("i1=%x\n", test_un.i1);
    getchar();
    return 0;
}

```

Rezultāts

c1=G c2=H i1=4347

Komentāri:

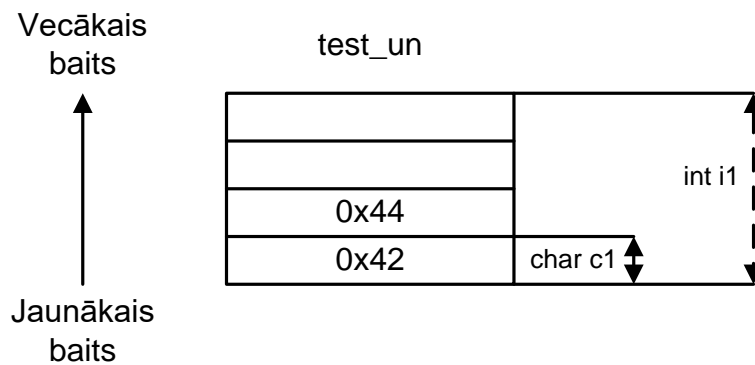
Programmā uzdota tipa *my_union* struktūra *test_un*. Mainīgajam *i1* tiek piešķirta vērtība 0x4442. Tā kā mainīgais *i1* aizņem 4 baitus atmiņā, tad jaunākais baits satur vērtību 0x42, nākamais – 0x42, bet abi vecākie baiti – vērtību 0 (sk. 5.5. att.).

Lai piekļūtu struktūras pirmajam baitam, kura vērtība ir 0x42, un tās saturam pieskaitītu skaitli 5, tiek izmantotas procesora instrukcijas:

```

mov AL, byte ptr test_un.c1
add AL, 5
mov byte ptr test_un.c1, AL

```



5.5.att. Struktūras *test_un* saturs pēc piešķiršanas

Lai piekļūtu struktūras otrajam baitam un no tā vērtības atņemtu skaitli 1, tiek izmantotas instrukcijas:

```
mov AL, byte ptr test_un.c1+1 //0x44
sub AL, 1
mov byte ptr test_un.c1+1, AL
```



5.3. piemērs. Bāzes adresācijas izmantošana darbā ar masīva elementiem

```
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
    int a1[4]= {33, 77, 99, 11};
    int a1size= sizeof(a1);
    __asm
    {
        mov ECX, a1size
        shr ECX, 2
        lea EBX, dword ptr a1
    next:
        mov EAX, 13
        mul [EBX]
        mov [EBX], EAX
        add EBX, 4
        dec ECX
        jnz next
    }
    for (int i1=0; i1<4; i1++)
        printf("a1[%d]*13= %d\n", i1, a1[i1]);

    getch();
    return 0;
}
```

Rezultāts

a1[0]*13=429 a1[1]*13=1001 a1[2]*13=1287 a1[3]*13=143

Komentāri

Efektīvās adreses veidošanai izmanto tikai bāzes reģistra saturu. Tādu adresāciju sauc par bāzes adresāciju un izmanto dinamisku struktūru datu adresācijai (piemēram, masīviem un rindām). Dažkārt to sauc par netiešo adresāciju.

Dots masīvs *a1* ar 4 elementiem. Katrs masīva elements jāpareizina ar skaitli 13. Par bāzes reģistru tiek izmantots EBX. Cikla izpildes nolūkā reģistrā ECX tiek iesūtīts masīva izmērs baitos, tad izmēru pārveido 4-baitu vienībās (komanda *shr ECX, 2*) un reģistrā EBX tiek ielādēta masīva *a1* adrese (komanda *lea EBX, dword ptr a1*).

Pēc šīs komandas izpildes pieeja masīva elementiem tiks veikta attiecībā pret bāzes adresi. Reizināšanas rezultātā jaunākā daļa nomaina attiecīgo masīva elementa saturu. Lai pārietu pie nākamā masīva elementa, reģistra EBX saturam pieskaita vērtību 4.



5.4. piemērs. Pieeja struktūras elementiem ar bāzes un nobīdes palīdzību

```
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
    struct my_struct
    {
        char* s1;
        int i1[5];
    } st1;
    __asm
    {
        lea ESI, dword ptr st1
        mov dword ptr [ESI+8], -3
        mov EAX, dword ptr [ESI+8]
        mov EBX, -17
        mul EBX
        mov dword ptr [ESI+8], EAX
    }
    printf ("%d\n", st1.i1[1]);
    getchar();
    return 0;
}
```

Rezultāts

51

Komentāri

Operanda efektīvās adreses veidošanā izmanto bāzes reģistra saturu plus nobīdi („bāze+nobīde”). Tādu adresācijas veidu lietderīgi izmantot šādos gadījumos:

- pieejai pie nepāra skaita masīva elementiem. Šajā gadījumā bāzes reģistrs satur masīva adresi, bet nobīde dod iespēju piekļūt patvaļīgam masīva elementam;
- pieejai pie ierakstu vai struktūru laukiem. Šajā gadījumā bāzes reģistrs satur ieraksta vai struktūras sākuma adresi, bet nobīde nosaka elementu, pie kura ir jāpiekļūst.



Bāzes adresācijas galvenais pielietojums – piekļuve pie rindu un masīvu elementiem, kad zināma datu sākuma adrese, bet nobīde tiek izskaitļota programmas izpildes laikā.

Asamblērā var izmantot vienu no divām pieraksta formām:

[bāze + nobīde] vai [bāze][nobīde],

kur bāze – reģistrs, kas satur adrese bāzes vērtību, bet nobīde norāda datu elementa pozīciju.

Piemērā uzdota tipa *my_struct* struktūra *st1*, kuras laukos ir norāde *s1* un masīvs *i1*. Pieņem, ka nepieciešams ierakstīt noteiktu vērtību otrajā masīva elementā, t.i., *i1[1]*. Par bāzes reģistru tiek izmantots ESI. Nobīde ir 8 (4 baitus aizņem rādītājs *s1* un 4 – elements *i1[0]*).

No sākuma šajā elementā tiek iesūtīta vērtība -3:

mov dword ptr [ESI+8], -3, tad izpilda instrukciju elementa *i1[1]* satura reizināšanai ar skaitli -17. Beigās iegūtais rezultāts tiek pārrakstīts atpakaļ masīvā : *mov dword ptr [ESI+8], EAX*.



5.5. piemērs. Adresācija „indekss + nobīde”

```
#include "stdafx.h"

int _tmain(int argc, _TCHAR* argv[])
{
    int a1[5] = {11, -17, 29, 31, 101};
    __asm
```

```

{
    mov ESI, 8
    mov dword ptr a1[ESI], -1001
}
for (int i1=0; i1<5; i1++)
    printf ("a1[%d]=%d\n", i1, a1[i1]);
getchar();
return 0;
}

```

Rezultāts

a1[0]=11 a1[1]=-17 a1[2]=-1001 a1[3]=31 a1[4]=101

Komentāri

Šajā režīmā efektīvā adrese tiek veidota pēc principa „indekss + nobīde”. Nobīde šajā adresācijas veidā norāda uz skaitļu masīva vai rindas sākumu, bet indeksu reģistrs satur datu elementa numuru. Piemēram, programmas fragmentā trešajā masīva elementā tiek ierakstīts skaitlis -1001.



5.6. piemērs. Adresācija „(indekss x reizinātājs) + nobīde”

```

#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
    int a1[5] = {11, -17, 29, 31, 101};
    __asm
    {
        mov ESI, 4
        mov dword ptr a1[ESI*4], -1001
    }
    for (int i1=0; i1<5; i1++)
        printf ("a1[%d]=%d\n", i1, a1[i1]);
    getchar();
    return 0;
}

```

Rezultāts

a1[0]=11 a1[1]=-17 a1[2]=-29 a1[3]=31 a1[4]=-1001

Komentāri

Šajā režīmā efektīvā adrese tiek veidota pēc principa „(indekss x reizinātājs) + nobīde”. Reizinātāju parasti izmanto pieejai elementiem, kuru izmērs dalās ar 2, piemēram, vārdam, dubultvārdam. Nobīde norāda uz skaitļu masīva vai rindas sākumu, bet indeksu reģistrs satur datu elementa numuru. Šis piemērs ir 5.5. piemēra modifikācija. Izmantojot reizinātāja vērtību 4, komanda *mov dword ptr a1[ESI*4], -1001* ierakstīs skaitli -1001 masīva *a1* pēdējā elementā.



5.7. piemērs. Adresācija „bāze + indekss + nobīde”

```

#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
    int a1[8] = {11, -17, 29, 31, 101, -716, 82, -119};
    __asm
    {
        mov ESI, 4
        mov dword ptr a1[ESI]+8, -1001
    }
    for (int i1=0; i1<8; i1++)
        printf ("a1[%d]=%d\n", i1, a1[i1]);
    getchar();
    return 0;
}

```

Rezultāts

a1[0]=11 a1[1]=-17 a1[2]=29 a1[3]=101 a1[4]=-101 a1[5]=-716 a1[6]=82 a1[7]=-119

Komentāri

Šajā režīmā efektīvā adrese tiek veidota pēc principa „bāze + indekss + nobīde”. Tādu adresācijas veidu parasti izmanto divdimensiju masīvu elementu adresācijai vai pieejai pie rindu masīva atsevišķiem elementiem.



5.8. piemērs. Adresācija „bāze + (indekss x reizinātājs) + nobīde”

```
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
    int a1[3][5] = {11, -17, 29, 31, 101, -716, 82, -119, 5, 32, -7, 541, 4, 87, 902};

    __asm
    {
        lea EBX, dword ptr a1
        mov ESI, 5
        mov dword ptr [EBX][ESI*4]+8, -17777
    }
    for (int r1=0; r1<3; r1++)
        for (int c1=0; c1<5; c1++)
            printf ("a1[%d][%d]=%d\n", r1, c1, a1[r1][c1]);
    getch();
    return 0;
}
```

Rezultāts

a[0][0]=11	a[0][1]=-17	a[0][2]=29	a[0][3]=31	a[0][4]=101
a[1][0]=-716	a[1][1]=82	a[1][2]=-17777	a[1][3]=5	a[1][4]=32
a[2][0]=-7	a[2][1]=541	a[2][2]=4	a[2][3]=87	a[2][4]=902

Komentāri

Šajā režīmā efektīvā adrese tiek veidota pēc principa „bāze + (indekss x reizinātājs) + nobīde”. Programmā izmantots divdimensiju masīvs *a1* ar 15 elementiem. Pieņem, ka jāievieto skaitlis -17777 masīva *a1[1][2]* elementā. Šajā gadījumā reģistrā EBX ievieto masīva *a1* adresi: *lea EBX, dword ptr a1*. Reģistrā ESI ievieto skaitli 5 (masīva elementu skaitu rindā). Izmantojot šo vērtību, pareizinātu ar 4, var iegūt pieeju jebkurai masīva rindai.

Lai piekļūtu pie elementa *a1[1][2]* - jāpāriet uz otrās rindas sākumu, jānovieto norāde uz trešā elementa un jāievieto masīva elementā vajadzīgā vērtība:

```
mov dword ptr [EBX][ESI*4]+8, -17777
```

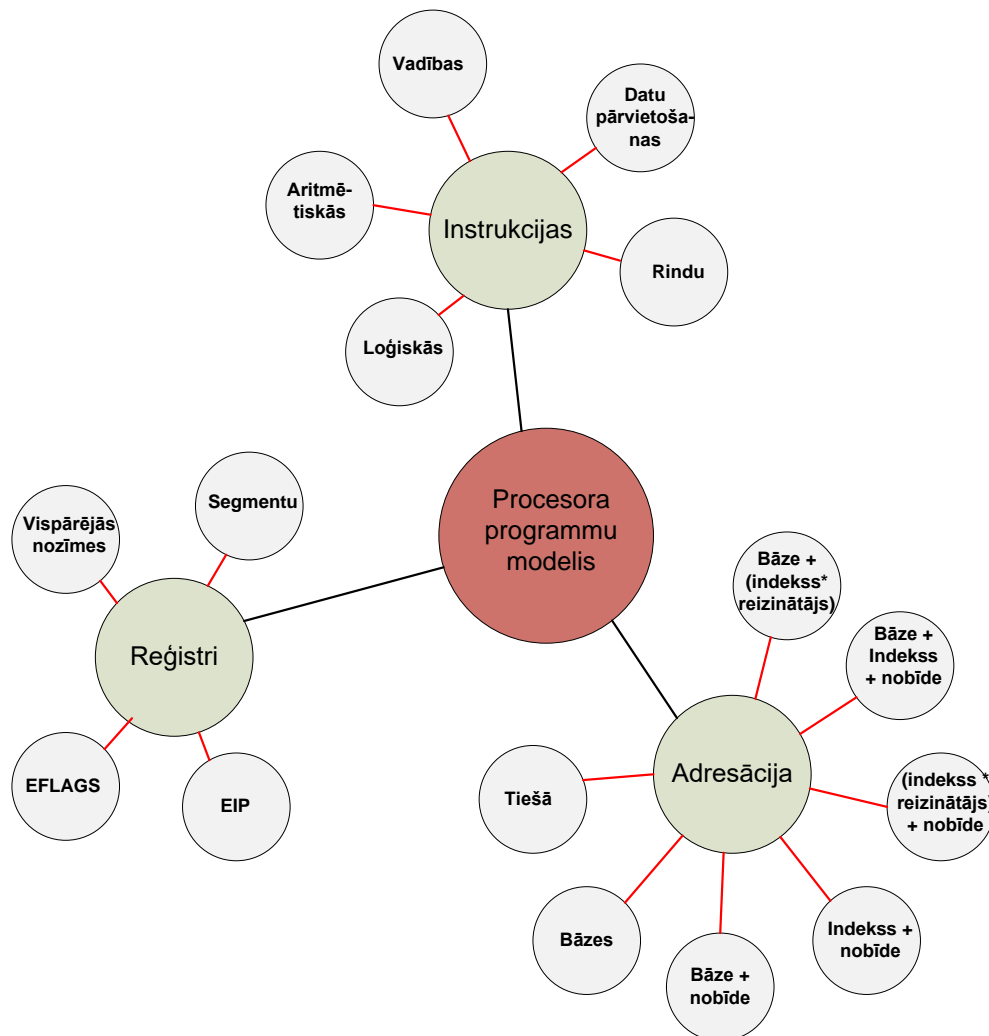
Pārsvārā programmētājiem nav vajadzības manipulēt ar asamblera valodas palīdzību tādā veidā, kā tika apskatīts piemēros. Augsta līmeņa programmēšanas valodas piedāvā ērtus līdzekļus datu apstrādei. Taču bieži programmu lāgošanas gadījumos var noderēt adresācijas režīmu zināšana un datu apstrāde ar asamblera palīdzību.

NODAĻAS KOPSAVILKUMS

- x86 procesora bāzes programmas modelis – tie ir procesora resursi.
- Bāzes programmas modelis satur:
 - 8 vispārējās nozīmes reģistrus datu un rādītāju uzglabāšanai;
 - segmentu reģistrus 6 segmentu selektoru uzglabāšanai;
 - vadības un stāvokļa reģistru EFLAGS, ar kura palīdzību notiek programmu izpildes stāvokļa un procesora stāvokļa vadība;
 - reģistru-norādi EIP, kas satur nākamo izpildāmo procesora instrukciju;
 - procesora komandu jeb instrukciju sistēmu;
 - datu adresācijas režīmus, ko izmanto procesora komandas.

- Selektors ir speciāla norāde, kas identificē segmentu atmiņā.
- Terminam „segments” ir divējāda nozīme. Ar to var apzīmēt iepriekš noteikta apjoma fiziskās atmiņas apgabalu vai mainīga izmēra atmiņas apgabalu, kurā var atrasties programmas kods, dati vai steks.
- Operanda adreses noteikšanas metode tiek saukta par adresācijas režīmu jeb adresāciju.
- Operanda adrese tiek veidota pēc shēmas „segments : nobīde”.
- Formula efektīvās adreses izskaitļošanai ir šāda: $EA = Bāze + Indeks \times Reizinātājs + Nobīde$.
- Iespējamie adresācijas režīmi:
 - tiešā adresācija;
 - bāzes adresācija;
 - bāze + nobīde;
 - indekss + nobīde;
 - (indekss x reizinātājs) + nobīde;
 - bāze + indekss + nobīde;
 - bāze + (indekss x reizinātājs).

5.6.attēlā parādīts piektajā nodaļā minēto svarīgāko jēdzienu koks.



5.6. att. Nodaļas svarīgāko jēdzienu koks



Uzdevumi un jautājumi patstāvīgajam darbam

1. Kāda ir segmentu selektora būtība?
2. Paskaidrot karodziņu reģistra funkcijas programmās.
3. Pēc kāda principa procesors izvēlas nākamo izpildāmo komandu?
4. Iztirzāt segmentēšanas būtību.
5. Raksturot adresācijas režīmus.
6. Raksturot efektīvās adreses izskaitļošanas būtību.
7. Kā skaitļotājā, kurā nav komandas CLR (*clear*), var attīrīt atmiņas vārdu?
8. Kādā veidā pamainīt vietām divu mainīgo saturu – neizmantojot trešo mainīgo vai reģistru?

6. INTEL PROCESORU MULTIVIDES PAPLAŠINĀJUMS

Šajā nodaļā ir apskatītas Intel izstrādātā SIMD datu apstrādes tehnoloģija. SIMD tehnoloģija ir bāzes arhitektūras IA-32 paplašinājums un satur papildus reģistrus, datu tipus un komandas. Galvenais šo paplašinājumu iekļaušanas mērķis – panākt augstāku ražīgumu multivides pielietojumos, kā arī datu pārsūtīšanas un apstrādes sistēmās.

Praktiskajā nozīmē SIMD ir realizēta divu savstarpēji saistītu datu apstrādes tehnoloģiju veidā:

- MMX tehnoloģija, ar kuras palīdzību tiek veikta augsti efektīva 64-bitu datu apstrāde;
- SSE tehnoloģija, kas paredzēta 128-bitu datu apstrādei.

Šīs tehnoloģijas dod iespēju izstrādāt pielietojumus šādiem uzdevumiem:

- signālu kodēšana, dekodēšana un apstrāde;
- runas atpazīšana;
- videosignālu apstrāde;
- 3-D grafisko objektu apstrāde;
- 3-D skaņas apstrāde;
- projektēšana (CAD/CAM).

SIMD arhitektūras galvenā priekšrocība ir tāda, ka ar tās palīdzību var veikt datu paralēlās apstrādes operācijas, kas ļauj palielināt programmas koda ātrdarbību. Asamblera izmantošana SIMD pielietojumos dod iespēju uzrakstīt kompaktu un ātru programmu ar speciālu procesora instrukciju palīdzību SIMD paplašinājumiem.

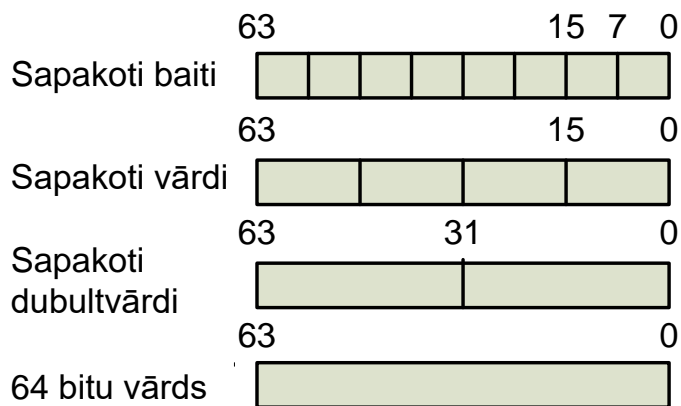
6.1. MMX tehnoloģija

Procesora MMX paplašinājuma instrukcijas strādā ar jauniem datu tiem: 64 bitu datiem, kā arī ar sapakotiem datiem (kopējais izmērs - 64 biti). Dati var atrasties atmiņā vai arī astoņos reģistros, kurus apzīmē MMX0 – MMX7.

MMX paplašinājuma komandas tiek izpildītas pilnīgi tādā pašā veidā kā komandas ar peldošo punktu, t.i., procesors neprasa sevišķu darba režīmu vai papildus aparatūras resursus [9].

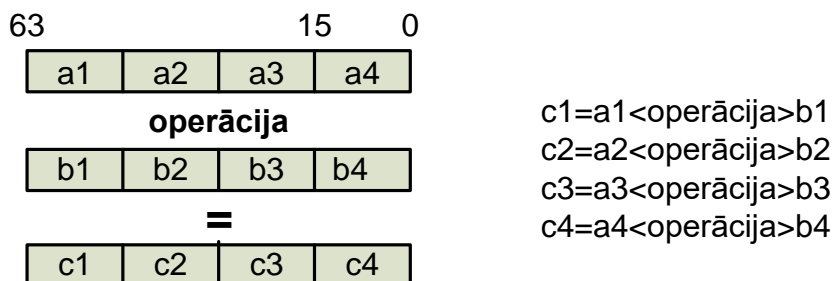
i MMX komandas nodrošina vairāku baitu, vārdu vai dubultvārdu paralēlu apstrādi, operējot ar šādiem datu tiem (sk. 6.1 att.):

- sapakotiem baitiem (*packed byte*) - viens 64 bitu reģistrs satur 8 baitus;
- sapakotiem vārdiem (*packed word*) – viens 64 bitu reģistrs satur četrus 16 bitu vārdus;
- sapakotiem dubultvārdiem (*packed doubleword*) – viens 64 bitu reģistrs satur divus 32 bitu vārdus;
- 64 bitu vārdiem (*quadword*).



6.1. att. MMX datu formāti

Operācijas tiek izpildītas vienlaicīgi ar visām 64 bitu vērtības daļām. 6.2. attēlā parādīta MMX paplašinājuma operāciju izpildes shēma 4 vārdu operandam.



6.2. att. MMX instrukciju izpildes shēma

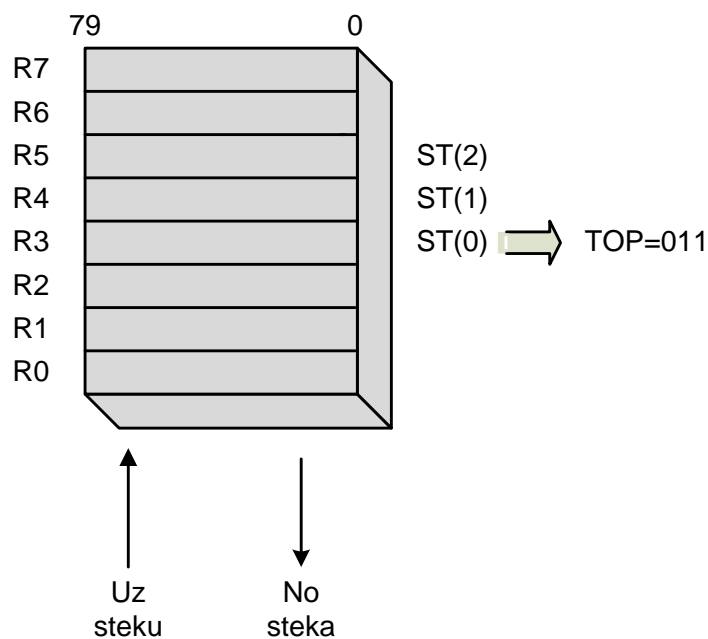
! Darbam ar MMX komandām tiek izmantoti matemātiskā līdzprocesora steka reģistri R0 – R7. Turklāt 80 bitu vietā tiek izmantoti tikai 64 biti, bet steka organizācija, kas obligāta līdzprocesoram, netiek izmantota.

Turpmākamajam izklāstam nepieciešamas pamatzināšanas par līdzprocesora darbības principiem.

i Matemātisko līdzprocesoru sastāda reģistru grupa, datu tipi un komandas. Līdzprocesora reģistru grupā ietilpst:

- astoņi 80 bitu reģistri, kas organizēti reģistru steka veidā;
- trīs dienesta 16 bitu reģistri: stāvokļa reģistrs *swr* (*status word register*), vadības reģistrs *cwr* (*control word register*) un tagu reģistrs *twr* (*tag word register*). Tags - viena vai vairākas rakstzīmes, kas piekārtotas datu kopai, lai to identificētu un sniegtu informāciju par kopu;
- datu reģistrs-rādītājs *dpr* (*data point register*) un komandu reģistrs-rādītājs *ipr* (*instruction point register*), kas tiek pielietoti izņēmuma situāciju apstrādē.

Visi līdzprocesora reģistri pieejami ar speciālu līdzprocesora komandu palīdzību. No aparātūras viedokļa līdzprocesors satur astoņus 80 bitu adresējamus reģistrus, ko sauc par datu reģistriem un kas veido reģistru steku. Reģistri tiek numurēti no R0 līdz R7 (sk. 6.3. att.).



6.3. att. Līdzprocesora reģistru steks

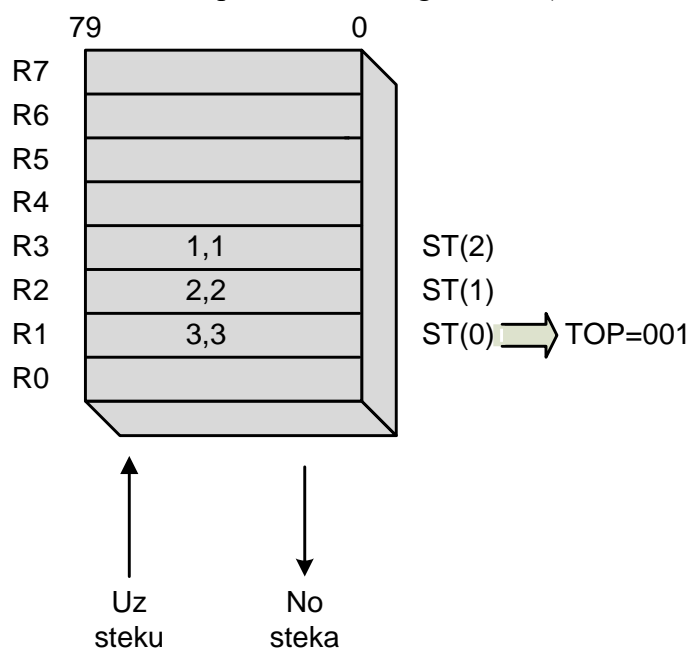
Līdzprocesora komandas adresē datu reģistrus atbilstoši tam reģistram, kurš ir steka virsotnē. Attiecīgajā laika momentā šī reģistra numurs tiek ierakstīts līdzprocesora stāvokļa reģistrā *swr* trīs bitu laukā TOP.

Kā redzams 6.3. attēlā, TOP vērtība ir 011, kas norāda uz reģistru R3 kā steka virsotni. Steka virsotni apzīmē ar ST(0) vai vienkārši ST. Pārējie reģistri tādā gadījumā tiek apzīmēti ar ST(1), ST(2) utt.

Kad dati tiek ievietoti stekā, TOP vērtība tiek samazināta par 1 un operands tiek ievietots reģistrā, kas pašlaik ir steka virsotne un tiks apzīmēts kā ST(0). Ja pirms datu ielādes TOP vērtība ir 0, tad nākamā steka virsotne būs reģistrs R7.

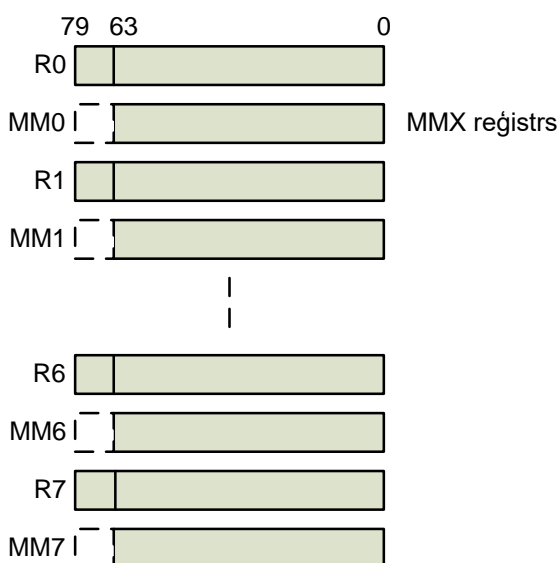
Ja dati tiek izņemti no steka, TOP vērtība tiek palielināta par 1. Ja pirms datu izņemšanas TOP=7, tad pēc operācijas tā vērtība būs 0 un par steka virsotni tiks izvēlēts reģistrs R0.

Ja, piemēram, stekā no 6.3. attēla tiks secīgi ievietoti skaitļi 1,1, 2,2 un 3,3, tad steka stāvoklis izmainīsies un steka virsotne tiks pārvietota uz reģistru R1 (sk. 6.4. att.):



6.4. att. Līdzprocesora reģistru steks pēc datu ielādes

MMX operācijās reģistru steks tiek traktēts kā grupa ar 8 neatkarīgiem 64 bitu reģistriem (sk. 6.5. att.).



6.5. att. Līdzprocesora un MMX reģistru atbilstība

Tādējādi MMX paplašinājums izmanto tos pašus reģistrus, ko izmanto līdzprocesors. Visas MMX instrukcijas izpildās tādā pašā veidā kā komandām ar peldošo punktu, kas izraisa līdzprocesora stāvokļa reģistra *swr* izmaiņas. Šī iemesla dēļ pēdējai MMX paplašinājuma komandai jābūt *emms*, jo tā nodrošina korektu procesora pāreju no programmas koda izpildes ar MMX instrukcijām uz parasto komandu apstrādi ar peldošo punktu.

Instrukcija *emms* iestāda vērtību 1 visās stāvokļa reģistra *swr* pozīcijās. Ja *emms* nav uzrādīts, tad visas nākamās operācijas ar peldošo punktu var uzrādīt nekorektus rezultātus, izsaucot stāvokli *stack overflow*.

Vairākums MMX paplašinājuma komandu strādā ar diviem operandiem - ieejas (operands-avots) un izejas (operands-mērķis). Parasti ieejas informācija tiek paņemta no abiem operandiem, un rezultāts tiek ierakstīts operandā-mērķī.

MMX instrukcijas veic datu apstrādi divos veidos - vai nu izmantojot t.s. ciklisko aritmētiku (*wraparound arithmetic*), vai t.s. piesātināšanas aritmētiku (*saturation arithmetic*). Pārsvārā visas MMX instrukcijas izmanto ciklisko aritmētiku – ja cikliskās aritmētikas operācijas rezultāts iziet ārpus izmantojamā datu tipa robežām, tad „liekie” vecākie rezultāta biti tiek atmesti (piemēram, baitu 01h un FFh saskaitīšanas rezultāts būs 00h). Ja piesātinātās aritmētikas operācijas rezultāts pārsniedz dotā datu tipa maksimālo vērtību, tad operandā-mērķī tiek ierakstīta šī maksimālā vērtība. Analogiski, ja rezultāts ir mazāks par minimālo vērtību, tad operandā-mērķī tiek ierakstīta minimālā vērtība (piemēram, ja rezultāts ir mazāks par 8000h, tad 16-bitu vārds ar zīmi ir vienāds ar 8000h; ja iegūtā vērtība ir lielāka par 7FFFh, tad vārda ar zīmi vērtība ir vienāda ar 7FFFh).

MMX instrukcijas satur sufiksu komandas nosaukumā, kas nosaka datu tipu un izpildāmās aritmētikas tipu:

- *us* (*unsigned saturation*) - aritmētika ar piesātinājumu, dati bez zīmes;
- *s* vai *ss* (*signed saturation*) – aritmētika ar piesātinājumu, dati ar zīmi;
- ja sufiksā nav ne *s*, ne *ss* – tad tiek izmantota cikliskā aritmētika;
- *b,w,d,q* – norāda datu tipus. Ja sufiksā ir abi no šiem burtiem – pirmais atbilst operandam-avotam, otrais atbilst operandam-mērķim.

Piemēram, instrukcija *paddusw MM1, var1* saskaita vārdus bez zīmes. Sufikss *us* norāda uz aritmētiku ar piesātinājumu bez zīmes, bet pēdējais simbols *w* nosaka, ka operandi ir vārdi. Rezultāts tiek saglabāts reģistrā MM1.

Visas MMX instrukcijas var nosacīti iedalīt šādās grupās:

- saskaitīšanas, atņemšanas un reizināšanas instrukcijas;
- nobīdes instrukcijas;
- loģisko operāciju instrukcijas;
- salīdzināšanas instrukcijas;
- sapakošanas un atpakošanas instrukcijas;
- datu pārsūtīšanas instrukcijas.



Pirms sākt darbu ar MMX instrukciju izmantošanu programmās, ir jāpārlicinās – vai procesors uztur MMX paplašinājumu.



6.1. piemērs. Procesora pārbaude uz MMX paplašinājuma atbalstu

```
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
    bool supMMX = true;
    __asm
    {
        mov EAX, 1
        cpuid
        test EDX, 800000h
        jnz exit
        mov supMMX, 0
    exit:
    }
    if (supMMX)
        printf("MMX tiek uztureets!!\n");
    else
        printf("MMX netiek uztureets!!\n");

    getch();
    return 0;
}
```

Rezultāts

Loģiskā mainīgā *supMMX* patiesuma vērtība nemainās un tiek izvadīts paziņojums „MMX tiek uzturēts”.

Komentāri

Tiek izmantota procesora instrukcija *cpuid*, bet pirms tam reģistrā EAX iesūta skaitli 1. Pēc izpildes tiek pārbaudīts reģistra EAX 23 bits (ja 1 – tad procesors uztur MMX paplašinājumu).

Turpmākajos piemēros tiks demonstrēta MMX paplašinājuma pamata instrukciju pielietošana.

Datu pārsūtīšanas instrukcija *movd* ļauj kopēt 32 bitu skaitli no viena MMX reģistra jaunākajiem bitiem uz cita reģistra jaunākajiem bitiem (vecākie biti tiek aizpildīti ar nullēm). Instrukcija *movq* dod iespēju kopēt 64 bitu operandus:

- no viena MMX reģistra uz citu;
- no atmiņas MMX reģistrā;
- no MMX reģistra atmiņā.



6.2. piemērs. Datu pārsūtīšana

```
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
    int i1[2]={34, 18};
    int a1[2];
    __asm
```

```

{
    movq MM0, qword ptr i1
    movq qword ptr a1, MM0
    add dword ptr a1, 100
    sub dword ptr a1+4, 100
    emms
}
printf ("a1[0]= %d, a1[1]= %d\n", a1[0], a1[1]);
getchar();
return 0;
}

```

Rezultāts

a1[0]= 134 a1[1]=-82

Komentāri

Tiek kopēti divi dubultvārdi no masīva *i1* uz masīvu *a1*. Komandās izmanto direktīvu *qword* (četrkāršots vārds). Tālāk pirmajam masīva elementam pieskaita vērtību 110, no otrā masīva elementa atņem 100. Pieejai masīva *a1* otrajam elementam instrukcijā *sub* tiek izmantota nobīde +4 attiecībā uz masīva sākumu.



6.3. piemērs. Datu paralēla saskaitīšana ar iebūvēto funkciju palīdzību

```

#include "stdafx.h"
#include <mmintrin.h>
int _tmain(int argc, _TCHAR* argv[])
{
    int i1[2]= {45, 12};
    int i2[2]= {11, -78};
    __m64 smm1= _mm_set_pi32 (i1[1], i1[0]);
    __m64 smm2= _mm_set_pi32 (i2[1], i2[0]);
    __m64 res= _mm_add_pi32 (smm1, smm2);

    int *psum= (int*)&res;
    _mm_empty();
    printf("i1[0] + i2[0]= %d\n", *psum);
    printf("i1[1] + i2[1]= %d\n", *(++psum));
    getchar();
    return 0;
}

```

Komentāri

Tiek veikta masīvu *i1* un *i2* paralēla saskaitīšana, turklāt saskaitīšanas rezultāts tiek ievietots tipa *__m64* mainīgajā *res*. Šajā piemērā tiek izmantotas *Visual Studio 2005* iebūvētās funkcijas *intrinsics* no bibliotēkas *mmintrin.h* - *_mm_set_pi32* un *_mm_add_pi32*.

Funkcija *_mm_set_pi32* ieraksta divu 32 bitu mainīgo vērtības 64 bitu mainīgā vecākajā un jaunākajā daļā. (Šai funkcijai nav analoga assemblera valodā, tā dota kā iebūvētās funkcijas piemērs).

Funkcija *_mm_add_pi32* saskaita divus *__m64* tipa mainīgos un ieraksta rezultātu tāda paša tipa mainīgajā (Šai funkcijai ir procesora instrukcijas analogs *paddd*).

Visām MMX paplašinājuma operācijām jābeidzas ar procesora instrukciju *emms* (starp iebūvētajām funkcijām ir analogs *_mm_empty*, kas arī pabeidz MMX operācijas).

Operators *int *psum=(int*)&res* definē norādi *psum* uz vesela tipa mainīgo, kurš satur 8 32 bitu skaitļus. Tas dod iespēju programmai vērsties pie 64 bitu mainīgā *res* atsevišķiem 32 bitu komponentiem.



6.4. piemērs. Paralēlā datu saskaitīšana (modificēts 6.3. piemēra variants ar procesora MMX instrukciju izmantošanu)

```

#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])

```

```

{
    int i1[2]= {45, 12};
    int i2[2]= {11, -78};
    int res[2];
    asm
    {
        movq MM0, mmword ptr i1
        padd MM0, mmword ptr i2
        movq mmword ptr res, MM0
        emms
    }
    printf("res[0]= %d\n", res[0]);
    printf("res[1]= %d\n", res[1]);
    getchar();
    return 0;
}

```

Komentāri

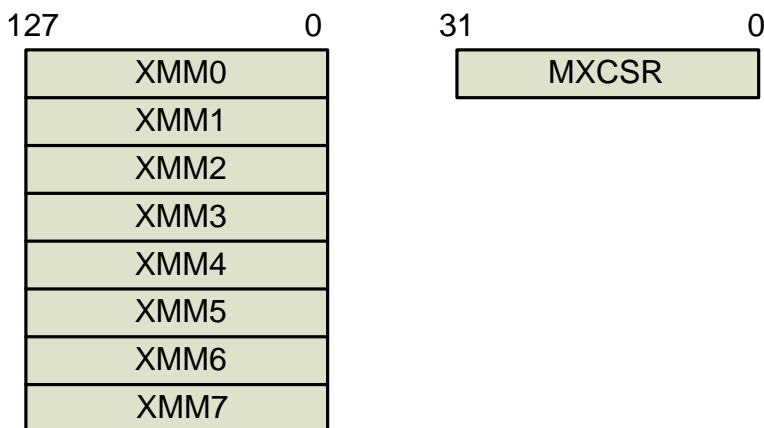
Šī programma veic tieši to pašu, ko iepriekšējā programma, bet programmas kods ir vienkāršāks.

Dotie piemēri dod ieskatu par MMX instrukciju pielietošanu (detalizētāka informācija jāmeklē Intel dokumentācijā [19]).

6.2. SSE un SSE2 tehnoloģija

SSE paplašinājums turpina MMX paplašinājumu, ļaujot apstrādāt datus ar peldošo punktu.

i SSE paplašinājums realizēts kā aparatūras un programmu modulis, kas iever astoņus 128 bitu reģistrus (XMM0-XMM7) un 32 bitu vadības/stāvokļa reģistru MXCSR (sk. 6.6. att.).

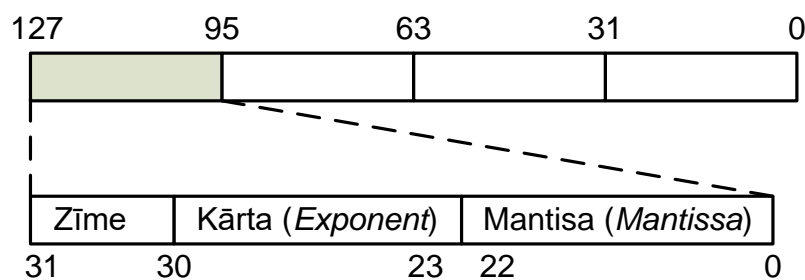


6.6.att. SSE paplašinājuma reģistri

SSE paplašinājuma programmu daļa ietver SSE instrukcijas ar datiem peldošā punkta formātā. XMM reģistrs satāv no četriem 32 bitu operandiem ar peldošo punktu īsajā formātā SPFP (*single precision floating point*). Pats datu formāts parādīts 6.7. attēlā (parādīts tikai vecākā dubultvārda formāts).

Mantisa (*mantissa*) un kārtas (*exponent* vai *exp*) veido skaitli SPFP formātā atbilstoši formulai: $\text{mantissa} \cdot 2^{\text{exp}}$. Tādā veidā var attēlot skaitļus diapazonā $2^{-126} - 2^{127}$.

SSE modulis realizēts neatkarīgi no citām procesora ierīcēm, kas dod iespēju izpildīt SSE instrukcijas paralēli līdzprocesora vai procesora komandām. Izmantojot SSE, nav vajadzīga sinhronizācija ar citām ierīcēm (tāpēc līdzīga instrukcija kā *emms* nav nepieciešama).

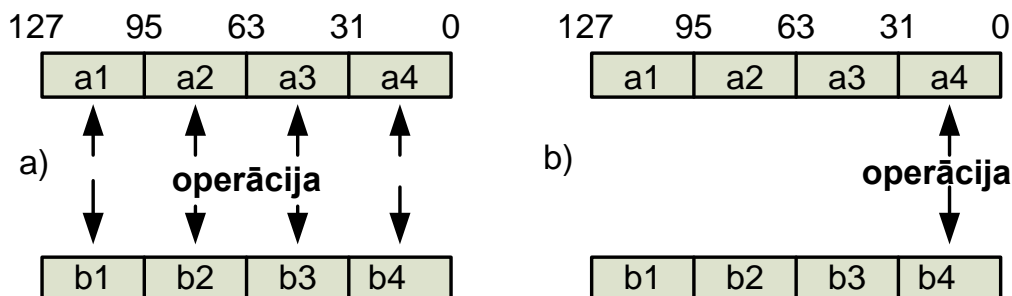


6.7.att. SSE paplašinājuma datu formāts

SSE paplašinājuma instrukciju kopa ietver 70 komandas (SSE arhitektūras un instrukciju aprakstu meklēt procesoru *Pentium* III un jaunāku modeļu *Intel* dokumentācijā).

Vairākums SSE instrukciju var tikt izpildītas divos režīmos - skalārajā un paralēlajā. Paralēlā tipa instrukcijas vienlaicīgi apstrādā 4 dubultvārdus un savā mnemonikā ietver sufiksu *ps* (sk. 6.8. a) attēlā).

Skalārās operācijas apstrādā tikai operandu jaunāko dubultvārdu un savā mnemonikā ietver sufiksu *ss* (tādu komandu izpildes shēma parādīta 6.8.b) attēlā).



6.8.att. Datu paralēlās (a) un skalārās apstrādes (b) shēma

SSE instrukciju darbības laikā var notikt izņēmuma stāvokļi šādu darbību rezultātā:

- nekorekta operācija (*invalid operation*);
- denormalizēts operands (*denormalised operand*);
- dalīšana ar 0 (*divide by zero*);
- aritmētiskā pārpildīšanās (*numeric overflow*);
- zīmīgo ciparu nozaudēšana (*numeric underflow*);
- precizitātes zaudēšana (*inexact result*);

Tādās situācijās tie iestatīti attiecīgi karodziņi reģistra MXCSR 0-5 bits.

Nosacīti SSE paplašinājuma instrukcijas var sagrupēt šādās grupās:

- datu pārsūtīšana;
- aritmētiskās komandas;
- salīdzināšana;
- pārveidošana;
- loģiskās;
- papildus komandas.



Pirms sākt darbu ar SSE instrukciju izmantošanu programmās, ir jāpārlicinās – vai procesors uztur SSE paplašinājumu.



6.5. piemērs. Procesora pārbaude uz SSE paplašinājuma atbalstu

```
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
    bool supSSE = true;
    __asm
    {
        mov EAX, 1
        cpuid
        test EDX, 2000000h
        jnz ex
        mov supSSE, 0
    ex:
    }
    if (supSSE)
        printf("SSE tiek uztureets!!\n");
    else
        printf("SSE netiek uztureets!!\n");
    getchar();
    return 0;
}
```

Tālāk ir doti vienkārši piemēri SSE paplašinājuma instrukciju izmantošanai atņemšanas un saskaitīšanas operācijās.



6.6. piemērs. Datu atņemšana ar SSE paralēlo instrukciju palīdzību

```
#include "stdafx.h"
#include <stdio.h>
int main(void)
{
    float src[8]={34.9, -67.13, -47.57, 92.3};
    float dst[8]={-6.99, 23.19, -409.56, 1.03};
    float res[8];
    __asm
    {
        movups XMM0, xmmword ptr src
        movups XMM1, xmmword ptr dst
        subps XMM0, XMM1
        movups xmmword ptr res, XMM0
    }
    for(int i1=0; i1<4; i1++)
        printf("res[%d]=%5.2f\n", i1, res[i1]);
    getchar();
    return 0;
}
```

Rezultāts

res[0]=41,89 res[1]=-90,32 res[2]=361,99 res[3]=91,27

Komentāri

Atņemšanas komanda *subps* izpilda paralēlu 128 bitu operandu atņemšanu (*addps* veic saskaitīšanu).



6.7. piemērs. Atņemšana ar iebūvēto funkciju palīdzību

```
#include "stdafx.h"
#include <xmmintrin.h>
int _tmain(int argc, _TCHAR* argv[])
{
    float src[8]={34.9, -67.13, -47.57, 92.3};
    float dst[8]={-6.99, 23.19, -409.56, 1.03};
```

```

float res[8];
__m128 xsrc, xdst, xres;
xsrc= _mm_loadu_ps(src);
xdst= _mm_loadu_ps(dst);
xres= _mm_sub_ps(xsrc, xdst);
_mm_storeu_ps(res, xres);

for(int i1=0; i1<4; i1++)
    printf("res[%d]=%5.2f\n", i1, res[i1]);
    getchar();
return 0;
}

```

Rezultāts

Analoģisks 6.6. piemēra rezultātam.

Komentāri

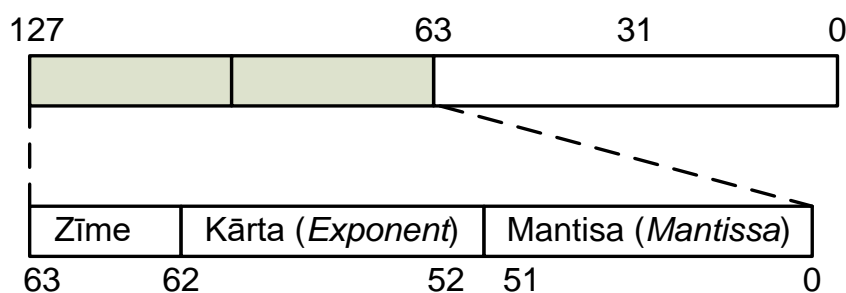
Līdzīgi kā MMX paplašinājumam *Visual Studio 2005* ir vesela virkne iebūvēto funkciju darbam ar SSE paplašinājumu. Visas šīs funkcijas izmanto speciālu mainīgo tipu `__m128`. Šī tipa mainīgie tiek izlīdzināti uz 16 bitu robežas.

SSE paplašinājums ļauj paātrināt lietojumprogrammu darbu lielu informācijas apjomu apstrādē, jo dati var tikt apstrādāti paralēli vienā ciklā. SSE2 tehnoloģija ir paralēlās datu apstrādes tehnoloģijas turpinājums. Tās nozīme – palielināt 128-bitu datu apstrādes efektivitāti peldošā punkta operācijās skaitļiem ar dubulto precizitāti (*double-precision floating point*). Tas tiek panākts ar jaunu datu tipu ieviešanu – 128 bitu dubultās precizitātes operandi ar peldošo punktu un 128 bitu sarakotiem veseliem skaitļiem.

SSE2 tehnoloģija dod iespēju paaugstināt skaitļojumu efektivitāti:

- uz datu vadības uzlabošanas kešatmiņā rēķina;
- palielinot to operāciju veikspēju, kam nepieciešama lielāka precizitāte;
- paplašinot 64 bitu komandu iespējas līdz 128 bitu komandām.

SSE2 par bāzes formātu kalpo 64 bitu dubultās precizitātes skaitļi ar peldošo punktu (sk. 6.9. att.).



6.9.att. Sapakota 64 bitu dubultās precizitātes skaitļa ar peldošo punktu formāts

SSE2 instrukcijas izmanto astoņus 128 bitu reģistrus (XMM0-XMM7) un var strādāt gan skalārajā, gan paralēlajā režīmā, operējot ar šādiem datu tiem:

- sarakotiem un skalāriem skaitļiem ar peldošo punktu īsajā formātā SPFP (*single precision floating point*);
- sarakotiem un skalāriem dubultās precizitātes skaitļiem ar peldošo punktu;
- sarakotiem un skalāriem 128 bitu veseliem skaitļiem.

Ar SSE2 instrukciju palīdzību kļūst iespējams:

- izstrādāt algoritmus, kuros vienlaicīgi var apstrādāt jauktu datu tipus;
- operēt ar dažāda izmēra datiem: baits, vārds, dubultvārds, četrkārtīgs vārds (*quad-word*) un divkārtīgs četrkārtīgs vārds (*double-quad-word*).

SSE2 instrukcijas pārsvarā prasa datu izlīdzināšanu uz 16 bitu robežas. Paralēlo komandu mnemoniskais apzīmējums satur sufiksu *pd*, skalārajām operācijām – *sd*.



Pirms sākt darbu ar SSE2 instrukciju izmantošanu programmās, ir jāpārlicinās, vai procesors uztur SSE2 paplašinājumu.



6.8. piemērs. Procesora pārbaude uz SSE2 paplašinājuma atbalstu

```
#include "stdafx.h"

int _tmain(int argc, _TCHAR* argv[])
{
    bool supSSE2 = true;
    __asm
    {
        mov EAX, 1
        cpuid
        test EDX, 400000h // 26-tais bits
        jnz exit
        mov supSSE2, 0
    exit:
    }
    if (supSSE2)
        printf("SSE2 tiek uztureets!!\n");
    else
        printf("SSE2 netiek uztureets!!\n");
    getchar();
    return 0;
}
```

Komentārs

Pēc komandas *cpuid* tiek analizēts reģistra EDX 26 bits. Ja tas ir atšķirīgs no 0, procesors uztur SSE2 tehnoloģiju.



6.9. piemērs. Skaitļu sareizināšana ar SSE2 instrukciju palīdzību

```
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
    double d1[2]={34.988, -6509.237};
    double d2[2]={-744.501, 2779.419};
    double dres[2];
    __asm
    {
        movupd XMM0, xmmword ptr d1
        movupd XMM1, xmmword ptr d2
        mulpd XMM0, XMM1
        movupd xmmword ptr dres, XMM0
    }
    printf("dres[0]=%6.3f, dres[1]=%6.3f\n", dres[0], dres[1]);
    getchar();
    return 0;
}
```

Rezultāts

dres[0]= -26048,601 dres[1]= -18091896,993

Komentāri

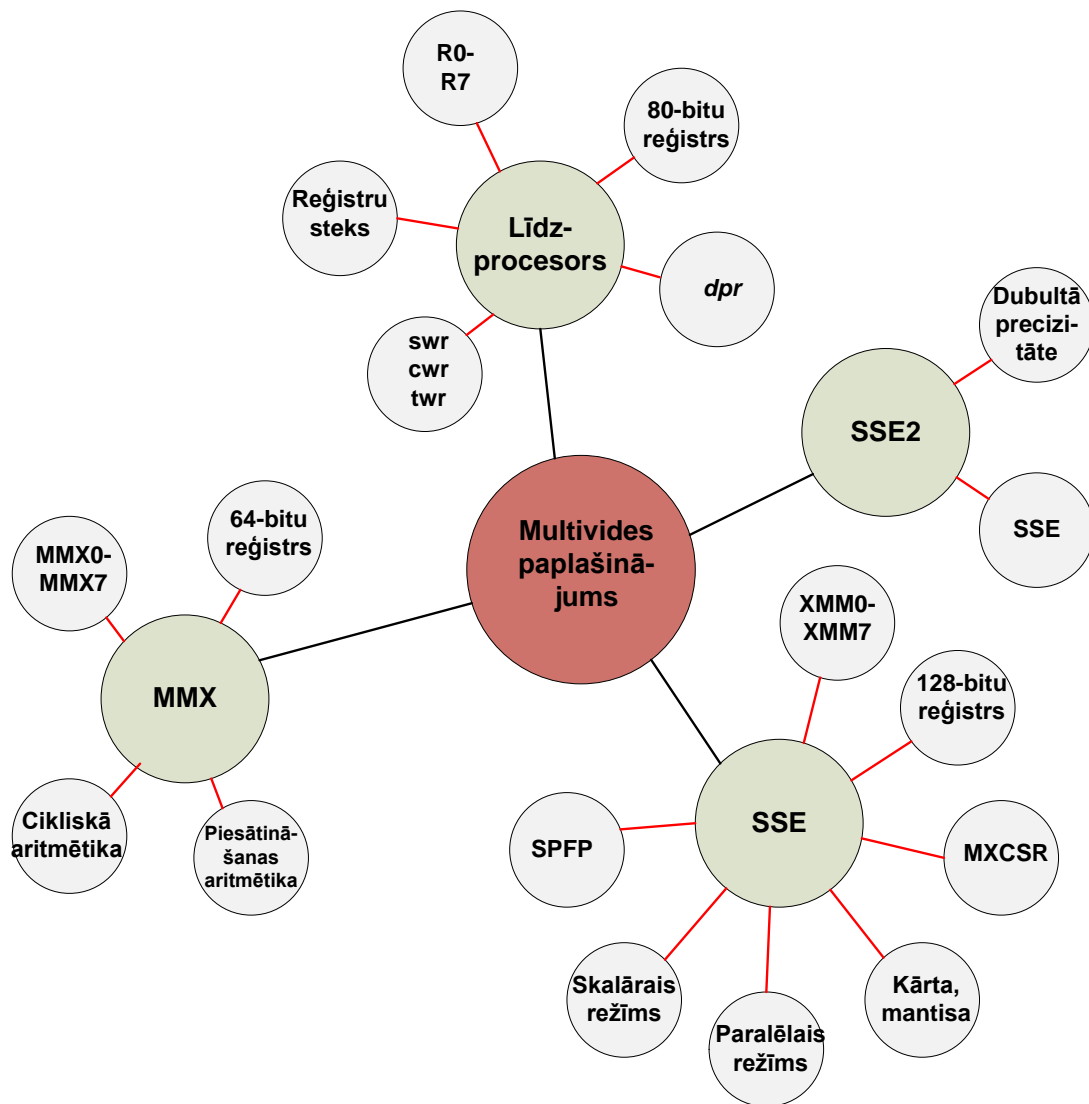
Doti divi dubultās precizitātes skaitļu masīvi ar peldošo punktu. Piemērā paralēli pa pāriem tiek sareizināti divu masīvu elementi.

Visos šajos piemēros tika apskatīta tikai neliela daļa no MMX, SSE un SSE2 iespējām.

NODAĻAS KOPSAVILKUMS

- Matemātisko līdzprocesoru sastāda reģistru grupa, datu tipi un komandas. Līdzprocesora reģistru grupā ietilpst:
 - astoņi 80 bitu reģistri, kas organizēti reģistru steka veidā;
 - trīs dienesta 16 bitu reģistri: stāvokļa reģistrs *swr (status word register)*, vadības reģistrs *cwr (control word register)* un tagu reģistrs *twr (tag word register)*. Tags - viena vai vairākas rakstzīmes, kas piekārtotas datu kopai, lai to identificētu un sniegtu informāciju par kopu;
 - datu reģistrs-rādītājs *dpr (data point register)* un komandu reģistrs-rādītājs *ipr (instruction point register)*, kas tiek pielietoti izņēmuma situāciju apstrādē.
- MMX komandas nodrošina vairāku baitu, vārdu vai dubultvārdu paralēlu apstrādi, operējot ar šādiem datu tiem:
 - sapakotiem baitiem (*packed byte*) - viens 64 bitu reģistrs satur 8 baitus;
 - sapakotiem vārdiem (*packed word*) – viens 64 bitu reģistrs satur četrus 16 bitu vārdus;
 - sapakotiem dubultvārdiem (*packed doubleword*) – viens 64 bitu reģistrs satur divus 32 bitu vārdus;
 - 64 bitu vārdiem (*quadword*).
- Darbam ar MMX komandām tiek izmantoti matemātiskā līdzprocesora steka reģistri R0 – R7. Turklāt 80 bitu vietā tiek izmantoti tikai 64 biti, bet steka organizācija, kas obligāta līdzprocesoram, netiek izmantota.
- SSE paplašinājums realizēts kā aparatūras un programmu modulis, kas iever astoņus 128 bitu reģistrus (XMM0-XMM7) un 32 bitu vadības/stāvokļa reģistru MXCSR.
- SSE2 tehnoloģija ir paralēlās datu apstrādes tehnoloģijas turpinājums. Tās nozīme – palielināt 128 bitu datu apstrādes efektivitāti peldošā punkta operācijās skaitļiem ar dubulto precizitāti (*double-precision floating point*).

6.10.attēlā parādīts nodaļā minēto svarīgāko jēdzienu koks.



6.10.att. Nodaļas svarīgāko jēdzienu koks

Uzdevumi un jautājumi patstāvīgam darbam

1. Kā ir realizēts paralēlisms MMX instrukcijās?
2. Kādā veidā procesors atšķir MMX komandu no līdzprocesora komandas?
3. Ja jau MMX tehnoloģija izmanto līdzprocesora reģistrus, kāpēc netiek izmantota līdzprocesora steka organizācija?
4. Raksturot līdzprocesora reģistru steka darbības principu.
5. Paskaidrot ar savu piemēru cikliskās aritmētikas operācijas.
6. Raksturot SSE skalārās un paralēlās instrukciju izpildes būtību.
7. Raksturot kopīgo un atšķirīgo SSE un SSE2.

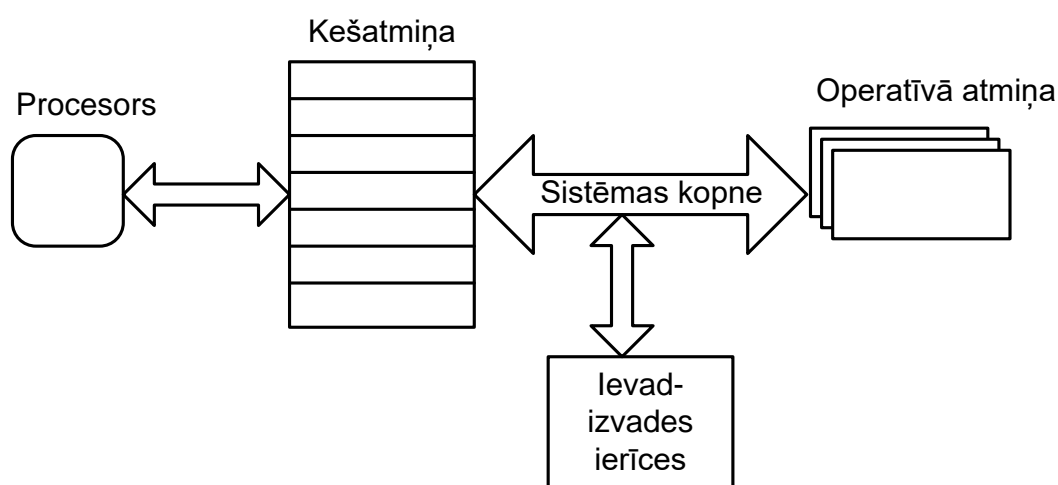
7. DATU UN KOMANDU KEŠDARBE

Mūsdienās neviena datorsistēma nevar iztikt bez kešatmiņas un datu kešdarbes. Datu kešdarbe kā tehnoloģija tiek izmantota datu apmaiņas ražīguma palielināšanai datorsistēmās. Nodaļā tiek apskatīti principi, uz kādiem pamatojas šī tehnoloģija, un analizēta datu apmaiņas organizācijas datorsistēmās praktiskie aspekti.

i Kešdarbe (*cacheing*) – tās aparatūras daļas un programmu risinājums, kas tiek izmantots datu apmaiņas sinhronizācijai un paātrināšanai starp dažādām datorsistēmas ierīcēm, kam ir dažāda ātrdarbība.

Pārsvarā kešdarbe tiek veikta datu apmaiņā starp procesoru un operatīvo atmiņu, kā arī starp procesoru un I/O ierīcēm [3,4,7,9].

Vispārīga kešdarbes shēma parādīta 7.1. attēlā.



7.1. att. Datu kešdarbes vispārīgā shēma

Šī tehnoloģija tiek izmantota arī pašās I/O ierīcēs, piemēram, cieto disku uzkrājējos, kur nepieciešams optimizēt mijiedarbību starp disku mehānisko un elektronisko daļu. Fiziski kešatmiņa ir statiska tipa ierīce, kas nodrošina tās lielu ātrdarbību.

Kešatmiņas apjoms nav pārāk liels, turklāt kešatmiņas ātrdarbība ir apgriezti proporcionāla tās apjomam – jo lielāks apjoms, jo mazāka ātrdarbība. Principā nekas netraucē izstrādāt liela apjoma kešatmiņu ar lielu ātrdarbību – problēma slēpjas tajā apstākļī, ka tāda mikroschéma (ja kešatmiņa tiek realizēta kā atsevišķa ierīce) patērēs lielus jaudas resursus, ko grūti apiet tehnoloģiskajā ziņā. Šī paša iemesla dēļ dotajā tehnoloģiju attīstības līmenī procesorā nav realizējama ļoti ātrdarbīga kešatmiņa. Pašlaik kešdarbes apakšsistēma realizēta procesorā, tam mijiedarbojoties ar operatīvo atmiņu ar procesora kopnes palīdzību.

! Kešdarbes apakšsistēmas izstrādē rodas divas problēmas:

- kešatmiņas optimālās caurlaidības spējas izvēle;
- kešatmiņas optimālā apjoma izvēle.

Kad tiek runāts par kešatmiņas raksturlielumiem – tiek saprasta tā saskarne ar procesoru. Datu apmaiņa ar operatīvo atmiņu, ieskaitot kešatmiņas ielādi un izlādi, galvenokārt atkarīga no sistēmas kopnes saskarnes un operatīvās atmiņas ātrdarbības.

Kešdarbes apakšsistēmas kopnes caurlaidība jeb kopnes izmērs (bitos) atkarīgs no aparatūras platformas un variējas robežās no 4 līdz 256 bitiem. Piemēram, *Intel Pentium 4* procesoros L2 kešatmiņas dati tiek pārraidīti pa iekšējo kopni, kuras izmērs ir 256 biti. Dati

kešatmiņā parasti izvietojas rindu veidā, kuru izmērs var būt vienāds ar kopnes izmēru vai arī pārsniegt to.

Liela nozīme augsta ražīguma sasniegšanā ir kešatmiņas datu bloka optimālais apjoms – kešatmiņai jābūt pietiekami lielai. Ja, piemēram, programma apstrādā datu vārdu, kas atrodas kešatmiņā, tad ar lielu varbūtību tai tuvākajā laikā būs nepieciešami blakus šūnās atrodošies dati.

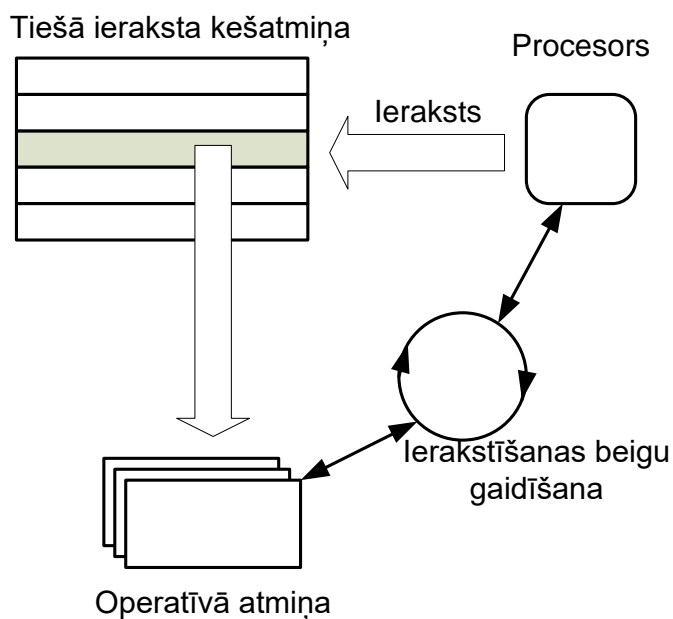
Analoģiski, ja kešatmiņā atrodas programmas koda bloks un tiek izpildīta kāda instrukcija no šī bloka, tad ar lielu varbūtību nākamās izpildīsies komandas augošā adrešu secībā. Liela apjoma kešatmiņas priekšrocības izpaužas arī datu masīvu apstrādē, jo pieeja masīva elementiem notiek secīgi. Tādos gadījumos saka, ka trāpīšanas biežums (*cache hit*) lielākai kešatmiņai būs lielāks.

Tajā pašā laikā pārāk lielam kešatmiņas apjomam ir savi trūkumi. Ja kešatmiņā iztrūkst dati, nākas vairāk laika veltīt nākamās datu porcijas ielādei no nākamā līmeņa kešatmiņas vai operatīvās atmiņas. Ja datu bloka apjoms pārsniedz kešatmiņas apjomu, tad var nākties ielādēt vairākus mazāku apjomu blokus, kas var izraisīt kļūdas (*cache miss*).



Pastāv divas stratēģijas datu ierakstīšanai no kešatmiņas operatīvajā atmiņā – tiešais ieraksts (*write-through*) un apgrieztais ieraksts (*write-back*). Attiecīgi tādas kešatmiņas sauc par kešatmiņu ar tiešo ierakstu un kešatmiņu ar apgriezto ierakstu.

Ja tiek izmantots tiešais ieraksts, tad pēc datu izmaiņas kešatmiņā tiek iniciēts kopnes cikls un dati nekavējoties tiek ierakstīti operatīvajā atmiņā. Šajā gadījumā procesoram jāgaida ieraksta atmiņā cikla beigas (skat. 7.2. att.).

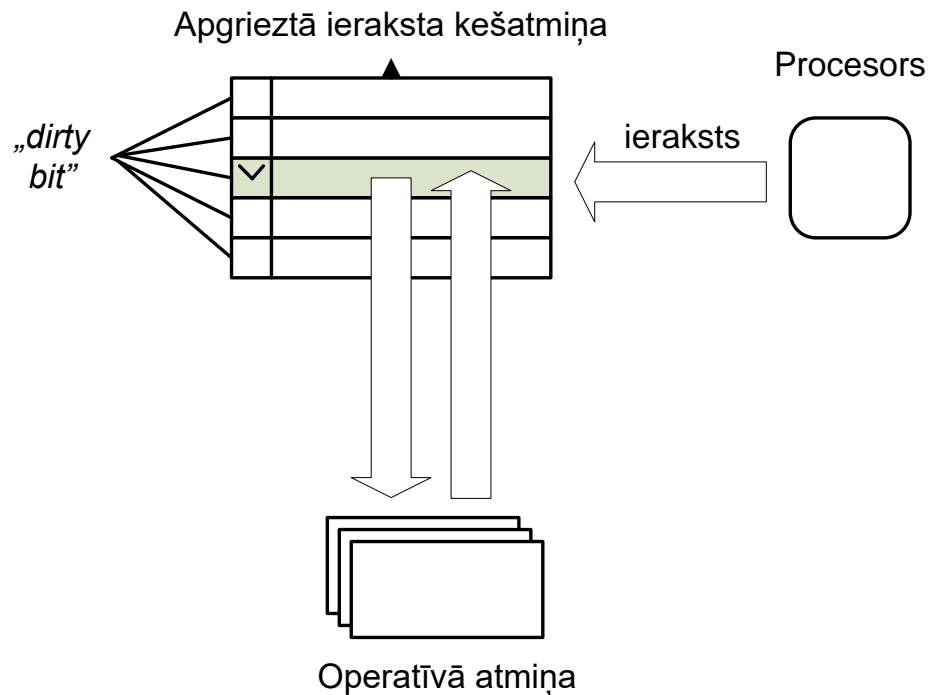


7.2. att. Tiešā ieraksta kešatmiņas funkcionēšana

Apgrieztā ieraksta gadījumā izmainītie dati no kešatmiņas ierakstās asinhroni – kešatmiņas satura ar jaunu datu bloku no atmiņas nomainas laikā. Tādā kešatmiņā katrai rindai tiek piešķirts tā saucamais ieraksta pieprasījuma bits (*dirty bit*), kurš tiek uzstādīts tām rindām, kuras jāieraksta atmiņā (skat. 7.3. att.).

Tiešais un apgrieztais ieraksts dažādi ietekmē operatīvās atmiņas darbu. Tiešā ieraksta gadījumā visi izmainītie dati uzreiz attēlojas operatīvajā atmiņā, bet apgrieztā ieraksta gadījumā dati operatīvajā atmiņā ierakstās ar aizturi, kas var sasniegt dažus simtus vai pat tūkstošus kopnes ciklu. Vairākumā kešdarbes apakšsistēmās izmanto operāciju, kurā ierakstās visas kā „dirty” apzīmētās rindas (*cache flush operation*). Jāatzīmē, ka procesoram vajag bloķēt visas pārējās operācijas ieraksta laikā, lai nodrošinātu no kešatmiņas pārrakstāmo datu veselumu.

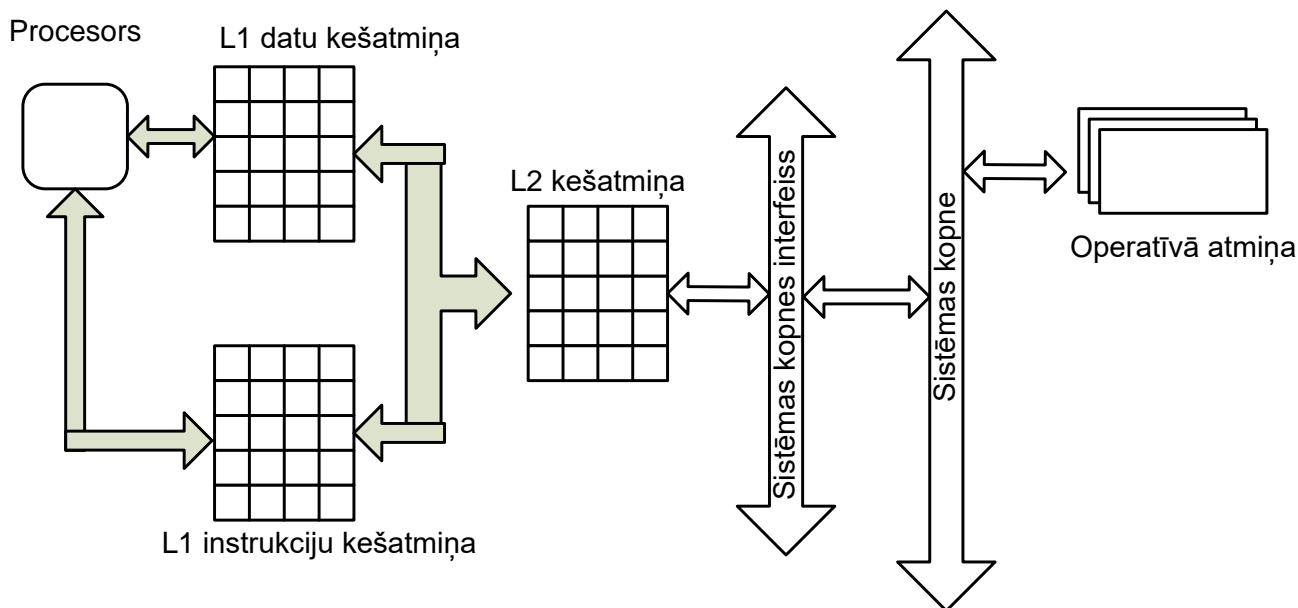
Ja, piemēram, vissliktākajā gadījumā visas kešatmiņas rindas tiek apzīmētas kā „dirty” un ja pieņem kešatmiņas apjomu 256 Kb, tad visas kešatmiņas ierakstam būtu nepieciešami 256 K / kopnes_platums atmiņas cikli.



7.3. att. Apgrieztā ieraksta kešatmiņas funkcionēšana

Mūsdienu datorsistēmu arhitektūrās vienas kešatmiņas vairs nepietiek, tāpēc izmanto 2 vai pat 3 kešatmiņas, kas izvietotas procesorā, turklāt ražīguma paaugstināšanai komandu un datu plūsmas tiek nodalītas tā, lai tās apstrādātu ar dažādu kešatmiņu palīdzību. *Intel Pentium 4* procesoru kešdarbes apakšsistēma parādīta 7.4. attēlā.

Šajos procesoru modeļos izmanto vispārējas nozīmes L2 kešatmiņu, kurā var atrasties gan dati, gan komandas. No kopējās plūsmas izdala datus (apstrādā L1 datu kešatmiņa) un instrukcijas, kas tiek ievietotas L1 instrukciju kešatmiņā.



7.4. att. Kešdarbes apakšsistēma

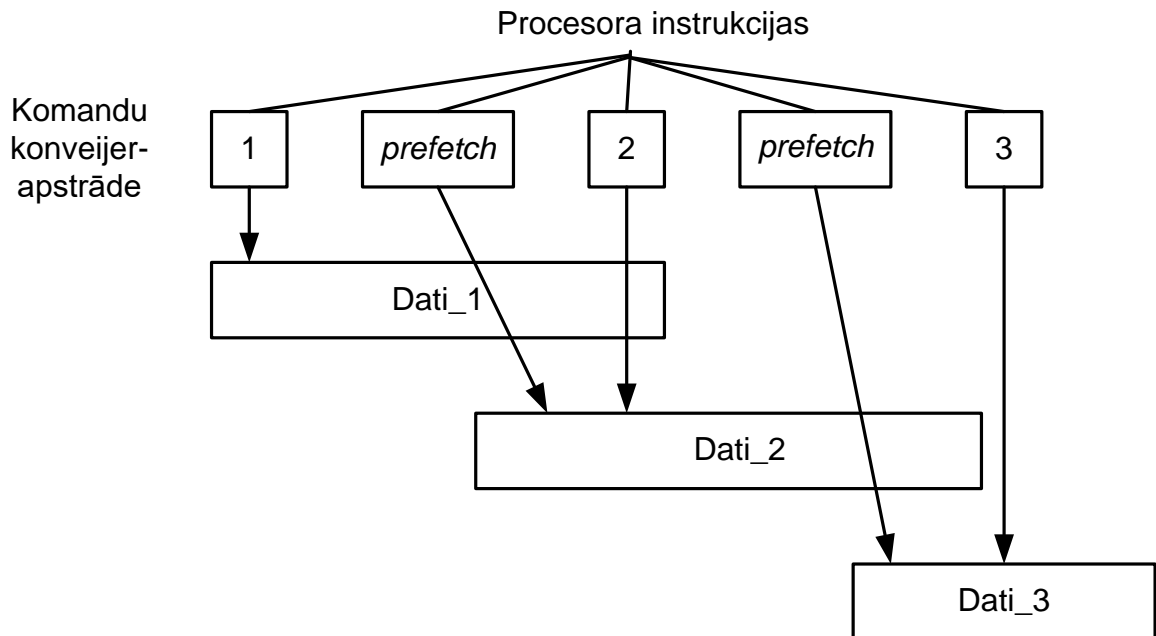
Daļēji sistēmas kešdarbes būtība tika aprakstīta iepriekšējās nodaļās, tagad uzmanība tiks pievērsta jautājumam par datu apmaiņas optimizāciju darbā ar atmiņu un kešatmiņu.



Programmētājam pieejami vairāki paņēmieni lietojumprogrammu veikspējas palielināšanā, efektīvi izmantojot operācijas ar kešatmiņu un atmiņu. Kā svarīgākos var atzīmēt:

- datu pirmsienes kešatmiņā paņēmieni (*prefetch*). Šim nolūkam izmanto SSE paplašinājuma instrukciju grupu – tā saucamās *prefetch*-instrukcijas, kuras iekļauj programmas atsevišķās vietās, lai varētu ielādēt kešatmiņā kā minimums 128 baitus (*Pentium 4*) pirms tam, kad tie būs nepieciešami programmai. Tas ļauj samazināt aizturi, ko rada datu ielāde kešatmiņā, jo instrukcijas izpildes gadījumā tie jau tur būs. Datu pirms ieneses kešatmiņā paņēmieni var tikt izmantoti gadījumos, ja zināms pieejas algoritms datiem, kas ļauj spriest par to, kādi dati tiks drīzumā ielādēti;
- datu kešdarbes vadības mehānismu izmantošana (*cacheability control*). Kešatmiņas vadības procesora instrukcijas ļauj programmētājam noteikt kešdarbes vadības stratēģiju, minimizējot kešatmiņas izmantošanu.

Datu pirms ienesi kešatmiņā realizē sekojošas instrukcijas: *prefetchnta*, *prefetcht0*, *prefetcht1*, *prefetcht2* (instrukciju nozīme tika parādīta 2.4. apakšnodaļā). Šādu instrukciju pielietošana ir efektīva gadījumos, kad dati vēl nav ielādēti kešatmiņā. Pirmsienes mehānisma darbības būtība parādīta 7.5. attēlā.



7.5. att. Datu pirmsieneses kešatmiņā darbības shēma

Pieņem, ka procesora instrukcija 1 apstrādā datus_1. Instrukcija *prefetch*, kas seko pēc instrukcijas_1, negaidot instrukcijas_2 sākumu, veic datu_2 pirmsienesi kešatmiņā (šie dati būs nepieciešami instrukcijai_2). Tādējādi, sākoties instrukcijai_2, tās darbam nepieciešamie dati jau atrodas kešatmiņā.

Lai panāktu programmas koda maksimāli efektīvu darbu, svarīgi ir noteikt instrukcijas *prefetch* atrašanās vietu un nobīdi attiecībā pret datiem, kas jāievieto kešatmiņā. Ja nobīde būs neliela, tad instrukcijas *prefetch* pielietošana būs bezjēdzīga, jo nākamo datu porcija jau var atrasties kešatmiņā vēl pirms instrukcijas izpildes. Ja nobīde būs pārāk liela, ar instrukcijas *prefetch* palīdzību izņemtie dati var tikt pārrakstīti ar citiem datiem līdz tam, kad tie būs nepieciešami. Pareizās nobīdes izvēle balstās uz empīriskām sakarībām. Piemēram, cikliskajos skaitļojumos nobīdi pirms ieneses operācijai var aptuveni izskaitļot, ņemot vērā iterāciju skaitu.



7.1. piemērs. Instrukcijas *prefetch* izmantošana

```

top_loop:
prefetchnta [edx + esi + 128*3]
prefetchnta [edx*4 + esi + 128*3]
.....
movaps xmm1, [edx + esi]
movaps xmm2, [edx*4 + esi]
movaps xmm3, [edx + esi + 16]
movaps xmm4, [edx*4 + esi + 16]
.....
.....
add esi, 128
cmp esi, ecx
jl top_loop

```

Šeit instrukcija *prefetch* ievietota ciklā. Pirms ieneses nobīdes adreses vērtība ir 3, turklāt reģistrs ESI satur rindas adresi, EDX satur datu adresi, bet reģistri XMM1-XMM4 tiek izmantoti skaitļošanai. Dotajā piemērā vienā iterācijā izmantotas divas atsevišķas kešatmiņas rindas.

Panākt augstāku ražīgumu darbā ar atmiņu var, izmantojot datu kešdarbes vadības mehānismu. Tāda mehānisma izmantošanas ideja balstās uz pieņēmumu, ka daudzos gadījumos

apstrādājamiem datiem tiek izmantoti tikai vienu reizi un turpmāk tie netiks izmantoti. Tādus datus *Intel* terminoloģijā sauc par neizmaināmiem (*non-tempora*).

Neizmaināmos datus ir jēga apstrādāt ar minimālu kešatmiņas izmantošanu. Šādiem nolūkiem SSE paplašinājumā ir iekļauta instrukciju grupa, kas minimizē kešatmiņas izmantošanu – *maskmovq*, *maskmovdqu*, *movntq*, *movntdq* un *movntps*, kuras var izmantot, piemēram, datu ātrai kopēšanai vai pārvietošanai.



7.2. piemērs. Instrukcijas *maskmovdqu* izmantošana

```
// 7-2 maskmovdqu izmantoshana
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{
    __declspec (align(16)) char src[]="0123456789ABCDEF";
    __declspec (align(16)) unsigned char msk[]={ 0x80, 0x0, 0x0, 0x0, 0x0,
                                                0x80, 0x0, 0x0, 0x80, 0x0,
                                                0x0, 0x0, 0x0, 0x80, 0x0, 0x80};
    __declspec (align(16)) char dst[]="";
    __asm
    {
        lea EDI, dword ptr dst
        lea ESI, dword ptr src
        movdqu XMM0, [ESI]
        lea EBX, dword ptr msk
        movdqu Xmm1, [EBX]
        maskmovdqu XMM0, XMM1
    }
    printf("%s\n", dst);
    getchar();
    return 0;
}
```

Rezultāts

0 5 8 D F.

Komentāri

Instrukcija *maskmovdqu* pārsūta datus no reģistra XMM datu apgabālā, uz kuru norāda reģistrs EDI. Otrs XMM reģistrs satur masku no 16 baitiem. Katra baita vecākais bits nosaka, vai baits no pirmā XMM reģistra tiks pārsūtīts, vai nē. Ja vecākais bits ir 1, baits tiek kopēts, ja 0 – baits netiek kopēts. Šīs instrukcijas gadījumā procesors neielādē datus kešatmiņā, bet ieraksta datus tieši atmiņā.

Programmā ir izmantoti trīs datu masīvi:

- simbolu masīvs *src*;
- masīvs *dst*, kurā tiek iekopēti dati;
- masīvs *msk*, kas satur maskas elementus (vērtība 0x80 nozīmē to, ka masīva *src* elements tiks pārkopēts uz attiecīgo pozīciju masīvā *dst*, bet vērtība 0 nozīmē to, ka masīva *dst* elements paliks bez izmaiņām).



7.3. piemērs. Instrukcijas *movntdq* izmantošana

```
// 7-3 Instrukcijas movntdq pielietošana
#include "stdafx.h"
#include <stdio.h>

int _tmain(int argc, _TCHAR* argv[])
{
    __declspec (align(16)) double src[2]={154.38, 729.15};
    __declspec (align(16)) double dst[2];
    __asm
    {
```

```

lea ESI, dword ptr src
lea EDI, dword ptr dst
movapd XMM0, [ESI]
sqrtpd XMM0, XMM0
movntdq [EDI], XMM0
}
printf("dst[0]= %6.3f, dst[1]= %6.3f\n", dst[0], dst[1]);
getchar();
return 0;
}

```

Rezultāts

12,425 27,003

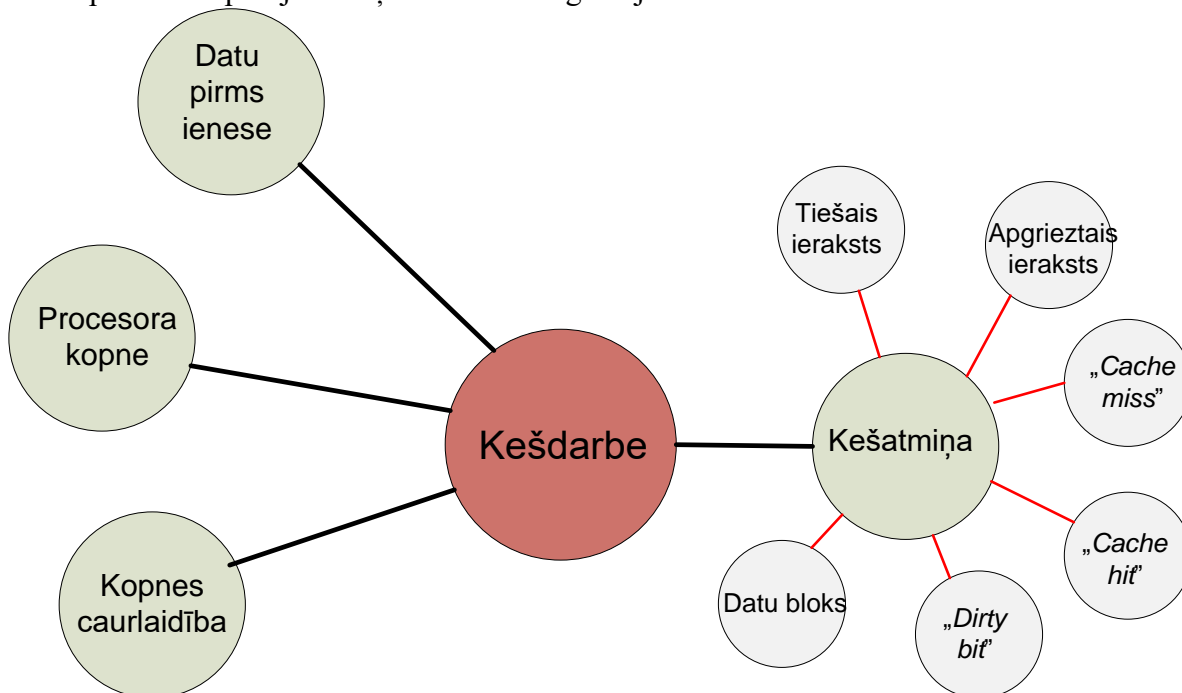
Komentāri

Masīvā *dst* tiek ierakstītas izskaitļotās kvadrātsaknes vērtības no skaitļiem, kas atrodas masīvā *src*. Instrukcija *movapd* veic datu izlīdzināšanu 16 baitu robežās.

NODAĻAS KOPSAVILKUMS

- Kešdarbe – tas ir aparatūras daļas un programmu risinājums, kas tiek izmantots datu apmaiņas sinhronizācijai un paātrināšanai starp dažādām datorsistēmas ierīcēm, kam ir dažāda ātrdarbība.
- Kešdarbes apakšsistēmas izstrādē rodas divas problēmas:
 - kešatmiņas optimālās caurlaidības spējas izvēle;
 - kešatmiņas optimālā apjoma izvēle.
- *Intel Pentium 4* procesoros L2 kešatmiņas dati tiek pārraidīti pa iekšējo kopni, kuras izmērs ir 256 biti.
- Pastāv divas stratēģijas datu ierakstīšanai no kešatmiņas operatīvajā atmiņā: tiešais ieraksts un apgrieztais ieraksts.
- Programmētājam pieejami vairāki paņēmieni lietojumprogrammu veikspējas palielināšanā, efektīvi izmantojot operācijas ar kešatmiņu un atmiņu.

7.6.attēlā parādīts septītajā nodaļā minēto svarīgāko jēdzienu koks.



7.6. att. Nodaļas svarīgāko jēdzienu koks




Uzdevumi un jautājumi patstāvīgajam darbam

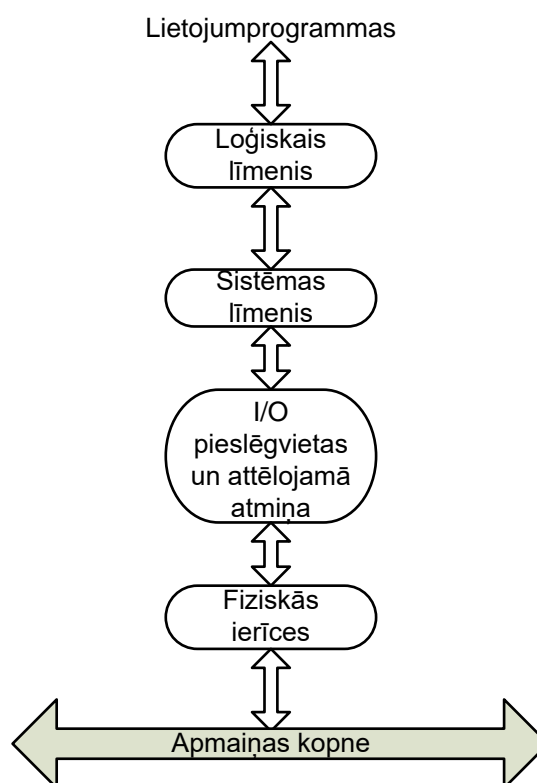
1. Cik lielā mērā kešdarbe ietekmē datorsistēmas veiktspēju?
2. Vai būtu racionāli ieviest L3, L4, L5 utt. kešatmiņas?
3. Vai tehnoloģiski ir iespējams izstrādāt kešatmiņu, kuras apjoms būtu, piemēram, 30% no operatīvās atmiņas apjoma?
4. Kāds datu ierakstīšanas veids no kešatmiņas operatīvajā atmiņā ir optimālāks – tiešais ieraksts vai apgrieztais ieraksts?
5. Kāpēc L1 kešatmiņā atsevišķi nodalīta datu kešatmiņa no instrukciju kešatmiņas?
6. Datu pārraides ātrums starp centrālo procesoru un atmiņu ir lielāks nekā datu pārraides ātrums starp procesoru un ievadizvades ierīcēm. Kādā veidā šī neatbilstība izraisa datora veiktspējas samazināšanu? Kā to pārvarēt?
7. Pieņemt, ka datorā ir pirmā līmeņa kešatmiņa ar pieejas laiku 6 ns un otrā līmeņa kešatmiņa ar pieejas laiku 11 ns. Pieejas operatīvajai atmiņai laiks ir 60 ns. Ja 15% no pieejas atmiņai veic pirmā līmeņa kešatmiņa, bet 55% - otrā līmeņa kešatmiņa, kāds ir vidējais pieejas laiks?

8. DATORSISTĒMAS IEVADIZVADES APAKŠSISTĒMA

Vienkāršots ievadizvades modelis kopējā datorsistēmas kontekstā jau tika parādīts 2. nodaļā. Šī nodaļa veltīta detalizētākam ievadizvades apakšsistēmas funkciju aprakstam. Galvenā datu ievadizvades atšķirība no operācijām ar atmiņu ir tāda, ka iekārtu mijiedarbība ar procesoru (aparātūras un instrukciju līmenī) un operētājsistēmu tiek realizēta pēc citiem principiem nekā datu apmaiņa ar atmiņu.


8.1. Ievadizvades pieslēgvietas

 Termins „ievadizvades” apakšsistēma ir ļoti nosacīts un ar to saprot gan datora aparatūru (dažādi adapteri, disku iekārtas utt.), gan programmu saskarni, kas ļauj mijiedarboties ar to [3,4,5,9,23]. Lai labāk izprastu šo mijiedarbību, tiks analizēta 8.1. attēlā parādītā shēma.



8.1. att. Datorsistēmas ievadizvades hierarhija

Zemākajā līmenī ierīces mijiedarbojas ar pārējo aparatūras daļu ar vienas vai vairāku apmaiņas kopņu palīdzību. Dažādām aparatūras platformām kopnes saskarne var atšķirties, kaut gan atsevišķi standarti (piemēram, PCI) ir no platformām neatkarīgi. Ierīču mijiedarbība šajā līmenī notiek ar signālu līniju palīdzību, kuru raksturlielumiem jāatbilst izmantojamās kopnes standartiem. Turklāt fiziskās ierīces elektroniskajai saskarnei jānodrošina visu nepieciešamo apmaiņas signālu ģenerāciju.

 Datu apmaiņa pa kopni tiek iniciēta, vērstoties pie fiziskās ierīces programmu resursiem, par kuriem kalpo I/O pieslēgvietas un attēlojamā atmiņa. Šajā līmenī visas operācijas ar ierīcēm pieejamas ar procesora instrukciju palīdzību. No elektronisko shēmu viedokļa I/O pieslēgvietas realizētas reģistru veidā, katrai no tām ir adrese un apjoms. Pieeja reģistriem tiek realizēta ar nolasišanas/ierakstīšanas operācijām, izmantojot procesora instrukcija *in* un *out* un to modifikācijām. Ja ierīce strādā ar reģistriem, kas tiek attēloti uz atmiņu, tad pieeja šiem reģistriem notiek ar instrukcijas *mov* palīdzību.

Nākamos augstākos mijiedarbības līmeņus ar ierīcēm izmanto OS (sistēmas līmenis) un lietojumprogrammas.

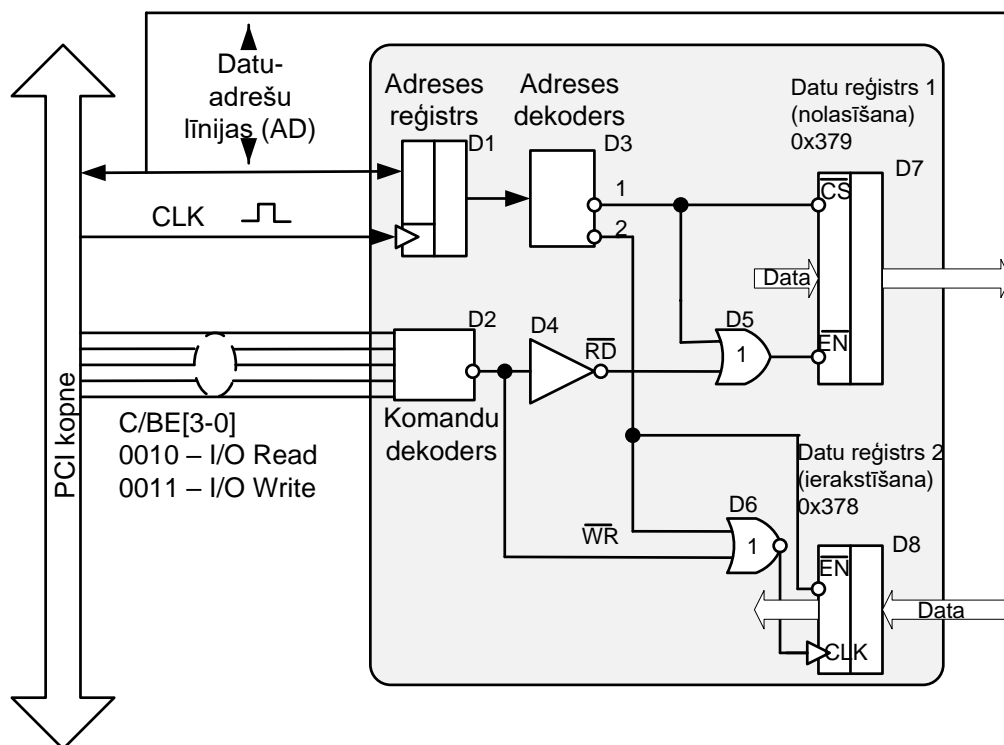
⚠ Jāatzīmē, ka OS visas I/O ierīces, kā arī visas datnes un mapes traktē kā datņu sistēmas objektus. Tas nozīmē, ka pieeja šiem objektiem un operācijas ar tiem notiek pēc noteiktiem unificētiem likumiem neatkarīgi no objekta dabas (ierīce, datne, kanāls utt.). Tāds modelis ļauj sistēmas procesiem un lietotāja pielietojumiem izmantot darbam ar datņu sistēmas objektiem WinAPI saskarnes standarta funkcijas.

Procesors jebkuru I/O ierīci traktē kā I/O pieslēgvietu kopumu [4,7]. Atkarībā no ierīces konfigurācijas tā var saturēt pieslēgvietas ierakstīšanai, nolasīšanai vai ierakstīšanai un nolasīšanai. Ja pieslēgvietas paredzētas tikai nolasīšanai vai ierakstīšanai, tās sauc par vienvirzienu. Ja tās pieļauj gan nolasīšanas, gan ierakstīšanas operācijas, tad tās ir divvirzienu pieslēgvietas, turklāt dati, ar kuriem operē pieslēgvietā, var būt dažādu apjomu: 1 baits, 2 baiti (vārds) un 4 baiti (dubultvārds).

i Pieslēgvietā ir uzskatāma par reģistriem, pieeja kuriem realizēta pašās ierīcēs, dekodējot adresu līnijas un vadības signālus kopējā kopnē. Tādējādi tikai ierīces shēmas atbild par pareizu pieslēgvietas adreses atpazīšanu un apmaiņas operācijām ar tām.

Procesors iniciē apmaiņu ar I/O pieslēgvietu, nosakot sistēmas kopnē pieslēgvietas adresi, operācijas tipu, bet ierīce atpazīst adresi, operācijas tipu un atbild par korektu apmaiņas operācijas izpildi. Jāatzīmē, ka sistēmas kopnes signālus (tātad procesora signālus) tās kopnes controlleris, pie kuras pieslēgta ierīce, pārveido dotās kopnes standartā noteiktajos signālos.

Neskatoties uz procesora un ierīču mijiedarbības šķietamo sarežģītību, I/O pieslēgvietu darbības loģika ir diezgan vienkārša (skat. 8.2. att.).



8.2. att. Ievadizvades ierīces demonstrācijas shēma

Attēlā parādīta vienkāršas I/O ierīces shēma ar divām pieslēgvietām, kas uzdotas shēmā kā *datu reģistrs 1* un *datu reģistrs 2* ar adresēm 0x379 un 0x378. Ierīce pieslēgta PCI kopnei, turklāt no *datu reģistra 1* programma var tikai nolasīt datus, bet *datu reģistrā 2* var ierakstīt datus.

Šī ierīce no elektronisko shēmu realizācijas viedokļa ir vienkāršota un satur tikai adreses dekodēšanas shēmu un nolasīšanas/ierakstīšanas loģiku. Materiāla izklāsta vienkāršības nolūkā nav parādīta shēmas daļa, kas apstiprina datu apmaiņas iniciatora gatavību. Shēmā parādīti tikai tie signāli, kas tiks izmantoti dotajā ierīcē (PCI kopnes darbības principu un signālu skaidrojums ir dots 2. nodaļā):

- takts frekvences impulsi CLK;
- kopnes adreses/dati multipleksētu signālu līnijas (PCI specifikācijā tie apzīmēti kā AD[0:31]). Jāatgādina, ka adreses noteikšanas ciklā šajā līnijā uzdota 32 bitu adrese, bet I/O ierīces adresācijai izmanto tikai 16 bitus (šī iemesla dēļ IA-32 arhitektūrā pieejamo portu skaits ir 64K). Nolasīšanas/ierakstīšanas ciklā šajās līnijās ir dati (tie var būt 8,16 vai 32 bitu dati);
- četras līnijas, kas iestata komandas/baita numuru C/BE[3-0] kodu. Šī hipotētiskā ierīce var izpildīt divas komandas: datu nolasīšanu no ierīces (komandas kods 0x0010) un datu ierakstīšanu ierīcē (kods 0x0011). Vienkāršības labad pieņem, ka abi ierīču reģistri (*datu reģistrs 1* un *datu reģistrs 2*) ir 8 bitu reģistri, tāpēc baita izvēles signāli C/BE3 – C/BE0 netiks izmantoti.

Ieejas pieslēgvietas nolasīšanai var izmantot instrukciju *in*. Piemēram, procesors var nolasīt pieslēgvietas saturu šādi :

```
mov DX, 0x379
in AL, DX
```

Pirmā komanda ievieto DX reģistrā pieslēgvietas adresi, otrā nolasa viena baita datus procesora reģistrā AL.



Datu nolasīšana notiek šādi (sk. 8.2. att.):

- procesors adrešu līnijās iestata adresi 0x379;
- nākošajā impulsa CLK taktī adrese tiek iegaumēta adreses reģistrā D1 un tiek dešifrēta dekoderā D3. Ja adrese ir atpazīta, tad dekodera izejā 1 tiek iestatīts zems signāla līmenis, kas atļauj datu reģistra D7 darbu (zems līmenis ieejā CS (*Crystal Select*));
- Pēc iniciatora apstiprinājuma par gatavību pieņemt datus un datu avota apstiprinājuma tos saņemt (shēmā šie signāli nav parādīti) procesors nolasa datus D7 izejā. Jāpievērš uzmanība nolasīšanas signāla veidošanai – tā ir adrešu dekodera un komandu dekodera signālu kombinācija. Tā rezultātā mikroshēmas D5 izejā tiek iestatīts zems signāla līmenis EN ieejai, kas ļauj izvadīt datus uz kopni (procesoram tie ir ieejas dati). Procesors nolasa datus un ievieto 8 bitu vērtību reģistrā AL.

Pēc datu nolasīšanas cikla beigām attiecīgie signāli tiek dzēsti. Ierakstīšanas operācija reģistrā ar adresi 0x378 ir līdžīga.

Apskatītā shēma ir tikai viens no daudziem ierīces saskarnes uzbūves variantiem. Jāņem vērā, ka datu apmaiņā ar I/O pieslēgvietām procesors nemaskē paritātes kļūdas, kas var kļūt par potenciālu kļūdu avotu.

Visā I/O adrešu telpā (0xFFFF) daļu adrešu rezervē sistēma, un tās nedrīkst izmantot programmas – tās ir adreses no 0xF8 līdz 0xFF.

Ja agrākajās OS (*Windows 98/ME*) lietojumprogrammas varēja pa tiešo vērsties pie I/O pieslēgvietām, izmantojot procesora instrukcijas *in* un *out*, tad vēlākajās OS lietotāja programmas kods nevar izmantot sistēmas adrešu telpu (ieskaitot I/O pieslēgvietu adreses), tāpēc operācijas ar I/O ierīcēm iespējamas tikai ar draiveru palīdzību, bet tiešā piekļūšana pieslēgvietām daudzos gadījumos nav iespējama pat ar draiveru palīdzību. Tas saistīts ar to, ka I/O ierīču draiveri strādā OS aizsargātajā režīmā, kas izslēdz tādu piekļūšanu.

Kad procesors strādā aizsargātajā režīmā, pieejas pieslēgvietām aizsardzību nodrošina šāds mehānisms - tiek iestatīts I/O privilēģiju līmenis lauka IOPL (*I/O Privilege level*) divpadsmitajā un trīspadsmitajā bitā procesora karodziņu reģistrā EFLAGS.

Lauks IOPL karodziņu reģistrā ļauj kontrolēt pieeju adrešu telpai, ierobežojot atsevišķas procesora instrukcijas. Šis aizsardzības mehānisms dod iespēju OS iestatīt privilēģiju līmeni, kas nepieciešams I/O operāciju izpildei. Vispārīgā gadījumā piekļuve I/O adrešu telpai ir ierobežota ar 0 un 1 līmeni – I/O operācijas var veikt tikai kodola draiveri un ierīču draiveri, kamēr lietojumprogrammas ar mazāku prioritāti to nespēj izdarīt. Mazāk privilēģētām programmām pieejamas I/O metodes ar WinAPI saskarnes funkciju izsaukšanu.

Instrukcijas *in*, *ins*, *out*, *outs*, *cli* un *sti* tiks izpildītas tikai tad, ja izpildāmās programmas privilēģiju līmenis CLP (*Current Privilege Level*) ir mazāks vai vienāds ar to, kurš iestādīts IOPL. Jebkurš mazāk privilēģētas programmas mēģinājums izpildīt I/O operāciju, izsauks kopējās aizsardzības stāvokli GPE (*General-Protection Exception*). Tā kā katra izpildāmā programma saņem savu karodziņa reģistra kopiju, tad tai iestata savu IOPL.

Procesora instrukcijas, kas tiek izmantotas I/O operācijās, var iedalīt divās grupās:

- instrukcijas, kas pārsūta baitu, vārdu vai dubultvārdu;
- instrukcijas, kas pārsūta baitu rindu, vārdu rindu vai dubultvārdu rindu.

Instrukcijas *in* (datu ievads no pieslēgvietas) un *out* (datu izvads pieslēgvietā) pārsūta datus starp pieslēgvietu un reģistru EAX (32 bitu operācija), reģistru AX (16 bitu operācija) vai AL (operācija ar vienu baitu). Pieslēgvietas adrese var tikt uzdota tieši komandā vai ar reģistra DX palīdzību.

Piemēram:

```
in AL, 0x378
mov DX, 0x379
out DX, AL
```

Rindu instrukcijas *ins* un *outs* pārsūta datus starp atmiņu un I/O pieslēgvietām. Turklāt pieslēgvietas adrese tiek ierakstīta reģistrā DX, bet atmiņas apgabalu nosaka reģistru pāris DS:ESI (ievada operācijām) vai ES:EDI (izvada operācijām).

8.2. Ievadizvades ierīces *Windows* vidē

Jebkurā gadījumā programma, kas operē ar ierīcēm, tieši vai netieši izmanto I/O pieslēgvietu mehānismu operāciju veikšanai, kaut arī šīs mijiedarbības detaļas paliek slēptas lietotājam [23].

Tieša datu apmaiņa ar ierīcēm caur I/O pieslēgvietām teorētiski ir iespējama no jebkuras lietojumprogrammas (neskatoties uz aizsardzības iespējām), vēl vairāk – daudzos gadījumos tas tiek pielietots gan lietojumprogrammās, gan ierīču draiveros, lai panāktu lielāku ātrdarbību. Tomēr daudzos gadījumos var iztikt bez tiešas I/O pieslēgvietu programmēšanas, bet izmantot OS iespējas.

Lietojumprogrammas mijiedarbību ar ierīcēm OS var ilustrēt ar 8.3. attēlā redzamo shēmu.

Programmas kods, kas izpildās OS, var funkcionēt vienā no diviem režīmiem:

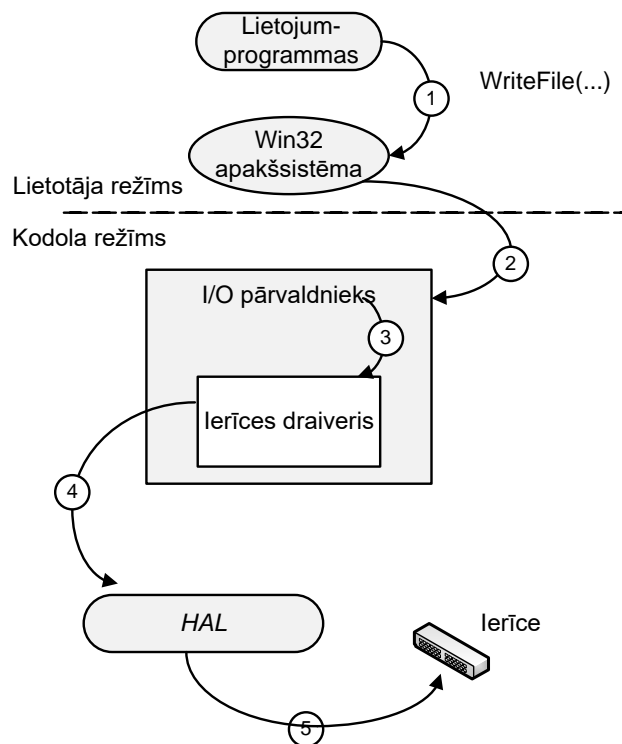
- lietotāja (*user mode*) vai
- kodola režīmā (*kernel mode*).

Lielākā daļa no OS komponentiem (sistēmas dienesti, kodola draiveri, ierīču draiveri utt.) funkcionē kodola režīmā. Kodola režīma programmas kods dod praktiski neierobežotu pieeju sistēmas aparatūras resursiem, ieskaitot I/O ierīces.

Apskatītajā piemērā ieejas pieslēgvietas ar adresi 0x379 nolasīšana iespējama tikai no programmas, kas strādā kodola režīmā.



Lietojumprogramma darbojas tikai lietotāja režīmā un mijiedarbojas ar OS ar pielietojamās programmēšanas WinAPI saskarnes palīdzību, kas realizēts Win32 apakšsistēmas veidā (sk. 8.3. att.).



8.3. att. Lietojumprogrammas un ierīces mijiedarbība

Piemēram, ja lietojumprogramma mēģina nolasīt datnes saturu, tai vajag izsaukt API funkciju *ReadFile*, pēc tam bibliotēkas *kernel32.dll* programmas kods vēršas pie sistēmas dienestiem, kas izsauc citas funkcijas – izpildāmās kodola režīmā. Dotajā gadījumā tiek izsaukta funkcija *NtReadFile*, kas arī veic visu tālāko darbu. Tātad lietojumprogramma var vispār iztikt bez kodola funkciju izsaukšanas, ja izpildāmās operācijas to neprasa.

8.3. attēlā dotā shēma darbojas šādā veidā [9]:

- Pieņem, ka lietojumprogrammas kods vēršas pie ierīces, izsaucot vienu no Win32 apakšsistēmas API funkcijām (atvēršanu, aizvēršanu vai nolasīšanu un ierakstīšanu). Šajā piemērā nepieciešams veikt datu ierakstīšanu ierīcē ar funkcijas *WriteFile* palīdzību ①
- Win32 apakšsistēma kalpo par saiti starp lietojumprogrammu un kodolu. Tā kā pieprasīta ierakstīšanas operācija ierīcē, kas prasa pieeju kodola funkcijām, Win32 vēršas pie sistēmas komponenta, ko sauc par I/O pārvaldnieku (*I/O manager*) ②. Vienkāršības labad šis komponents attēlā parādīts viena bloka veidā, kaut funkcionāli tas sastāv no sistēmas servisu grupas. Tāpat arī nav parādīta vēl virkne komponentu, kas darbojas ierakstīšanas ierīcē laikā - objektu pārvaldnieks (*Object manager*), drošības apakšsistēma (*Security Subsystem*) u.c.
- Tālāk I/O pārvaldnieks veido tā saucamo pieprasījuma paketi (*I/O Request Packet*), kurā „nošifrēta” ierakstīšanas operācija, un nodod šo paketi ierīces draiverim (*Device driver*) – speciālai programmai, kas apkalpo operācijas ar šo ierīci ③.
- Ierīces draiveris analizē pieprasījuma paketi, pēc tam izsauc aparatūras abstrakcijas līmeņa funkcijas *HAL* (*Hardware Abstract Layer*), kas atrodas bibliotēkā *hal.dll*, pieprasījuma apstrādei ④, ⑤. Turpmāk ar *hal.dll* funkciju palīdzību tiek veiktas darbības ar ierīces aparatūru, pēc kā operācijas statuss (bet pie nolasīšanas operācijas vēl arī dati) pa to pašu ķēdīti atgriežas lietojumprogrammā.

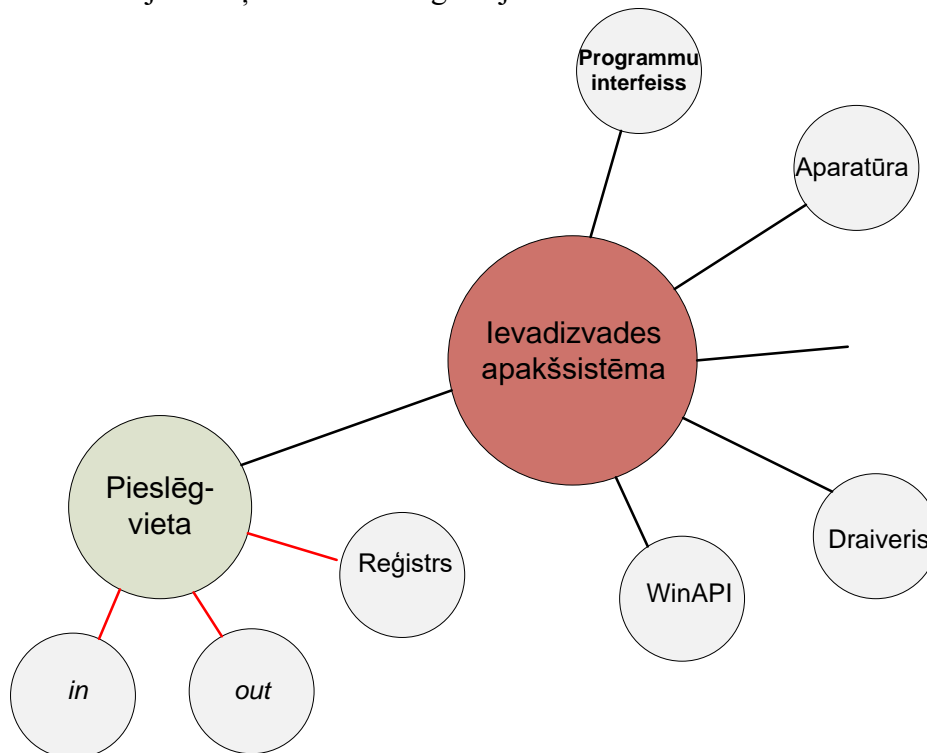
Aparatūras abstrakciju līmenis – pēdējais posms, līdz kuram visas operācijas sistēmā izpildās loģiskā līmenī, t.i., visas vēršanās pie aparatūras līdzekļiem izpildās neatkarīgi no

platformas, kurā OS strādā (x86, Alpha, SPARC utt.). HAL līmenī ar bibliotēkas *hal.dll* funkciju palīdzību tiek veiktas tikai no platformas atkarīgas operācijas. Ko tas nozīmē? Instrukcijas *in* un *out* strādā tikai ar Intel procesoriem. Tas pats attiecas uz dinamisko bibliotēku *hal.dll*, tajā noteiktas funkcijas, kas ļauj strādāt tikai ar platformas x86 aparatūru.

NODAĻAS KOPSAVILKUMS

- Termins „ievadizvades” apakšsistēma ir ļoti nosacīts un ar to saprot gan datora aparatūru, gan programmu saskarni, kas ļauj mijiedarboties ar to.
- Procesors jebkuru I/O ierīci traktē kā I/O pieslēgvietu kopumu.
- Pieslēgvietā ir uzskatāma par reģistriem, piekļuve kuriem realizēta pašās ierīcēs - dekodējot adresu līnijas un vadības signālus kopējā kopnē. Tikai ierīces shēmas atbild par pareizu pieslēgvietas adreses atpazīšanu un apmaiņas operācijām ar tām.
- Agrākajās OS (*Windows 98/ME*) lietojumprogrammas varēja pa tiešo vērsties pie I/O pieslēgvietām, izmantojot procesora instrukcijas *in* un *out*.
- Jaunākajās OS operācijas ar I/O ierīcēm iespējamas tikai ar draiveru palīdzību, bet tiešā pieeja pieslēgvietām daudzos gadījumos nav iespējama pat caur tiem.
- Lietojumprogrammas strādā tikai lietotāja režīmā un mijiedarbojas ar OS ar pielietojamās programmēšanas WinAPI saskarnes palīdzību.

8.4. attēlā parādīts astotajā nodaļā minēto svarīgāko jēdzienu koks.



8.4. att. Nodaļā minēto svarīgāko jēdzienu koks



Uzdevumi un jautājumi patstāvīgajam darbam

1. Kādas datora ierīces (bez PCI) ir no platformām neatkarīgas?
2. Vai procesors veic ierīču pieslēgvietu atpazīšanu?
3. Kas kalpo par datu apmaiņas iniciatoru?
4. Kāpēc datu apmaiņā ar I/O pieslēgvietām procesors nemaskē paritātes kļūdas?
5. Vai lietojumprogramma var strādāt kodola režīmā?
6. Kuras OS ļauj piekļūt pie pieslēgvietām pa tiešo?
7. Paskaidrojiet ierakstīšanas operācijas reģistrā ar adresi 0x378 shēmu 8.2. attēlā.

9. PARALĒLĀ PIESLĒGVIETA UN TĀS PROGRAMMĒŠANA

Nodaļā apskatītā paralēlā pieslēgvietā [3,5,9,11] ir viena no senākajām aparatūras un programmu saskarnēm, kas tika izstrādāta datu apmaiņai ar drukas ierīcēm. Mūsdienās paralēlā pieslēgvietā papildus savu tiešo uzdevumu veikšanai tiek izmantota dažādos projektos, kas saistīti ar datu apstrādi un dažādu ārējo ierīču pieslēgšanu pie datora.

Pieslēgvietā dod iespēju saņemt 9 bitu datus vai sūtīt 12 bitu datus. Aparatūras saskarne satur 4 vadības līnijas, 5 stāvokļa līnijas un 8 datu līnijas. Paralēlās pieslēgvietas izvades pieslēgtas pie 25 kontaktu savienotāja (connector) DB-25, kurš atrodas uz datora aizmugurējā paneļa.

9.1. Paralēlās pieslēgvietas signāli

Par pirmo paralēlās pieslēgvietas standartu kļuva saskarne, kas pazīstama ar nosaukumu „Centronics”. Šī saskarne apraksta signālus, apmaiņas protokolu un ārējo kontaktu izvietojumu.

i No elektronisko shēmu viedokļa Centronics saskarne realizēta kā programmētājam pieejams triju ievadizvades reģistru kopums: datu, stāvokļa un vadības reģistri.

9.1. tabulā dots saskarnes signālu apraksts, DB-25 kontaktu numerācija, kā arī signālu virziens attiecībā pret datora paralēlo pieslēgvietu.

9.1. tabula

Centronics saskarnes specifikācija

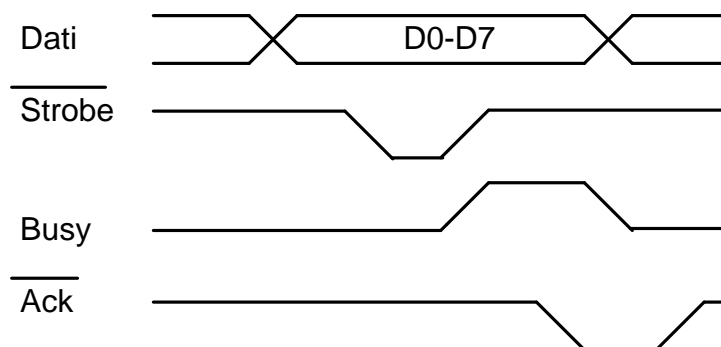
Kontakta numurs	Saskarnes signāls	Signāla apraksts	Reģistra bits	Signāla virziens	Signāla aktīvais līmenis
1	<i>Strobe</i>	Datu gatavības signāls	Vadības-0	Ievade/izvade	Zems, inversija
2	<i>Data0</i>	Datu bits 0	-	Izvade	-
3	<i>Data1</i>	Datu bits 1	-	Izvade	-
4	<i>Data2</i>	Datu bits 2	-	Izvade	-
5	<i>Data3</i>	Datu bits 3	-	Izvade	-
6	<i>Data4</i>	Datu bits 4	-	Izvade	-
7	<i>Data5</i>	Datu bits 5	-	Izvade	-
8	<i>Data6</i>	Datu bits 6	-	Izvade	-
9	<i>Data7</i>	Datu bits 7	-	Izvade	-
10	<i>Ack</i>	Datu saņemšanas apstiprinājums	Stāvokļa -6	Ievade	-
11	<i>Busy</i>	Datu saņemšanas gatavība	Stāvokļa -7	Ievade	Inversija
12	<i>Paper-out</i>	Papīra beigas	Stāvokļa -5	Ievade	-
13	<i>Select</i>	Printeris ieslēgts/izslēgts	Stāvokļa -4	Ievade	-
14	<i>AutoLineFeed</i>	Automātiska rindas pārnese	Vadības-1	Ievade/izvade	Inversija
15	<i>Error/Fault</i>	Ierīces kļūda	Stāvokļa -3	Ievade	-
16	<i>Initialize</i>	Ierīces inicializācija	Vadības-2	Ievade/izvade	-
17	<i>Select-Printer Select-In</i>	Printera izvēle	Vadības-3	Ievade/izvade	Inversija
18-25	<i>Gnd</i>	Kopīgie	-	-	-

Doto signālu specifikācija atbilst *Centronics* saskarnei, tā tika standartizēta un ieguva nosaukumu SPP (*Standard Parallel Port*). Jaunākie datu apmaiņas režīmi EPP (*Enhanced Parallel Port*) un ECP (*Extended Capabilities Port*) tika izstrādāti un pieņemti *IEEE1284* standarta ietvaros.

9.1. attēlā parādīta vienkāršota paralēlās pieslēgvietas datu apmaiņas laika diagramma.



Datu apmaiņas procesā starp ierīcēm viena no tām ir apmaiņas iniciators (avots), bet cita – mērķis. Standarta saskarnei SPP par datu apmaiņas iniciatoru parasti kalpo dators (hosts), bet par mērķi – printeris vai cita ierīce.



9.1. att. Datu apmaiņa saskaņā ar SPP

Datu baita nodošana no avota līdz mērķim notiek šādā veidā:

- tiek pārbaudīts signāla *Busy* stāvoklis – ja tas ir 0, tad avots sāk datu pārraidi, šim nolūkam līnijās *D0-D7* iesūtot datus un iestatot zemu signāla *Strobe* līmeni (dati ir gatavībā);
- avots iestata zemu signāla *Strobe* līmeni, norādot uz to, ka dati *D0-D7* ir gatavībā;
- mērķis nolasa baitu no datu kopnes, iestatot signāla *Busy* augstu līmeni. Apstrādes beigās mērķis iestata zemu signāla *ACK* līmeni, apstiprinot apstrādes beigas, un atbrīvo signālu *Busy*. Ja noteiktā laika periodā (parasti 5 mks) netiek iestatīts signāls *ACK*, tad avots uzskata, ka ir notikusi operācijas izpildes laika kļūda mērķa pusē (vairumā pielietojumos šo signālu laika ekonomijas nolūkā ignorē).

Ja pēc kārtējā baita saņemšanas mērķis kaut kādu iemeslu dēļ nav gatavs saņemt nākamo datu porciju, tad netiek noņemts signāls *Busy*.

Agrīnajās datorsistēmās paralēlā pieslēgvietā tika realizēta kā atsevišķa ierīce, tagad pieslēgvietas loģika atrodas datora mātesplatē. Kā jau tika teikts, paralēlās pieslēgvietas bāzes modelis realizēts kā triju reģistru kopums: datu, stāvokļa un vadības reģistri.

Par paralēlās pieslēgvietas bāzes adresi pieņem datu reģistra adresi, attiecībā pret kuru tiek izskaitļotas pārējās adreses. Ja, piemēram, pieslēgvietas bāzes adrese ir *Base*, tad datu reģistram ir adrese *Base*, stāvokļa reģistram – *Base+1*, vadības reģistram – *Base+2*.

Par bāzes reģistriem visbiežāk izmanto reģistrus ar adresēm 3BCh, 378h un 278h. Paralēlās pieslēgvietas draiveris datu apstrādē var izmantot aparatūras pārtraukumu (parasti tas ir IRQ7 vai IRQ9). Pieslēgvietas darba režīmu var mainīt BIOS iestatījumos.



Pārtraukuma pieprasījuma (*Interrupt Request* – IRQ) līnijas ir fiziskas līnijas, pa kurām dažādas ārējās ierīces (piemēram, ievadizvades pieslēgvietas, tastatūra, disku atmiņa, tīkla adaptera plates) var aizsūtīt datora mikroprocesoram pieprasījumu par apkalpošanu. IRQ līnijām ir dažādas prioritātes, kas ļauj procesoram noteikt vissvarīgāko no pieprasījumiem. Datora ierīcēm jāizmanto dažādas IRQ līnijas, lai nebūtu konfliktu.

Bāzes adrese 3BCh tika izmantota agrīno modeļu kombinētajās video kartēs, tagad 3BCh tiek piedāvāta par integrētās paralēlās pieslēgvietas izvēles adreses variantu. OS paralēlajām pieslēgvietām piešķir simboliskus nosaukumus LPTn, kur n – pieslēgvietas numurs.

LPT1 parasti tiek piešķirts pieslēgvietai ar bāzes adresi 378h, bet LPT2 – pieslēgvietai ar bāzes adresi 278h (sk. 9.2. tabulu).

9.2. tabula

Paralēlās pieslēgvietas bāzes adreses

Adrese	Piezīmes
3BCh – 3BEh	Neuztur ECP
378h – 37Fh	Piešķirtas pieslēgvietai LPT1
278h – 27Fh	Piešķirtas pieslēgvietai LPT2

Datorsistēmas ieslēgšanas laikā BIOS pārbauda adresi 3BCh un, konstatējot, ka tā pieder paralēlajai pieslēgvietai, pieslēgvietai piešķir nosaukumu LPT1. Tālāk tiek pārbaudītas adreses 378h un 278h. Ja adreses ir atrastas, tām piešķir atbilstošus nosaukumus LPT2 un LPT3. Mūsdienu mātesplates adresi 3BCh paralēlajām pieslēgvietām vairs neuztur, tāpēc LPT1 bāzes adrese ir 378h, bet LPT2 – 278h.

Tā kā paralēlās pieslēgvietas var strādāt dažādos režīmos, tad vairumā BIOS nodrošina šādu režīmu uzturēšanu:

- *Printer* (var būt nosaukumi *Default* vai *Normal*) – tas ir pamata režīms (SPP). Šajā režīmā paralēlā pieslēgvietā darbojas vienā virzienā, tāpēc vadības reģistra 5. bits netiek pielietots;
- *Standard & Bi-directional* (SPP) – pieslēgvietā darbojas abos virzienos. Vadības reģistra 5. bits ļauj mainīt signālu pārraides virzienu, kā rezultātā var nolasīt bitus no datu līnijām;
- EPP 1.7 un SPP – divu režīmu kombinācija. Pieejami SPP un EPP reģistri. Signāla virzienu var mainīt ar vadības reģistra 5. bita palīdzību. EPP 1.7 ir EPP agrāka versija, tāpēc nav paredzēta operācijas izpildes laika kontrole;
- EPP 1.9 un SPP – līdzīgs iepriekšējam režīmam, tiek izmantota EPP 1.9 versija;
- ECP;
- ECP un EPP 1.7;
- ECP un EPP 1.9.

Piemērā ir demonstrēts asamblera programmas fragments 1 datu baita pārsūtīšanai ar *Centronics* saskarnes palīdzību:



9.1. piemērs

```
.data
dataport      EQU 378h      ; datu reģistrs
statusPort    EQU 379h      ; stāvokļa reģistrs
controlPort   EQU 37Ah      ; vadības reģistrs
DATA          DB    <vērtība>
BUSY          EQU 7         ; stāvokļa reģistra 7. bits
.code
clc
mov  DX, controlPort
or   AL, 1h
out  DX, AL                ; augsts signāla Strobe līmenis
mov  AL, DATA
mov  DX, dataport
out  DX, AL                ; nosaka datus līnijās D0-D7

; signāla BUSY līmeņa pārbaude
mov  DX, statusPort
again:
```

```

in    AL, DX
bt    AX, BUSY
jc    again

; uz BUSY līnijas zems līmenis, iestata signāla Strobe zemu līmeni
mov   DX, controlPort
and   AL, 0FEh
out   DX, AL
wait:
mov   ECX, 10
loop wait

; pēc aiztures iestata signāla Strobe augstu līmeni
or    AL, 1h
out   DX, AL
;

```

Komentāri

Datu segmentā noteiktas visas pieslēgvietas, signālu biti un datu baits DATA.

Sākotnēji iestata signāla *Strobe* (sk. 9.1. att.) līmeni 1, tādējādi uzsākot jaunu datu pārraides ciklu:

```

mov   DX, controlPort
or    AL, 1h
out   DX, AL

```

Tad datu līnijās *D0-D7* (sk. 9.1. att.) iestata pārsūtāmo datu baitu:

```

mov   AL, DATA
mov   DX, dataPort
out   DX, AL

```

Jāgaida, kamēr ierīce paziņos par datu pieņemšanas gatavību – iestatot zemu signāla *Busy* līmeni (sk. 9.1. att.):

```

mov   DX, statusPort
again:
in    AL, DX
bt    AX, BUSY
jc    again

```

Saņemot gatavības signālu no ierīces, iniciators iestata zemu signāla *Strobe* līmeni un, nogaidot ne mazāk par 1 mks, atjauno augstu līmeni:

```

mov   DX, controlPort
and   AL, 0FEh
out   DX, AL
wait:
mov   ECX, 10
loop wait
or    AL, 1h
out   DX, AL

```

Parasti ierīce saņem datus signāla *Strobe* augošā līmenī, pēc kā iniciators var uzsākt jaunu datu pārraides ciklu. Šajā koda fragmentā netiek analizēts signāls *Ack*.

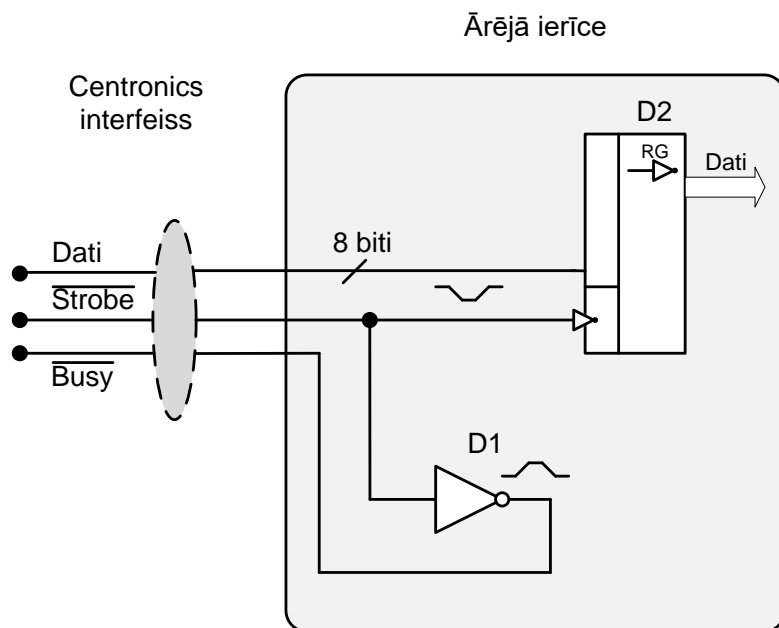
Centronics saskarni var veiksmīgi izmantot dažādu ievadizvades ierīču izstrādāšanā datu apstrādes projektos. Viena no šādām izstrādāšanām parādīta 9.2. attēlā.

Parādītajā shēmā par D2 var izmantot, piemēram, mikroshēmu *74LS374* un D1 – jebkuru inventoru, piemēram *74LS04*.



Jāatzīmē, ka tiešā ievadizvades pieslēgvietu programmēšana ar asamblera palīdzību bija iespējama tikai OS *Windows 98/ME*, jo visi sistēmas resursi pieejami lietojumprogrammu procesiem ar instrukciju *in* un *out* palīdzību.

Lai iegūtu pieeju pie ievadizvades ierīcēm jaunākajās OS, var mēģināt izmantot brīvi izplatāmo draiveri *PortTalk* (www.beyonlogic.org), iekļaujot tajā 9.1. piemērā doto asamblera kodu ar nepieciešamajām izmaiņām (skatīt *PortTalk* aprakstu un dokumentāciju).



9.2. att. Vienkāršots *Centronics* saskarnes realizācijas piemērs

9.2. EPP režīms

EPP protokols tika izstrādāts ar nolūku paplašināt standarta paralēlās pieslēgvietas (SPP) iespējas. Pēc EPP protokola praktiskās realizācijas *Intel* mikroshēmā 82360 tas tika aprakstīts *IEEE 1284* standartā.

EPP protokols nodrošina 4 dažādus datu pārraides ciklus:

- datu ierakstīšanas ciklu (*data write cycle*);
- datu nolasīšanas ciklu (*data read cycle*);
- adreses ierakstīšanas ciklu (*address write cycle*);
- adreses nolasīšanas ciklu (*address read cycle*).

Datu nolasīšanas un ierakstīšanas cikli tiek izmantoti datu pārraidē starp datorsistēmu un ārējām ierīcēm, turklāt EPP protokols nodrošina datu pārraidi abos virzienos.

Adreses nolasīšanas un ierakstīšanas cikli var tikt izmantoti komandas, ierīces adreses vai citas vadības informācijas iestatīšanai. 9.3. tabulā parādīti EPP protokola signāli un tiem atbilstošie protokola SPP signāli.

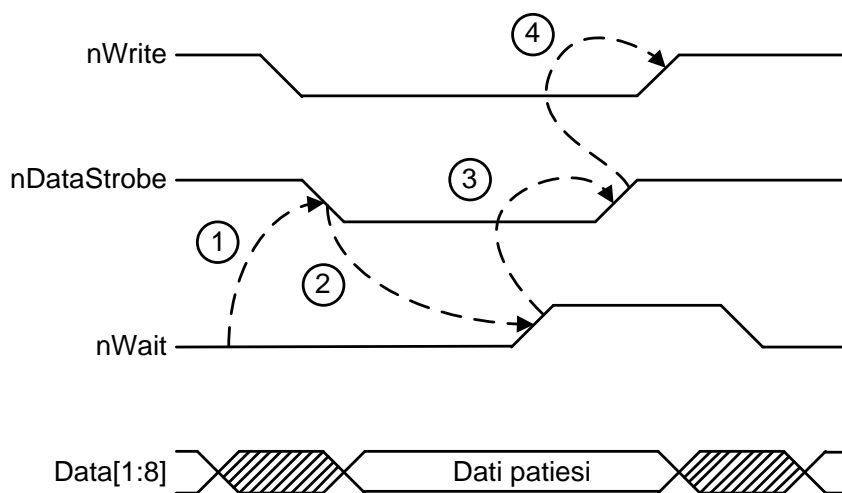
9.3. tabula

EPP protokola signāli

SPP signāls	EPP signāls	Signāla virziens	EPP protokola signāla apraksts
<i>nSTROBE</i>	<i>nWRITE</i>	Uz ierīci (Out)	Zems līmenis iniciē ierakstīšanas ciklu, augsts – nolasīšanas.
<i>nAUTOFEED</i>	<i>nDATASTB</i>	Uz ierīci (Out)	Iestatās zems līmenis, norādot uz to, ka izpildās datu ierakstīšana/ nolasīšana.
<i>nSELECTIN</i>	<i>nADDRSTB</i>	Uz ierīci (Out)	Iestatās zems līmenis, norādot uz to, ka izpildās adreses ierakstīšana/nolasīšana.
<i>nINIT</i>	<i>nRESET</i>	Uz ierīci (Out)	Iestatās zems līmenis, ja nepieciešams noņemt ierīces iestatījumus.
<i>nACK</i>	<i>nINTR</i>	No ierīces (In)	Pārtraukuma pieprasījums no ārējās ierīces.

<i>BUSY</i>	<i>nWAIT</i>	No ierīces (In)	<i>Handshaking</i> signāls. Zems līmenis signalizē par cikla sākšanas iespēju. Ja signāla līmenis kļūst augsts, var pabeigt ciklu.
<i>D[8:1]</i>	<i>AD[8:1]</i>	<i>Bi-directional</i>	Adrešu-datu abu virzienu līnijas.
<i>PE</i>	Nosaka izstrādātājs	No ierīces (In)	Ierīces var izmantot specifiskā nolūkā.
<i>SELECT</i>	Nosaka izstrādātājs	No ierīces (In)	Ierīces var izmantot specifiskā nolūkā.
<i>nERROR</i>	Nosaka izstrādātājs	No ierīces (In)	Ierīces var izmantot specifiskā nolūkā.

Labākai EPP protokola darbības izpratnei 9.3. attēlā parādīta datu ierakstīšanas cikla laika diagramma.



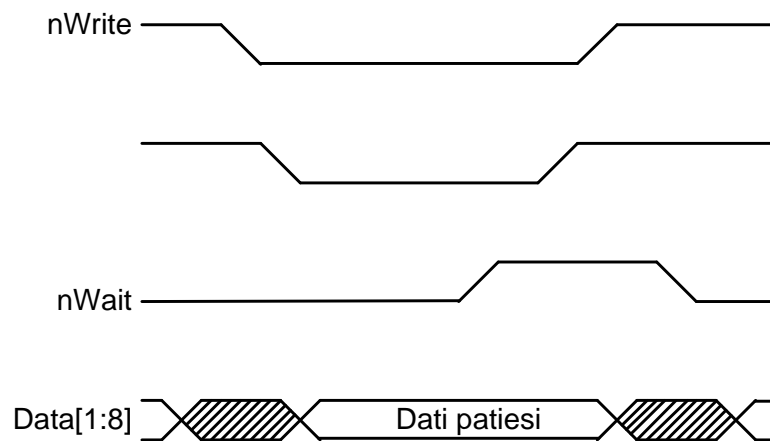
9.3. att. EPP protokola datu ierakstīšanas cikls

! EPP protokolā, tāpat kā daudzos citos, tiek izmantots rokspiešanas princips, kas šajā gadījumā nosaka to, ka viena signāla iestatījumi ir atkarīgi no citiem šīs ierīces signāliem.

Datu ierakstīšana sākas ar zema signāla *nWrite* līmeņa iestatīšanu (to dara datu apmaiņas iniciators). Ja mērķis (printeris vai cita ierīce) iestatīja signāla *nWait* (*nWAIT*) zemu līmeni (kas liecina par ierīces gatavību), tad avots iestata signāla *nDataStrobe* (*nDATASTB*) aktīvu līmeni – ①. No šī brīža dati līnijās *Data[1:8]* (*D[8:1]*) tiek uzskatīti par gataviem jeb patiesiem, un ierīce var tos pieņemt.

Pēc datu saņemšanas ierīce noņem aktīvā signāla līmeni līnijā *nWait* – ②. Atbildot uz to, datu avots noņem signālu *nDataStrobe* – ③ un *nWrite* aktīvo līmeni, tādējādi pabeidzot datu ierakstīšanas ciklu – ④.

9.4. attēlā ir nodemonstrēta adreses nolasīšanas cikla laika diagramma.



9.4. att. EPP protokola adreses nolasīšanas cikls

Tā kā tiek veikta nolasīšanas operācija, tad signālam līnijā *nWrite* ir augsts līmenis, bet signāls līnijā *nAddrStrobe* (*nADDRSTB*) tiek iestatīts zemā līmenī. Avota un mērķa signālu sinhronizēšana tiek veikta tāpat kā iepriekšējā piemērā.

EPP protokols aparatūras līmenī realizēts ar 8 reģistru palīdzību - 3 no tiem mantoti no SPP protokola (sk. 9.4. tabulu).

9.4. tabula

EPP protokola reģistri

Adrese	Reģistra nosaukums	Pieļaujamās operācijas
Base+0	Datu reģistrs (SPP)	Ierakstīšana
Base+1	Stāvokļa reģistrs (SPP)	Nolasīšana
Base+2	Vadības reģistrs (SPP)	Ierakstīšana
Base+3	Adreses reģistrs (EPP)	Nolasīšana/ Ierakstīšana
Base+4	Datu reģistrs (EPP)	Nolasīšana/ Ierakstīšana
Base+5	Rezervēts	-
Base+6	Rezervēts	-
Base+7	Rezervēts	-

Kā redzams no tabulas, pirmās trīs adreses atbilst paralēlās pieslēgvietas standarta režīmam, t.i., *Centronics* saskarnei. Tas nozīmē to, ka EPP režīmā var vērsties pie pirmajām trim pieslēgvietām pilnīgi tāpat, kā strādājot ar standarta paralēlo pieslēgvietu.

Ja izpildās datu apmaiņa ar ierīci pēc EPP protokola, tad, piemēram, datu nosūtīšanai tie jāievieto datu reģistrā ar adresi *Base+4* – EPP saskarnes aparatūras daļa nodrošinās automātisku visu signālu sinhronizāciju. Tā kā aparatūras daļa darbojas ātrāk par programmas daļu, tad datu apmaiņa EPP režīmā notiek ātrāk nekā ar SPP izmantošanu.

9.3. ECP protokols

ECP protokolu piedāvāja kompānijas *Hewlett Packard* un *Microsoft* kā SPP un EPP standartu paplašinājumu lielu informācijas masīvu no skeneriem, printeriem un citām ierīcēm efektīvai apstrādei. ECP arī nodrošina datu apmaiņu abos virzienos – starp datoru un ārējām ierīcēm. ECP protokols darbojas ar diviem ciklu tipiem – datu un komandu cikliem.

Komandu cikli savukārt iedalās divos tipos: datu izmēra rādītājos (*Run-length Count*) un apmaiņas kanālu adresēs (*Channel Address*).



ECP protokols ļauj apstrādāt liela izmēra, mainīga garuma datu masīvus (tā sauktā RLE tehnoloģija – (*Run_lenght Encoding data compression*)).

ECP protokols var strādāt ar FIFO datu rindām, kā arī ar DMA.

RLE tehnoloģija dod iespēju saspiest datus attiecībā 64:1, t.i., ļoti efektīvi var apstrādāt no skenera iegūtos attēlus un drukāt lielus datu apjomus. Vienīgā prasība - datoram un ārējām ierīcēm jāatbalsta šī tehnoloģija.

ECP kanālu adresācija atšķiras no tās, kas tika izmantota protokolā EPP, un pamatojas uz vienas fiziskās ierīces vairāku loģisko ierīču koncepciju. Piemēram, tāda kombinētā ierīce kā *Fax/Modem/Printer*, kurai ir viena fiziskā paralēlā pieslēgvietā, var darboties kā trīs neatkarīgas loģiskās ierīces. Turklāt ECP protokola kanālu adresācija ļauj multipleksēt datu apmaiņu starp loģiskajām ierīcēm. Ja, piemēram, printeris drukā un tā kanāls nav pieejams, tad ierīces draiveris var pārslēgties uz citu kanālu (modema vai faksa) un apstrādāt datus.

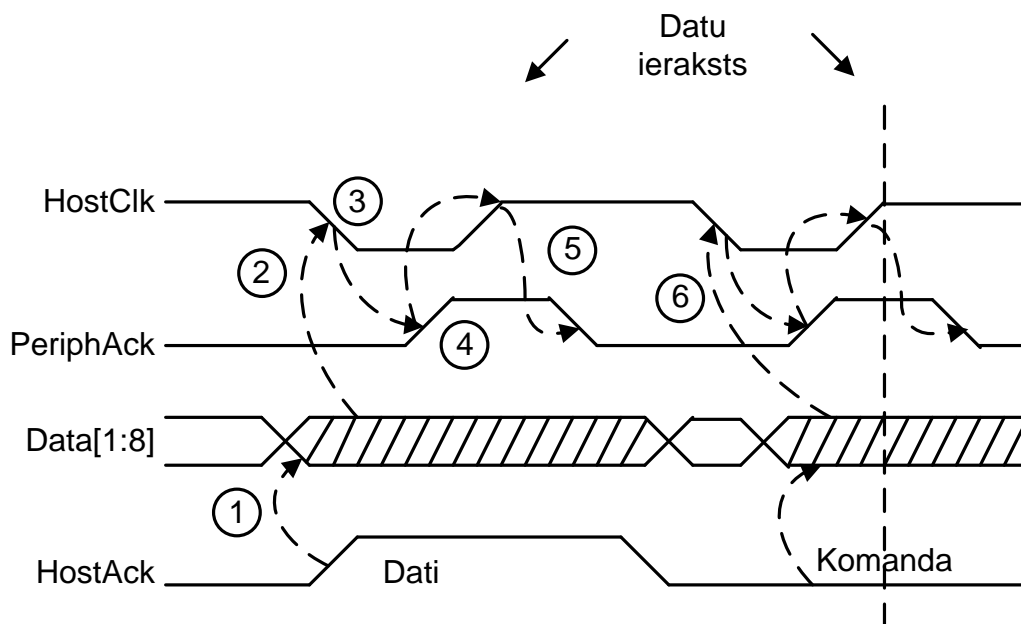
ECP saskarnes signāli ir aprakstīti 9.5. tabulā.

9.5. tabula

ECP saskarnes signāli

SPP signāls	ECP signāls	Signāla virziens	ECP signālu apraksts
<i>nSTROBE</i>	<i>HostClk</i>	Izvade	Izmanto kopā ar <i>PeriphAck</i> datu vai adreses pārraidei uz ierīci
<i>nAUTOFEED</i>	<i>HostAck</i>	Izvade	Norāda datu vai komandas statusu pārraidei tiešā virzienā (uz ierīci). Izmanto kopā ar <i>PeriphClk</i> datu pārraidei pretējā virzienā.
<i>nSELECTIN</i>	<i>1284Active</i>	Izvade	Iestata augstu līmeni pie pārraides režīma 1284.
<i>nINIT</i>	<i>nReverseRequest</i>	Izvade	Iestata zemu līmeni kanāla darbam pretējā virzienā.
<i>nACK</i>	<i>PeriphClk</i>	Ievade	Izmanto kopā ar <i>HostAck</i> datu pārraidei pretējā virzienā.
<i>BUSY</i>	<i>PeriphAck</i>	Ievade	Izmanto kopā ar <i>HostClk</i> datu vai adreses pārraidei tiešā virzienā. Norāda datu vai komandas statusu pārraidei pretējā virzienā.
<i>PE</i>	<i>nAckReverse</i>	Ievade	Iestata zemu līmeni, atbildot uz <i>nReverseRequest</i> .
<i>SELECT</i>	<i>Xflag</i>	Ievade	Paplašinājuma karodziņš.
<i>nERROR</i>	<i>nPeriphRequest</i>	Ievade	Iestata zemu līmeni, kas norāda uz pieeju pretējā virzienā pārraidāmajiem datiem.
<i>Data[8:1]</i>	<i>Data[8:1]</i>	Ievade/Izvade	Izmanto datu apmaiņā starp datoru un ārējo ierīci.

Analizē datu apmaiņas pēc ECP protokola laika diagrammu, kad dati no hosta tiek pārraidīti uz ārējo ierīci [9] (sk. 9.5. att.).



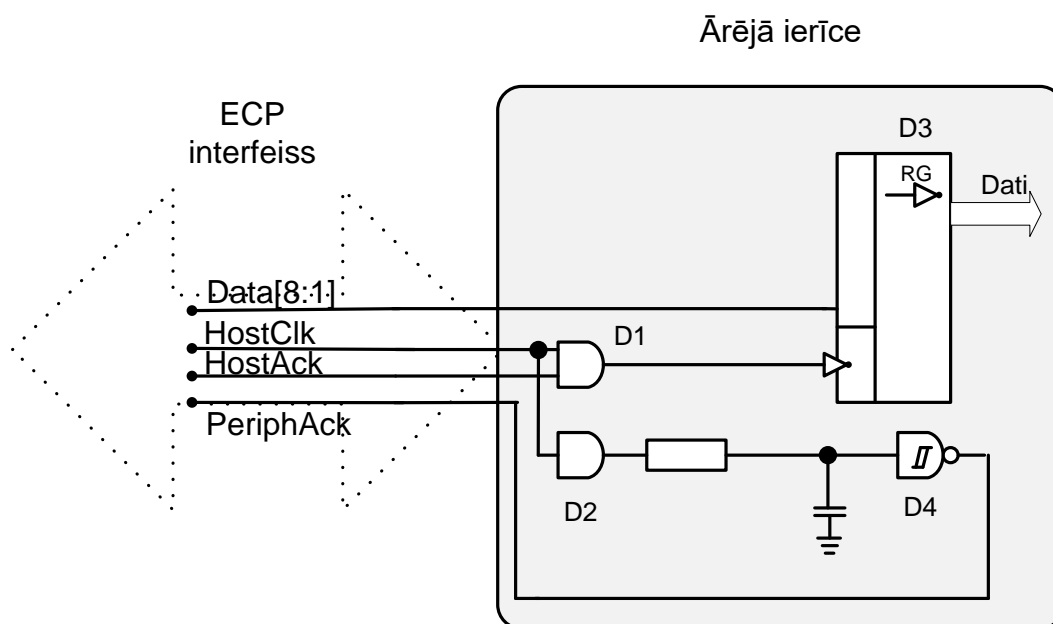
9.5. att. Datu ieraksta cikls ārējā ierīcē pēc ECP protokola

Šeit datu ierakstīšanas ciklu iniciē hosts, izvietojot datus līnijās $Data[8:1]$ un iestatot augstu signāla $HostAck$ līmeni – ①.

Sinhroimpulsam $HostClk$ tiek iestatīts zems līmenis, kas liecina par datu gatavību - ②. Ārējā ierīce atbild ar augstu signāla $PeriphAck$ līmeņa iestatīšanu - ③, pēc kā hosts iestata augstu signāla $HostClk$ līmeni – ④.

Ierīce iestata zemu signāla $PeriphAck$ līmeni, tādējādi norādot uz gatavību sākt pieņemt jaunu datu baitu – ⑤. Hosts iestata zemu signāla $HostAck$ līmeni, norādot uz to, ka nākamajā ciklā tiks pārraidīta komanda, nevis dati ⑥.

Tāda apmaiņas cikla viens no iespējamiem aparatūras realizācijas veidiem attēlots 9.6. attēlā.



9.6. att. Datu ierakstīšanas realizācijas shēma

Šī shēma ir ļoti vienkāršota un to var aplūkot tikai datu ierakstīšanas principa demonstrēšanas nolūkā.

Datu ierakstīšana tiek veikta pie augsta signāla *HostAck* līmeņa vienlaikus ar pieaugošo sinhroimpulsa *HostClk* līmeni. Tas realizēts ar mikroshēmas D1 palīdzību, kura veic datu ierakstu reģistrā D3. Tā kā signāls *PeriphAck* invertēts un nobīdīts laikā, ar D2 un D4 palīdzību tiek veidota ierīces atbildes reakcija.

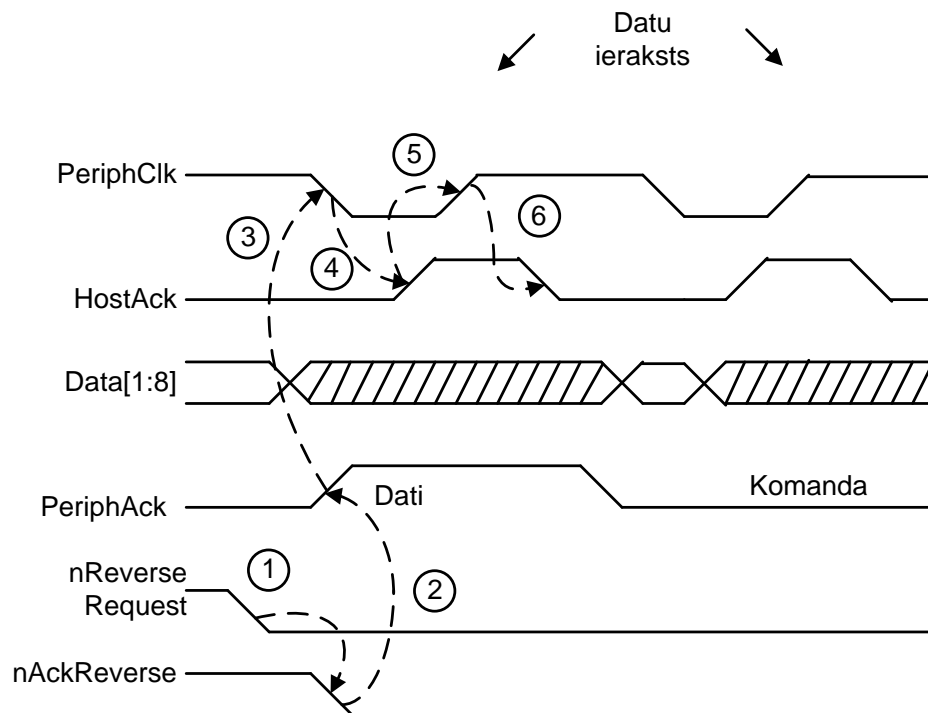


Par apmaiņas iniciatoru var kalpot arī ārējā ierīce – šajā gadījumā datu pārraides virziens mainās uz pretējo, turklāt izmaiņas virzienam savlaicīgi tiek piemērots rokspiešanas princips. Tādas datu apmaiņas shēma parādīta 9.7. attēlā.

Salīdzinot ar tiešo datu apmaiņu, apgrieztajam jeb reversajam režīmam nepieciešamas divas papildus signālu līnijas: *nReverseRequest* un *nAckreverse*.

Apmaiņas sākumā hosts iestata zemu signāla līmeni *nReverseRequest* līnijā, paziņojot ierīcei par datu pārraides nepieciešamību reversajā režīmā – ①. Ārējā ierīce apstiprina gatavību darbam reversajā režīmā, iestatot zemu signāla *nAckReverse* līmeni, tad izvieto datus *Data[8:1]* līnijās un paziņo par datu apmaiņas cikla sākumu ar augstu signāla *PeriphAck* līmeni – ②.

Turpmāk ārējā ierīce iestata zemu signāla *PeriphClk* līmeni, tādējādi norādot hostam par datu autentiskumu – ③. Hosts atbild ar augsta signāla *HostAck* līmeņa iestatīšanu – ④, pēc kā ārējā ierīce noņem signālu *PeriphClk* – ⑤. Šajā brīdī datus var saglabāt, izmantojot pieaugošo signāla *PeriphClk* līmeni. Pēc tam hosts iestata signālu *HostAck* zemā līmenī, norādot uz gatavību jaunam ciklam.



9.7. att. Datu pārraide no ārējās ierīces

ECP realizēts ar reģistru palīdzību, turklāt pirmie trīs no tiem pilnībā atkārtoti standarta SPP protokola funkcijas. Jāatzīmē, ka vadības reģistra 5. bits nosaka datu apmaiņas virzienu un ietekmē atsevišķu bitu iestatījumus reģistrā ECR (*Extended Control Register*), ar kura palīdzību tiek noteikts ECP darba režīms.

ECP reģistru nozīme aprakstīta 9.6. tabulā.

ECP reģistri

Adrese	Reģistra nozīme	Pieļaujamās operācijas
Base+0	Datu reģistrs (SPP)	Ierakstīšana
	FIFO reģistra adrese (ECP)	Nolasīšana/ Ierakstīšana
Base+1	Stāvokļa reģistrs (visi režīmi)	Nolasīšana/ Ierakstīšana
Base+2	Vadības reģistrs (visi režīmi)	Nolasīšana/ Ierakstīšana
Base+0x400	FIFO datu reģistrs (SPP)	Nolasīšana/ Ierakstīšana
	FIFO datu reģistrs (ECP)	Nolasīšana/ Ierakstīšana
	FIFO testa reģistrs (testa režīms)	Nolasīšana/ Ierakstīšana
	Konfigurācijas reģistrs A (konfigurēšana)	Nolasīšana/ Ierakstīšana
Base+0x401	Konfigurācijas reģistrs B (konfigurēšana)	Nolasīšana/ Ierakstīšana
Base+0x402	Reģistrs ECR (visi režīmi)	Nolasīšana/ Ierakstīšana

Starp tabulā uzskaitītajiem reģistriem būtiska loma ir reģistram ECR, jo no tā iestatījumiem atkarīgs paralēlās pieslēgvietas darba režīms. Piemēram, lai iestatītu pieslēgvietas standarta režīmu, bitiem 5-7 ir jābūt 0. Pārējiem bitiem ir specifiska nozīme, un tie netiek apskatīti.

Piemēra nolūkā dots asamblera fragments datu baita izvadei ECP pieslēgvietā (salīdzinājumā ar 9.1. piemēru šis ir ļoti vienkāršs).



9.2. piemērs

```
mov AL, <dati>
mov DX, <datu pieslēgvietā>
out DX, AL
```

Komentāri

Par datu pieslēgvietu jāuzrāda datu reģistra adrese, kas sakrīt ar *Centronics* standartu. Tā kā ECP režīmā saskarnes signāli izpildās aparatūras līmenī, maksimums, kas jāizdara programmai – jāiesūta dati pieslēgvietā. Faktiski datu baita iesūtīšanai ECP režīmā nepieciešamas tikai dažas instrukcijas asamblerā (dotajā piemērā 3 instrukcijas).

9.4. Lietotāja programmu saskarne

Vispārpieņemts, ka paralēlā pieslēgvietā tiek izmantota dažādu drukas ierīču, skeneru utt. pieslēgšanai, kaut arī tās izmantošana nav ierobežota tikai ar to. Paralēlās pieslēgvietas saskarne ir ļoti ērta dažādu vadības un kontroles iekārtu izmantošanā gan rūpniecībā, gan laboratorijas pētījumos [9,23].

Vairākkārtīgi tika teikts, ka OS *Windows* jaunākās versijas nedod iespēju lietotāja programmām tiešā veidā strādāt ar paralēlās pieslēgvietas reģistriem (tāpat arī ar citām ievadizvades pieslēgvietām). Tas saistīts ar to, ka piekļūt ievadizvades pieslēgvietām var tikai kodola režīmā izpildāmie procesi, piemēram, ierīču draiveri vai atsevišķi sistēmas dienesti.

Teorētiski darboties ar paralēlās pieslēgvietas ievadizvadi var ar WinAPI funkciju palīdzību: *CreateFile*, *ReadFile* un *WriteFile*. Šim nolūkam ar funkcijas *CreateFile* palīdzību jāatver ierīce, kā pirmo parametru uzdodot „LPTn” (n – paralēlās pieslēgvietas numurs, parasti n=1). Pēc ierīces deskriptora saņemšanas ar funkcijas *CreateFile* palīdzību var veikt ierakstīšanas/nolasīšanas operācijas ierīcē.

Tamlīdzīgas darbības demonstrē koda fragments no 9.3. piemēra.



9.3. piemērs

```

#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
int _tmain(int argc, _TCHAR* argv[])
{
    //.....
    HANDLE hLPT;
    char *buf="Kaut kas drukajams!";
    DWORD bytes;
    //.....
    hLPT = CreateFile("LPT1",
                    GENERIC_READ | GENERIC_WRITE,
                    0,
                    NULL,
                    OPEN_EXISTING,
                    0,
                    NULL);
    if (hLPT == INVALID_HANDLE_VALUE)
    {
        printf("Nevar atvert failu (error %d)\n", GetLastError());
        getchar();
        return 0;
    }
    bool fSuccess=WriteFile(hLPT,
                            buf,
                            strlen(buf),
                            &bytes,
                            NULL);
    //.....
    return 0;
}

```

Komentāri

Tādā veidā programmā var izvadīt datus uz printeri, bet neizdosies veikt datu ierakstīšanu tieši ierīces reģistrā. Programmas koda fragments strādā ar ierīci, izmantojot sistēmas draiveri. WinAPI funkciju izmantošanas gadījumā I/O pārvaldnieks veido standarta pieprasījuma paketi IRP, kas vēršas pie paralēlās pieslēgvietas draivera un veic datu apmaiņu. Pats draiveris izstrādāts tādā veidā, lai varētu pielietot uzstādītās SPP, EPP vai ECP saskarnes.

Lai dotu ieskatu par to, kādā veidā no lietojumprogrammas var vadīt paralēlās pieslēgvietas signālus – tiek formulēts šāds uzdevums: „Pieslēgvietā tiek iesprausta gaismas diode. Uzrakstīt programmu, ar kuras palīdzību, nospiežot attiecīgos taustiņus, var panākt lampiņas mirgošanu”.

Programmas kods dots 9.4. piemērā, bet darbu izpildes secība aprakstīta komentāros.



9.4. piemērs. Paralēlās pieslēgvietas tests

```

//port.cpp - LPT1 tests
#include "stdafx.h"
#include "conio.h"
#include "stdlib.h"
    // DLL definīcijas:
    short _stdcall Inp32(short PortAddress);
    void _stdcall Out32(short PortAddress, short data);

int _tmain(int argc, _TCHAR* argv[])
{
    int Address=888;
    printf("LPT tests\n");
    printf("Spied '1' gaismas diodes aktivācijai un '0' deaktivācijai.\n");
    printf("Izeja - 'e' taustīns.\n");
}

```

```

while(1)
{
    switch(getch())
    {
        case '0':    printf("Deaktivacija...\n");
                    Out32(Address, 0);
                    break;

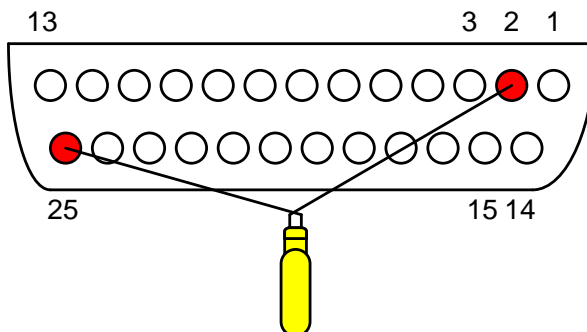
        case '1':    printf("Aktivacija...\n");
                    Out32(Address, 1);
                    break;

        case 'e':    exit(1);
    }
}
getch();
return 0;
}

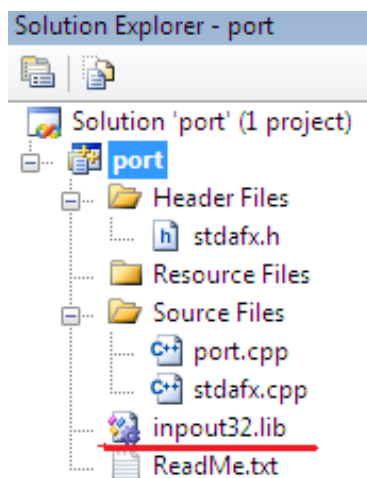
```

Komentāri

1. Iesprauž gaismas diodes izvadus printera pieslēgvietas 2. un 25. kontaktā, kā parādīts attēlā (ja izvadi iesprausti pareizi – lampiņa deg):



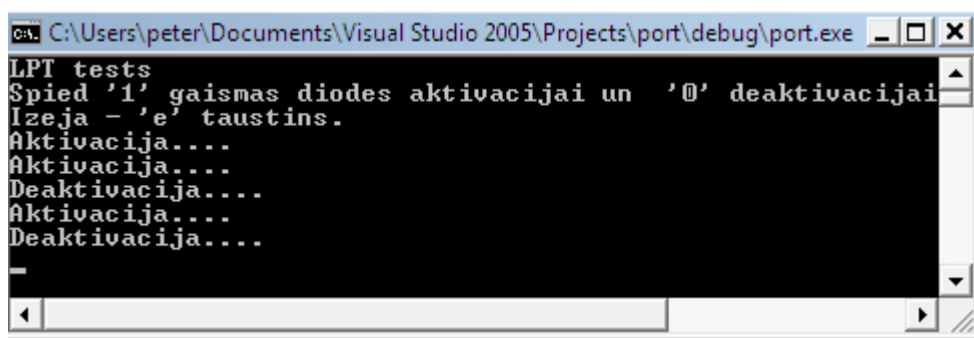
2. Iekopē datorā arhīvu, kurā būs dinamiskā bibliotēka *inpout32.dll* no <http://www.logix4u.net/inpout32.htm> (Download *inpout32.dll* and source code here).
3. *Visual Studio 2005* vidē izveido jaunu C++ projektu – konsoles aplikāciju un uzraksta programmas tekstu.
4. Projektam pievieno statisko bibliotēku *inpout32.lib* no 2. solī iekopētā arhīva (no mapes \test applications\VC_test_app):
Project\Add Existing ItemAll files....izvēlas *inpout32.lib*.....Add.



5. Kompilē projektu: Build\Build Solution
6. Projekta mapē \DEBUG iekopē dinamisko bibliotēku *inpout32.dll* no 2. solī iekopētā arhīva (no mapes \Binaries\Dll).
7. Var sākt testēšanu : \Debug\Start debugging.

Rezultāts

Programma tika testēta *Windows XP* vidē. Attiecīgi nospiežot ciparu taustiņus 1 un 0, tiek panākta gaismas diodes mirgošana (ar komandas *Out32(address,1)* palīdzību paralēlajā pieslēgvietā ar adresi 378h tiek iesūtīts datu bits 1, kura signāls liek mirgot lampiņai).

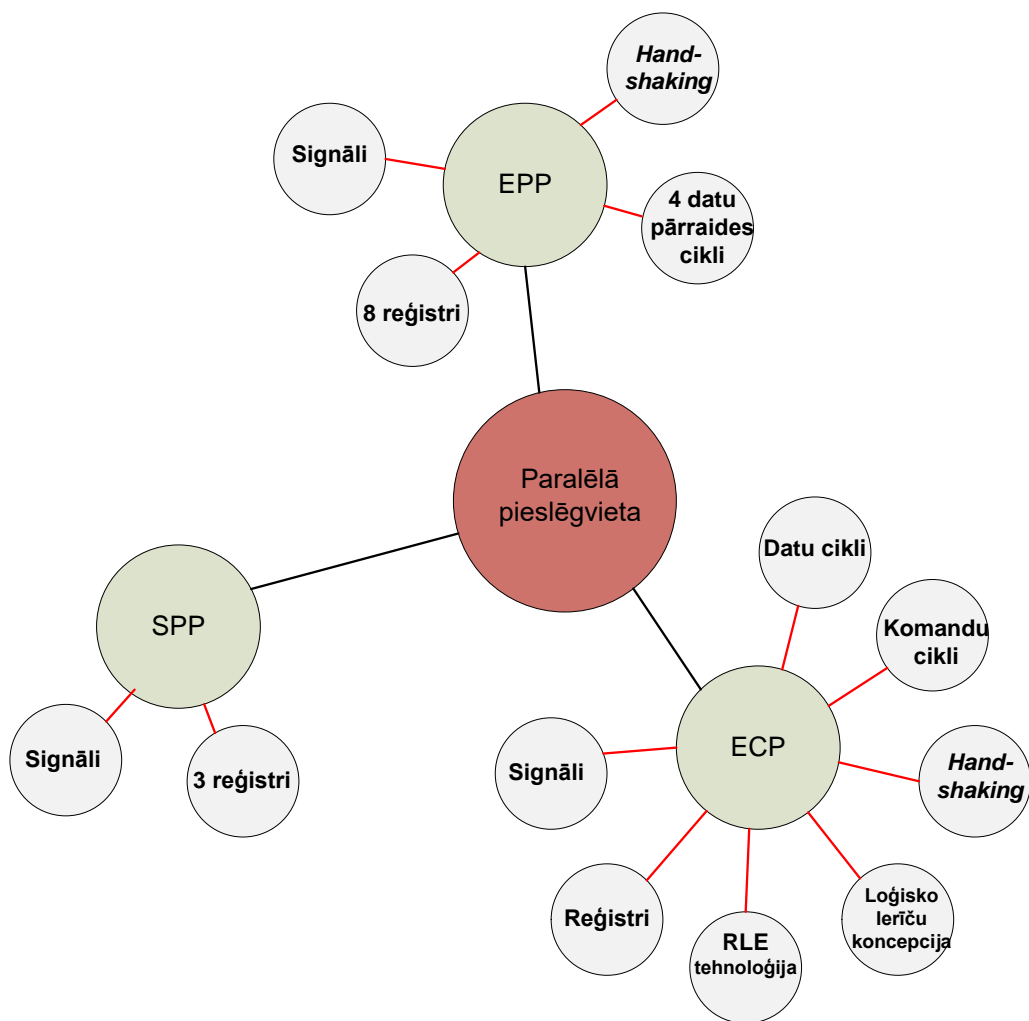


```
C:\Users\peter\Documents\Visual Studio 2005\Projects\port\debug\port.exe
LPT tests
Spied '1' gaismas diodes aktivacijai un '0' deaktivacijai
Izeja - 'e' taustins.
Aktivacija....
Aktivacija....
Deaktivacija....
Aktivacija....
Deaktivacija....
```

NODAĻAS KOPSAVILKUMS

- Paralēlā pieslēgvietā ir viena no senākajām aparatūras un programmu saskarnēm, kas tika izstrādātas datu apmaiņai ar drukas ierīcēm.
- Pieslēgvietā dod iespēju saņemt 9 bitu datus vai sūtīt 12 bitu datus. Aparatūras saskarne satur 4 vadības līnijas, 5 stāvokļa līnijas un 8 datu līnijas.
- Datu apmaiņas procesā starp ierīcēm viena no tām ir apmaiņas iniciators (avots), bet otra – mērķis. Par datu apmaiņas iniciatoru parasti kalpo dators (hosts), bet par mērķi – printeris vai cita ierīce.
- Paralēlās pieslēgvietas bāzes modelis realizēts kā triju reģistru kopums: datu, stāvokļa un vadības reģistri.
- Tiešā ievadizvades pieslēgvietu programmēšana ar asamblera palīdzību bija iespējama tikai OS *Windows 98/ME*, jo visi sistēmas resursi pieejami lietojumprogrammu procesiem ar instrukciju *in* un *out* palīdzību.
- Protokols EPP tika izstrādāts ar nolūku paplašināt standarta paralēlās pieslēgvietas (SPP) iespējas. EPP protokolā, tāpat kā daudzos citos, tiek izmantots rokspiešanas princips, kas šajā gadījumā nosaka to, ka viena signāla iestatījumi ir atkarīgi no citiem šīs ierīces signāliem.
- Par apmaiņas iniciatoru var kalpot arī ārējā ierīce – šajā gadījumā datu pārraides virziens mainās uz pretējo, turklāt izmaiņas virzienam savlaicīgi tiek piemērots rokspiešanas princips.
- ECP protokols ļauj apstrādāt liela izmēra mainīga garuma datu masīvus (tā sauktā RLE tehnoloģija – *Run_lenght Encoding data compression*).

9.8. attēlā parādīts devītajā nodaļā minēto svarīgāko jēdzienu koks.



9.8. att. Nodaļā minēto svarīgāko jēdzienu koks

Uzdevumi un jautājumi patstāvīgajam darbam

1. Veikt SPP un ECP salīdzinājumu.
2. Raksturot rokspiešanas principu EPP un ECP protokolos.
3. Kā izpaužas RLE tehnoloģijas darbība lielu informācijas masīvu izdrukāšanā?
4. Pēc kāda principa vienā fiziskajā ierīcē apvienotas vairākas loģiskās ierīces?
5. Kāda ir reverso signālu līniju funkcijas?
6. Analizēt datu apmaiņas laika diagrammas pēc EPP un ECP protokoliem.
7. Raksturot paralēlās pieslēgvietas programmēšanas problēmas.

10. DATU IEVADIZVADE CAUR SERIĀLO PIESLĒGVIETU

Seriālā pieslēgvietā ir viena no datorsistēmas ārējām saskarnēm, ar kuras palīdzību tiek veikta datu apmaiņa saskaņā ar RS-232 standartu. Lai pieslēgtu šim standartam atbilstošas ierīces, tiek izmantots 9 kontaktu savienotājs (dažkārt satopami arī 25 kontaktu savienotāji). Līdzīgi kā paralēlā pieslēgvietā – arī seriālā pieslēgvietā ir viena no vecākajām aparatūras un programmu saskarnēm, kas tika izstrādāta datu apmaiņai ar dažādām ierīcēm.

Nodaļā apskatīti datu apmaiņas caur seriālo pieslēgvietu darbības principi, signālu apraksts un programmēšanas iespējas.

10.1. Seriālās pieslēgvietas raksturojums



Seriālā pieslēgvietā (*serial port*) ir ievadizvades pieslēgvietā, kas nodrošina informācijas pārsūtīšanu virknes formātā starp datoru un ārējām ierīcēm.

Parasti datoriem bija RS-232 standarta seriālā pieslēgvietā, retāk lietoja RS-422. Datu apmaiņa pēc RS-232 standarta notiek virknes veidā – bits pēc bita, no šejienes arī ir nosaukums – virknes jeb seriālā pieslēgvietā. Pirms plašas USB izplatības šo pieslēgvietu lietoja, lai pieslēgtu gandrīz visu veidu ārējās ierīces (peli, tastatūru, modemu). Seriālās pieslēgvietas lietoja un vēl joprojām lieto datoru savienošanai ar specifiskām iekārtām (sviri, laboratorijas mērinstrumenti, termostati, GPS uztvērēji). Pirms plašas PS/2 pieslēgvietas izplatības seriālā pieslēgvietā bija galvenā metode peles pieslēgšanai [3,4,5,8,9].

Sākotnējā RS-232 standartā bija paredzēts lietot savienotāju ar 25 kontaktiem, taču lielākajai daļai datoru seriālajām pieslēgvietām ir 9 kontakti. MS-DOS vidē seriālajām pieslēgvietām varēja piekļūt kā ierīces datnēm ar nosaukumu COMx, kur x – pieslēgvietas numurs. Datoriem parasti bija 2 seriālās pieslēgvietas (COM1 un COM2), lai arī diezgan bieži bija vairāk (3 vai 4). Windows vidē seriālās pieslēgvietas arī apzīmē kā COM, tāpēc tās dažreiz sauc par COM pieslēgvietām. Mūsdienās visas seriālās pieslēgvietas funkcijas spēj nodrošināt USB pieslēgvietas. Tās ir mazākas, vienkāršākas, spēj nodrošināt lielākus ātrumus, signāli ir noturīgāki pret traucējumiem, taču to programmēšana ir nesalīdzināmi sarežģītāka.

Datorsistēmas standarta konfigurācijā seriālajām pieslēgvietām tiek piešķirti šādi aparatūras resursi (sk. 10.1. tabulu).

10.1. tabula

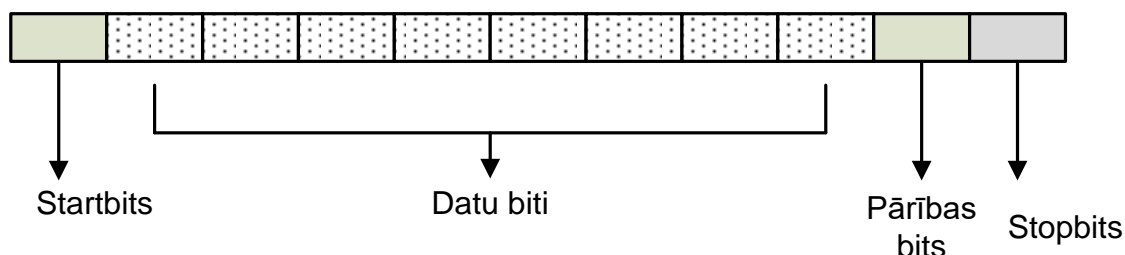
Seriālo pieslēgvietu resursi

Pieslēgvietas numurs	Pārtraukumu līnija	Reģistru bāzes adrese
COM1	IRQ4	0x3f8
COM2	IRQ3	0x2f8
COM3	IRQ4	0x3e8
COM4	IRQ3	0x2e8



Datu apmaiņa caur seriālo pieslēgvietu tiek realizēta sinhronajā vai asinhronajā režīmā. Sinhronajā režīmā datu avotam un datu mērķim jāstrādā vienā frekvencē. Datu apmaiņas sākumā avots sūta mērķim sinhronizējošo bitu virkni, pēc kuras seko pārsūtāmo bitu virkne. Sinhronajā režīmā datu apmaiņas ātrums ir ievērojami ātrāks nekā asinhronajā, jo nav nepieciešamības pārraidīt startbitu, stopbitu un pārības bitu. Sinhronās datu apmaiņas trūkums ir tas, ka avotam un mērķim jābūt augstai frekvences stabilitātei, citādi frekvenču nesaskaņotība var veicināt kļūdu uzkrāšanos.

Asinhronajā režīmā pirms katra baita ir startbits, pēc tam seko datu biti, pārības bits un datu pārraides noslēgumā stopbits, kas garantē noteiktu laika aizturi starp datu sūtījumiem jeb kadriem (sk. 10.1. att.).



10.1. att. Pārsūtāmo datu struktūra

Seriālās pieslēgvietas datu formātu parasti uzdod šādā pierakstā:

Datu_bitu_skaits – Pārības_tips – Stopbitu_skaits.

Piemēram, 8-N-1 tiek interpretēta kā 8 bitu datu pārraide bez pārības kontroles un ar vienu stopbitu. 7-E-2 gadījumā tiek pārsūtīti 7 biti ar pārības kontroli un 2 stopbitiem.

i Datu biti pārsvarā tiek interpretēti kā ASCII simbola biti. Amerikas informācijas apmaiņas standartkods jeb ASCII (*American Standard Code for Information Interchange*) ir rakstzīmju kopa un kodējums, kas balstīts uz angļu valodā un citās Rietumeiropas valodās lietoto latīņu alfabētu. Visplašāk to izmanto datoros un citās komunikāciju ierīcēs, lai attēlotu tekstu un kontrolētu ierīces, kas strādā ar tekstu. ASCII definē 7 bitu kodējumu, un tur ir 95 attēlojami simboli (32-126).

Startbits var tikt pārraidīti jebkurā brīdī pēc iepriekšējā sūtījuma stopbita. Datu bitu skaits var būt 5, 6, 7 vai 8 biti, bet stopbitu skaits – 1 vai 2. Pārības bits var arī nebūt.

! Atšķirībā no paralēlās pieslēgvietas, lai lietotu virknes pieslēgvietu, ir nepieciešams norādīt vairākus parametrus [5]:

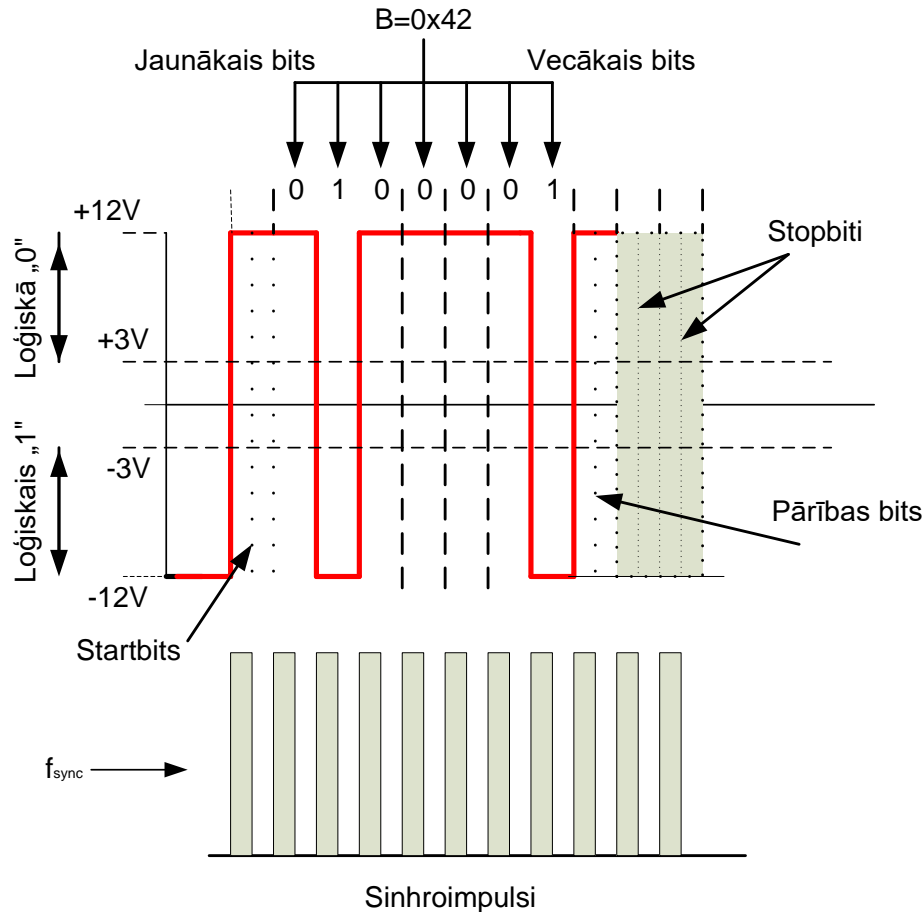
- datu pārraides ātrumu. RS-232 standartā paredzēts datu pārraides ātrums līdz 19200 bitiem sekundē. Taču reāli bieži vien lieto ātrumus līdz 115200 b/s (pēdējo gadu izstrādes ļauj palielināt šo ātrumu līdz 460 Kb/s);
- datu bitu skaitu. RS-232 standarts tika izstrādāts 20. gs. 60. gados, kad vēl nebija skaidri noteikts, ka baitā ir 8 biti, tāpēc te ir jānorāda datu bitu skaits (baita garums bitos), tas var būt robežās no 5 līdz 9 bitiem. Parasti lieto vai nu 7 (ASCII simboliem) vai 8 (jebkādiem citiem datiem);
- pārību (*parity*). Nosūta papildus bitu, lai pārbaudītu, vai baitā pārsūtot nav radusies kļūda;
- stopbitu - pēc baita nosūtīšanas nosūta vēl noteiktu daudzumu bitu, lai norādītu baita beigas;
- plūsmas vadība (*flow control*). Ja savieno dažāda ātruma ierīces, tad šis mehānisms ļauj sinhronizēt ātrāko ierīci ar lēnāko ierīci. Tā realizēšanai ir divas metodes:
 - aparatūras jeb RTS/CTS - te lieto papildu 2 līnijas. RTS (*ready to send*) aktivizē avots, lai norādītu, ka ir gatavs sūtīt datus. CTS (*clear to send*) aktivizē mērķis, lai norādītu, ka ir gatavs saņemt datus.
 - ja komunikācijām ir pieejamas 3 līnijas, tad var lietot programmu jeb XON/XOFF metodi. Šeit, ja mērķis netiek galā ar datiem, tas nosūta atpakaļ simbolu XOFF, lai apturētu avota datu sūtīšanu, un tad, kad var atsākt saņemt datus, nosūta XON.

Seriālo saskarni izmanto visi mūsdienu datu pārraides standarti (USB, Firewire utt.), jo ar tās palīdzību ir iespējams panākt lielākus ātrumus (nav nepieciešams sinhronizēt paralēlās bitu plūsmas).

10.2. RS-232 saskarne

i RS-232 ir ASV elektroniskās rūpniecības uzņēmumu apvienības izstrādāts saskarnes standarts, kas paredzēts datu pārraidei, izmantojot seriālās pieslēgvietas. Vairums personālo datoru ir apgādāti ar RS-232 saderīgām seriālajām pieslēgvietām, ko var izmantot modemu, printeru, skeneru un citu ārējo ierīču pievienošanai.

Asinhronās datu apmaiņas pamatā, kas tiek izmantota RS-232 saskarnes funkcionēšanā, ir atsevišķu datu kadru (freimu) apstrāde. RS-232 saskarne izmanto vairākas signālu līnijas, tai skaitā TD un RD, kas atbild par datu pārraidi (sk. 10.2. att.).



10.2. att. Datu baita pārsūtīšana

Loģiskie signālu līmeņi atrodas diapazonā no -3V līdz -12V (loģiskais 1) un no 3V līdz 12V (loģiskā 0). Diapazons no -3V līdz +3V atbilst neaktīvajam stāvoklim (histerēze).

10.2.attēlā ir demonstrēta simbola „B” pārraide asinhronajā režīmā – 7 datu biti, pārības bits un 2 stopbiti. Pirms katra datu baita ir startbits.

Pēc bitu pārraides tiek nosūtīts pārības bits (šajā gadījumā 0) un datu pārraidi noslēdz 2 stopbiti, kas garantē zināmu laika aizturi starp datu kadriem. Starp diviem kadriem var būt patvaļīga garuma pauzes.

Mērķa sinhronizācija tiek veikta ar avota startbita palīdzību, turklāt avotam un mērķim jāstrādā ar vienādu frekvenci f_{sync} . Pēc avota startbita pārraidīšanas mērķa sinhroimpulsu ģenerators sāk veidot sinhronizācijas impulsus, kas ar speciālu shēmu palīdzību veic datu pārraidi.

Aparatūras līmenī datu apmaiņu realizē speciālu mikroshēmu kompleksdaudzfunkcionālais kontrolleris UART (*Universal Asynchronous Receiver/Transmitter*). Tas ir kopīgs nosaukums šādu mikroshēmu tipam. Visbiežāk izplatītie datu pārraides caur seriālo pieslēgvietu ir 8250. sērijas kontrolleri, kas satur 16450, 16550, 16650, 16750 tipa mikroshēmas.

RS-232 saskarne realizēta šādu signālu līniju veidā:

- SG (*Signal Ground*) – signāls „zeme”, attiecībā pret kuru tiek aprēķināti signālu līmeņi;
- TD (*Transmit Data*) – avota izeja (TXD);
- RD (*Receive Data*) – mērķa ieeja (RXD);
- RTS (*Request-To-Send*) – datu apmaiņas pieprasījums, ko iestata virknes pieslēgvietas kontrolleris UART;
- CTS (*Clear-To-Send*) – nosaka modema vai termināla gatavību datu apmaiņai;
- DTR (*Data Terminal Ready*) – nosaka seriālās pieslēgvietas kontrollera gatavību savienojumam;
- DSR (*Data Set Ready*) – norāda seriālās pieslēgvietas kontrollerim par ārējās ierīces gatavību savienojumam;
- DCD vai CD (*Data Carrier Detect*) – ieejas signāls no modema vai citas ierīces;
- RI (*Ring Indicator*) – izsaukuma indikatora ieeja.

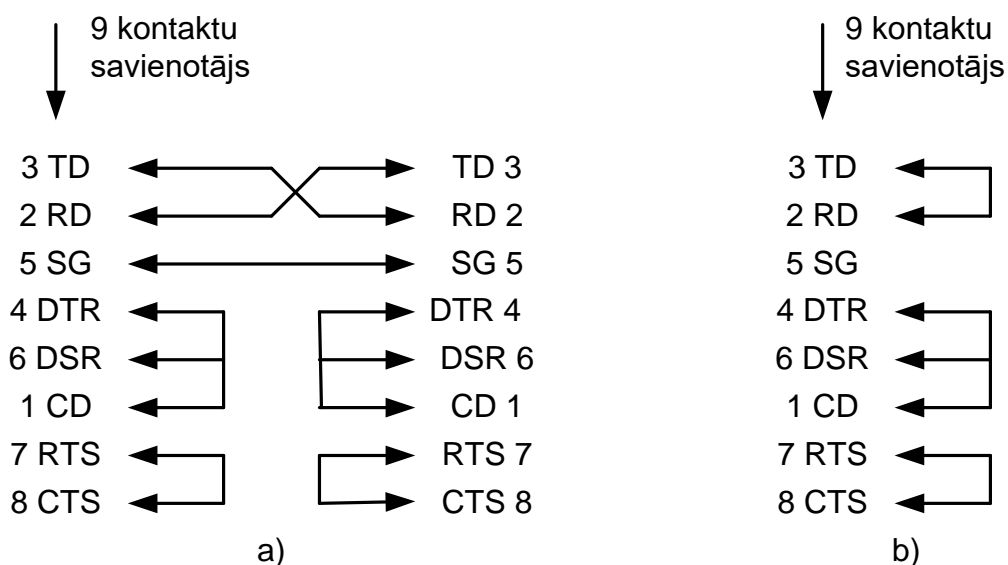
Signāli RD un TD, kas nosaka datu pārraidi, ir analizēti 10.2. attēla komentāros. Pēc būtības datu līnijas darbojas virknes režīmā, bet pārējās – paralēli. 10.2. tabulā doti 9 kontaktu savienotāja numuri un tiem atbilstošie signāli.

10.2. tabula

RS-232 saskarnes 9 kontaktu savienotāja numuri un signāli

Kontakta numurs	Signāls	Kontakta numurs	Signāls
1	CD	6	DSR
2	RD	7	RTS
3	TD	8	CTS
4	DTR	9	RI
5	SG		

Šie paši signāli figurē arī datora COM-pieslēgvietā, kurai tiek pieslēgta ārējā ierīce, piemēram, modems vai pele. Daudzos gadījumos pieslēdzamo ierīču testēšanai izmanto tā saucamo nullmodema savienojumu. Tādu vienkāršotu saskarni izmanto diagnostikas nolūkos, kā arī automātisko ierīču pievienošanai datoram. Nullmodema savienojuma shēma parādīta 10.3. a) attēlā.



10.3. att. Nullmodema savienojums (a) un apgrieztās saites saskarnes shēma (b)

Bieži sastopama situācija, kad programmētājam vajag pārbaudīt lietojumprogrammu funkcionēšanu, kas darbojas ar seriālo pieslēgvietu. Tādu programmu testēšanas nolūkā var izmantot tā saucamo apgrieztās saites saskarni (*loopback interface*), kuras shēma parādīta 10.3. b) attēlā.


RS-232 saskarnes signāli ļauj kontrolēt datu plūsmu atkarībā no avota un mērķa stāvokļa. Turklāt var tikt izmantoti vairāki datu apmaiņas protokoli. Uzreiz jāatzīmē, ka šie protokoli neskar iepriekš apskatītos datu pārraides pamata principus – tie ļauj sinhronizēt dažādas ātrdarbības ierīces.

Datu plūsmu vadībai var tikt izmantotas gan aparatūras, gan programmu metodes jeb protokoli. Biežāk izmantojamie ir šie:

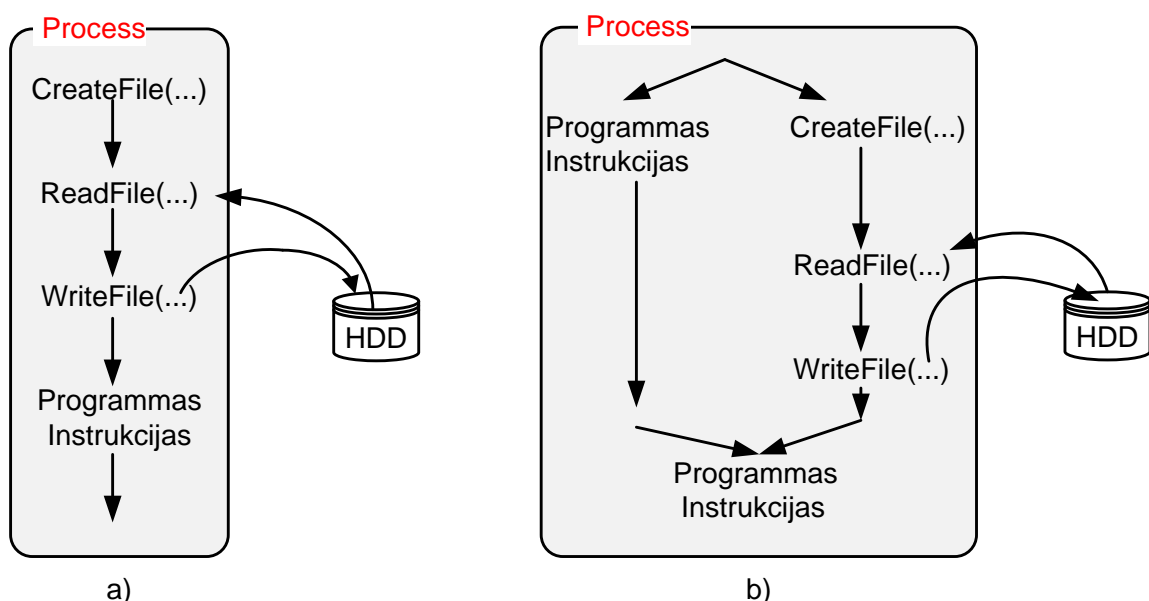
- aparatūras protokols RTS/CTS – izmanto printera pievienošanai vai datoru savienošanai (ja nevar izmantot nullmodema kabeli);
- aparatūras protokols DTR/DSR – analogisks iepriekšējam, bet izmanto citus signālus;
- programmu protokols XON/XOFF – izmanto abpusējos apmaiņas kanālos buferizētu datu apmaiņā. Ja piepildās mērķa buferis un datu pārraide tiek apturēta, tad tā atjaunojas tikai pēc datu apstrādes mērķa buferī. Mērķis aptur datu pārraidi ar komandas XOFF palīdzību un atsāk ar komandas XON palīdzību;
- programmu protokols ACK – viena datu baita vai baitu grupas pārraides sinhronizācija. Mērķis sūta avotam komandu ACK, bet avots pārraida baitu vai baitu grupu.

10.3. Seriālās pieslēgvietas programmēšanas aspekti

Datorsistēmās seriālā pieslēgvietā mijiedarbojas ar komunikāciju resursiem – fiziskām vai loģiskām ierīcēm, kas veic asinhronu datu apmaiņu. Datu apstrādes teorētiskie aspekti komunikāciju ierīcēs ir svarīgi ne tikai seriālās pieslēgvietas analīzē, bet ļauj izprast ievadizvades principus operētājsistēmās [9].

 Datu ievadizvade var būt sinhrona (bloķējoša) vai asinhrona (nebloķējoša). Reizēm asinhrono režīmu dēvē par ievadizvadi ar pārklāšanos (*overlapped*).


Bloķējošajā ievadizvadē datu apstrādes process gaida operācijas ar datiem beigšanos, bloķējot pārējās programmas koda daļas izpildi. Nebloķējošajā ievadizvadē datu apmaiņa noris paralēli pārējās programmas koda daļas izpildei (sk. 10.4. a) att.).



10.4. att. Datu apstrādes bloķējošais režīms (a) un nebloķējošais režīms (b)

Pieņem, ka procesam jānolasa dati no diska datnes, jāapstrādā tie un jāieraksta atpakaļ diskā. Ja tiek izmantots bloķējošais režīms, process gaidīs datu nolasīšanas no datnes beigas (API funkcija *ReadFile*), tad izmainītie dati tiks ierakstīti atpakaļ diskā (API funkcija *WriteFile*) – turklāt process tāpat gaidīs ierakstīšanas beigas. Ar tādu programmu struktūru ievadizvade samazina lietojumprogrammas veiktspēju.

Nebloķējošā (asinhronā) ievadizvade veic datu apmaiņu patvaļīgos laika momentos ar patvaļīga apjoma datu blokiem. Asinhronās datu apmaiņas mehānisms nodrošina paralēlu ievadizvades operāciju un cita programmas koda fragmenta izpildi, kas demonstrēts 10.4. b) attēlā. Datu apmaiņa ar ievadizvades ierīcēm tiek veikta nosacīti neatkarīgi no pārējās programmas daļas. Tiek ieviests jauns jēdziens – datu plūsma jeb straume.

 Plūsma (*stream*) – reālā laikā no viena punkta uz otru pārraidīta datu, audioinformācijas vai videoinformācijas plūsma. Straumēšanai nepieciešams ātri izveidot savienojumu un izmantot pietiekami ātrdarbīgu datoru, kas varētu izpildīt reālā laikā informācijas atspiešanas algoritmu.

Nebloķējošas datu apmaiņas praktiskās realizācijas paņēmieni operētājsistēmās ir šādi:

- vairāku straumējumu izmantošana, turklāt atsevišķi datu straumējumi var izmantot parasto bloķējošo ievadizvadi – netraucējot pārējiem;
- notikumu modeļa ar pārklāšanos izmantošana. Šajā gadījumā pēc datu ievadizvades sākuma straumējums turpina izpildi, bet, ja ir vajadzīgs I/O operācijas rezultāts, tas gaida notikuma iestāšanos, kurš signalizē par šīs operācijas beigšanos. Citiem vārdiem sakot, ievadizvades straumējumu sinhronizācija ar citiem straumējumiem tiek realizēta ar paziņojumu par notikumiem palīdzību (piemēram, ievades vai izvades beigas);
- nobeiguma procedūru izmantošana, kad pēc ievadizvades procedūras beigām tiek izveidots straumējums, kas izsauc beigu procedūru.



Visi asinhronās ievadizvades paņēmieni tādā vai citādā mērā izmanto straumējumus, jo programmas koda paralēla izpilde lietojumprogrammas līmenī nosaka straumējumu izmantošanu. Datu apmaiņa ar komunikāciju ierīcēm pārsvarā izmanto tieši asinhronos ievadizvades paņēmienus.

Komunikāciju ievadizvades izpratnes nolūkā tiek apskatīts asinhronās datu apmaiņas ar pārklāšanos piemērs, kad datu apmaiņa tiek veikta caur seriālo pieslēgvietu COM1.

Asinhrono operāciju programmēšanā izmanto jau zināmās WinAPI funkcijas *CreateFile*, *CloseHandle*, *ReadFile* un *WriteFile*, kas tiek pielietotas darbā ar datnēm. Komunikāciju ierīces operētājsistēmās tiek apstrādātas ar tādām pašām funkcijām kā parastās datnes. COM1 atvēršanai var izmantot funkciju *CreateFile*, kuras pirmais parametrs būs ierīces nosaukums: „COM1”. Pēc funkcijas izsaukšanas iegūto ierīces deskriptoru (*handle*) varēs izmantot turpmākajās operācijās ar datiem : *ReadFile*, *WriteFile*, *CloseHandle*.

Asinhronajai ievadizvadei ar pārklāšanos ir dažas īpatnības:

- ievadizvades operācijas netiek bloķētas, tātad funkcijas *ReadFile* un *WriteFile* atgriež vadību programmai nekavējoties – neatkarīgi no tā, vai ievadizvades operācija ir beigusies, vai nē;
- šo funkciju atgrieztā vērtība negarantē ievadizvades operāciju veiksmīgu vai neveiksmīgu beigšanos – šim nolūkam pastāv citas funkcijas, kas precīzāk interpretē ievadizvades operācijas rezultātus, piemēram, *HasOverlappedCompleted*;
- atgriežamā nosūtīto vai saņemto baitu skaita vērtība nav operācijas beigšanās pazīme;

- procesam jāparedz ievadizvades nodrošināšanu ar citu straumējumu palīdzību, turklāt jākontrolē, kura I/O operācija jau beigusies.

Pārklāšanās struktūra ir ļoti būtiska asinhronās datu apmaiņas operācijās un tās adrese tiek norādīta kā funkcijas *ReadFile* un *WriteFile* pēdējais arguments. Doto struktūru var aprakstīt šādā veidā (atbilstoši *Microsoft Platform SDK R2*):



10.1. piemērs. Pārklāšanās struktūras piemērs

```
typedef struct _OVERLAPPED
{
    ULONG_PTR Internal;
    ULONG_PTR InternalHigh;
    union
    {
        struct
        {
            DWORD Offset;
            DWORD OffsetHigh;
        };
        PVOID Pointer;
    };
    HANDLE hEvent;
} OVERLAPPED, *LPOVERLAPPED;
```

Komentāri

Lauku *Offset*, *OffsetHigh* un *hEvent* (pārējos laukus izmanto OS) nozīme ir šāda:

- lauks *Offset* norāda uz datnes sākuma pozīciju, no kurienes jāsāk datu ievadizvade. Šo lauku vajag inicializēt pirms funkciju *ReadFile* un *WriteFile* izmantošanas;
- lauks *OffsetHigh* ir vecākais vārds starta pozīcijā, no kura sākas ievadizvade. Parasti tā vērtība ir 0;
- lauks *hEvent* satur objekta „notikums” deskriptoru, kas signalizē par I/O operācijas beigšanos.

Tiek rekomendēts izmantot atsevišķas *OVERLAPPED* struktūras dažādām operācijām – tas saistīts ar to, ka atsevišķas operācijas var beigties dažādos laikos un vienas struktūras izmantošana visām operācijām var izsaukt grūti konstatējamas kļūdas.

Notikumi ir kodola objekti un nepieciešami paziņojumiem par kādas operācijas beigšanos.



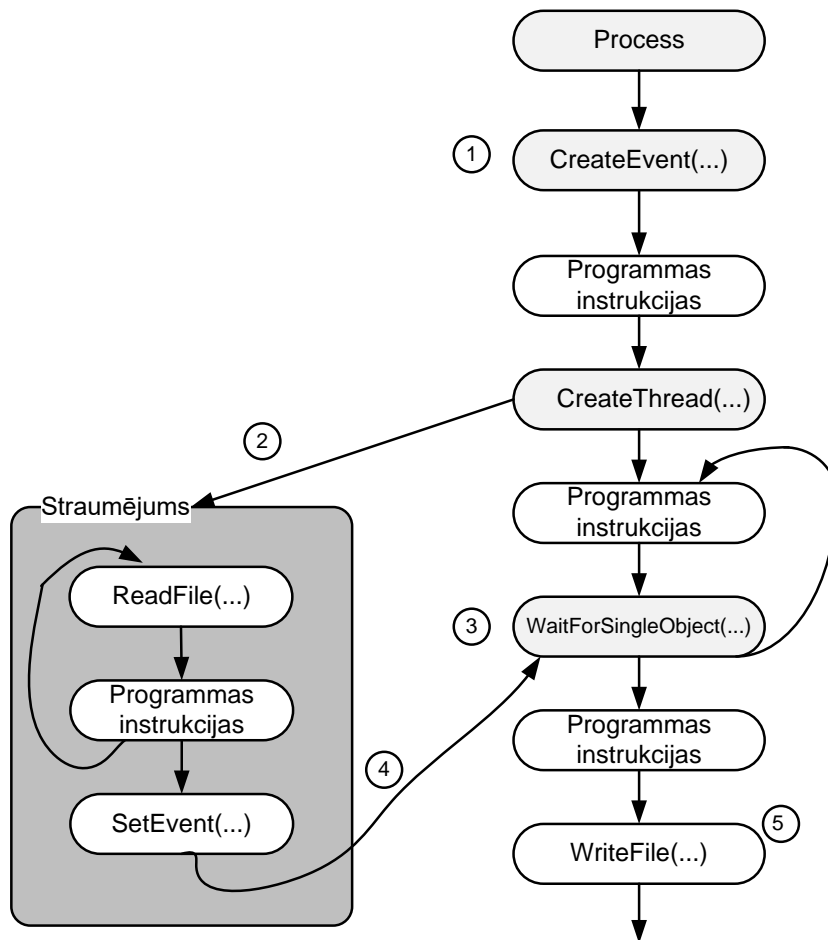
Objekts „notikums” var atrasties divos stāvokļos: signālstāvoklī (*signaled*) un nesignālstāvoklī (*nonsignaled*). Signālstāvoklis liecina par noteikta notikuma iestāšanos, nesignālstāvoklis ir notikuma iestāšanās gaidīšana.

Objektus „notikums” parasti izmanto situācijās, kad kādam straumējumam jāsaņem paziņojums no cita straumējuma par noteikta stāvokļa iestāšanos. Tā, piemēram, pamata straumējums izveido objektu „notikums” ar funkcijas *CreateEvent* palīdzību, kas nosaka tam nesignālstāvokli. Pēc tam straumējums turpina savu operāciju izpildi un to beigās iestata objektu „notikums” signālstāvoklī. Cits straumējums, kas gaidīja šī notikuma iestāšanos, ar vienas no funkcijām *WaitForSingleObject* vai *WaitForMultipleObjects* palīdzību aktivizējas un turpina apstrādi.

Objekta „notikums” funkcionēšanu ilustrē 10.5. attēlā parādītā asinhronā ievadizvades no datnes shēma.

Pieņem, ka programmai jāveic datu nolasīšana no datnes, jāapstrādā dati un jāieraksta atpakaļ datnē. Pieņem arī, ka programmai papildus jāveic citi uzdevumi. Tādā gadījumā datu nolasīšanas no datnes operāciju var veikt atsevišķā straumējumā, bet pēc to apstrādes paziņot pamata procesam par operācijas beigšanos. Šim nolūkam process izveido objektu „notikums” ar

WinAPI funkcijas *CreateEvent* palīdzību – ①, kurš signalizēs par operācijas beigām papildus izveidotajā straumējumā.



10.5. att. Notikumu izmantošana asinhronās ievadizvades programmā

Turpmāk ar funkcijas *CreateThread* process izveido straumējumu, kas veic datu nolasīšanu no datnes – ②. Datu nolasīšanas operācijas beigās tiek izsaukta funkcija *SetEvent*, kura nosaka objektam „notikums” signālstāvokli. Procesa pamata straumējums seko līdz „notikumam”, periodiski izsaucot vienu no tā saucamajām gaidīšanas funkcijām (*wait function*). Dotajā piemērā tā ir funkcija *WaitForSingleObject* – ③.

Šī funkcija saņem divus parametrus – pirmais ir objekta „notikums” deskriptors, kas tika iegūts, izsaucot funkciju *CreateEvent*, otrais – laika intervāls milisekundēs, kurā funkcija *WaitForSingleObject* gaida objekta „notikums” pāreju signālstāvoklī. No otrā parametra vērtības lielā mērā arī atkarīga programmas darbība. Ja laika intervālu uzdod 0, tad funkcija uzreiz nodod vadību pamata programmai; ja uzdod vērtību INFINITE, tad funkcija var gaidīt notikuma iestāšanos bezgalīgi ilgi (vismaz teorētiski!).

Funkciju *WaitForSingleObject* ērti izmantot ciklos – iestatot gaidīšanas intervālu 0. Ja piemērā objekts „notikums” pāriet signālstāvoklī – ④, tad ar funkcijas *WriteFile* palīdzību notiek datu ierakstīšana datnē – ⑤.

Tāds ir viens no iespējamajiem asinhronās datu apstrādes ar pārklāšanu scenārijiem. Var, protams, realizēt arī sarežģītākus asinhronās datu apstrādes algoritmus (semafori, gaidīšanas taimeri utt.). Vairāku objektu izmantošanā sinhronizācijas nolūkā var pielietot funkciju *WaitForMultipleObjects*.

Jāatzīmē, ka šāds augstāk minētais mehānisms tiek izmantots daudzās lietojumprogrammās, kas pieejamas Internetā, un var tikt izmantots datu apmaiņas demonstrējumos ar seriālās pieslēgvietas palīdzību.

Nākamajā piemērā ir demonstrēts seriālās pieslēgvietas konfigurācijas fragments.



10.2. piemērs. Seriālās pieslēgvietas konfigurēšana

```
#include "stdafx.h"
#include <windows.h>
#include <tchar.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    DCB dcb;
    HANDLE hCom;
    BOOL fSuccess;
    TCHAR *pcCommPort = TEXT("COM1");

    hCom = CreateFile( pcCommPort,
                     GENERIC_READ | GENERIC_WRITE,
                     0,
                     NULL,
                     OPEN_EXISTING,
                     0,
                     NULL
                    );

    if (hCom == INVALID_HANDLE_VALUE)
    {
        printf ("Porta kljuda %d.\n", GetLastError());
        getchar();
        return (1);
    }
    SecureZeroMemory(&dcb, sizeof(DCB)); //veido tekosho konfiguraciju
    dcb.DCBlength = sizeof(DCB);
    fSuccess = GetCommState(hCom, &dcb);

    if (!fSuccess)
    {
        printf ("COM1 stavoklja kljuda %d.\n", GetLastError());
        getchar();
        return (2);
    }
    dcb.BaudRate = CBR_57600;          // atrums
    dcb.ByteSize = 8;                  // datu biti
    dcb.Parity = NOPARITY;             // nav paaribas
    dcb.StopBits = ONESTOPBIT;        // 1 stopbits

    fSuccess = SetCommState(hCom, &dcb);

    if (!fSuccess)
    {
        printf ("COM1 stavoklja kljuda %d.\n", GetLastError());
        getchar();
        return (3);
    }
    _tprintf (TEXT("Serialaa pieslegvieta %s sekmigi nokonfigureta.\n"),
pcCommPort);
    getchar();
    return (0);
}
```

Rezultāts

Sekmīgas konfigurācijas gadījumā programma paziņo: „Seriālā pieslēgvietā COM1 sekmīgi nokonfigurēta”.

Nākamajā piemērā (C#) seriālajā pieslēgvietā tiek iesūtīta baitu virkne „Sveika, pasaule!”.



10.3. piemērs. Baitu virknes iesūtīšana seriālajā pieslēgvieta

```
using System;
using System.Collections.Generic;
using System.Text;
// SerialPort klase
using System.IO.Ports;

namespace ConsoleApplication3
{
    class Program
    {
        static void Main(string[] args)
        {
            // Serialaa porta iestadijumi
            SerialPort port = new SerialPort("COM1", 9600, Parity.None, 8,
StopBits.One);

            // atver portu
            port.Open();

            // raksta rindu
            port.Write("Sveika, pasaule !");

            // Ieraksta baitu rindu
            port.Write(new byte[] { 0x0A, 0xE2, 0xFF }, 0, 3);

            port.Close();
            Console.ReadLine();
        }
    }
}
```

Rezultāts

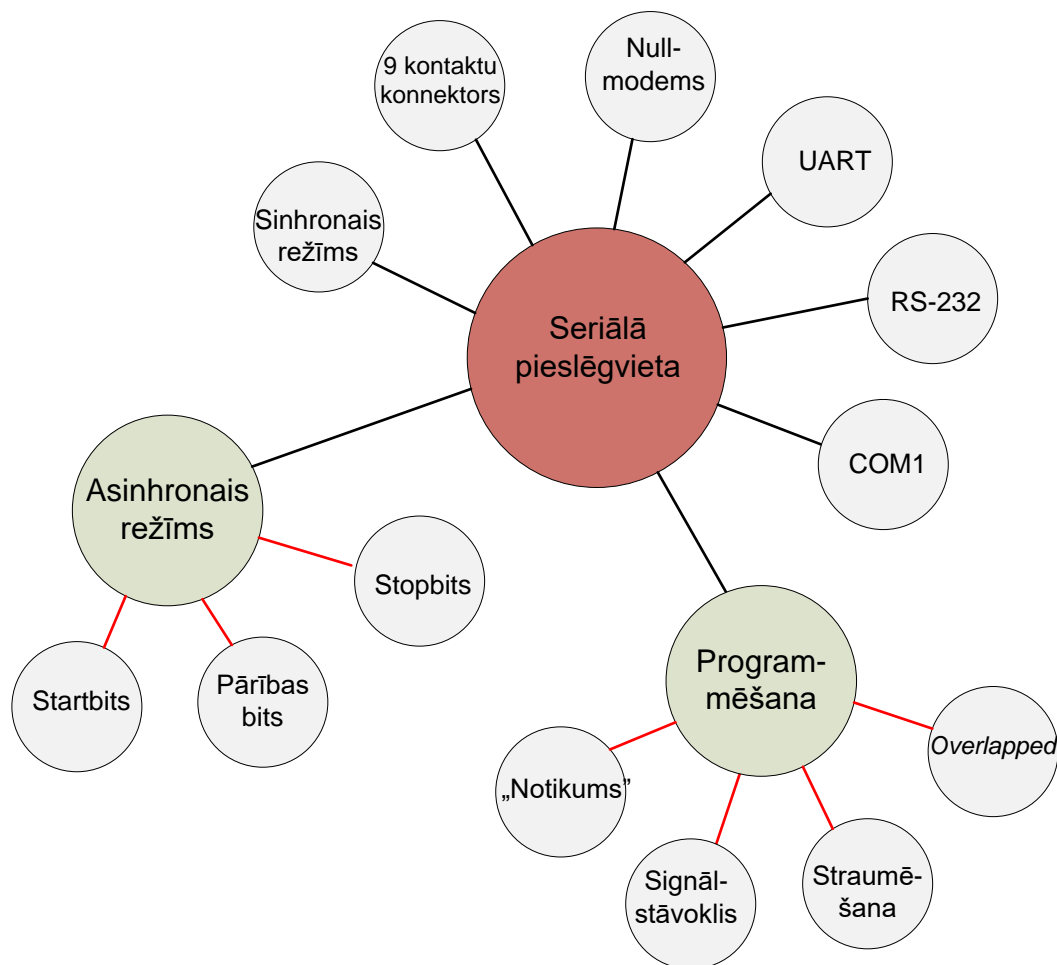
Diemžēl nav iespējas pārbaudīt programmas darbības rezultātu (šajā piemērā).

NODAĻAS KOPSAVILKUMS

- Seriālā pieslēgvieta ir ievadizvades pieslēgvieta, kas nodrošina informācijas pārsūtīšanu virknes formātā starp datoru un ārējām ierīcēm.
- Datu apmaiņa caur seriālo pieslēgvieta tiek realizēta sinhronajā vai asinhronajā režīmā. Sinhronajā režīmā datu avotam un datu mērķim jādarbojas vienā frekvencē. Asinhronajā režīmā pirms katra baita ir startbits, pēc tam seko datu biti, pārības bits un datu pārraides noslēgumā stopbits, kas garantē noteiktu laika aizturi starp datu sūtījumiem jeb kadriem.
- RS-232 ir ASV elektroniskās rūpniecības uzņēmumu apvienības izstrādāts saskarnes standarts, kas paredzēts datu pārraidei, izmantojot seriālās pieslēgvieta. Vairums personālo datoru ir apgādāti ar RS-232 saderīgām seriālajām pieslēgvieta, ko var izmantot modemu, printeru, skeneru un citu ārējo ierīču pievienošanai.
- Aparatūras līmenī datu apmaiņu realizē speciālu mikroshēmu kompleksdaudzfunkcionālais kontrolleris UART.
- Datu ievadizvade var būt sinhrona (bloķējoša) vai asinhrona (nebloķējoša). Asinhrono režīmu dēvē par ievadizvadi ar pārklāšanos).
- Straumējums - reālā laikā no viena punkta uz otru pārraidīta datu, audioinformācijas vai videoinformācijas plūsma.
- Visi asinhronās ievadizvades paņēmieni tādā vai citādā mērā izmanto straumējumus, jo programmas koda paralēla izpilde lietojumprogrammas līmenī nosaka straumējumu izmantošanu. Datu apmaiņa ar komunikāciju ierīcēm pārsvarā izmanto tieši asinhronos ievadizvades paņēmienus.

- Objekts „notikums” var atrasties divos stāvokļos: signālstāvoklī un nesignālstāvoklī. Signālstāvoklis liecina par noteikta notikuma iestāšanos, nesignālstāvoklis ir notikuma iestāšanās gaidīšana.

10.6.attēlā parādīts nodaļā minēto svarīgāko jēdzienu koks.




10.6. att. Desmitās nodaļas galveno jēdzienu koks


Uzdevumi un jautājumi patstāvīgajam darbam

1. Kāpēc sinhronajā režīmā datu apmaiņa notiek ātrāk nekā asinhronajā?
2. Kāda nozīme ir vairākiem stopbitiem datu pārraides seansā (piem., 7-E-2)?
3. Kā zināms, pārības bits datu pārraidē var arī nebūt. Vai varētu iztikt bez startbita vai stopbita?
4. Kas ir asinhronās datu apmaiņas iniciators?
5. Raksturot asinhronās datu apmaiņas būtību datu pārsūtīšanas operācijās.
6. Dažkārt ir sastopams seriālās pieslēgvietas 25 kontaktu savienotājs. Kādas varētu būt atlikušo 16 (25-9) kontaktu funkcijas?
7. Vai ir lietderīgi pieslēgt drukas iekārtu seriālajai pieslēgvietai?
8. Paskaidrot nullmodema savienojuma būtību.
9. Kāda nozīme ir apgrieztās saites saskarnei?
10. Raksturot seriālās pieslēgvietas programmēšanas problēmas.

11. USB KOPNES ARHITEKTŪRA


Nodaļa veltīta USB arhitektūras apskatam. USB kopne izstrādāta ar mērķi kļūt par datoru arhitektūras standartu un paredzēta ārējo ierīču un datorsistēmu saskarņu izstrādāšanai.

 USB tika iepļānota kā universāls standarts, kas ļautu izmantot unificētu metodi ārējo ierīču pieslēgšanai, nomainot seriālās un paralēlās saskarnes [3,4,5,9].

-  USB kopnei piemīt vesela virkne priekšrocību salīdzinājumā ar seriālo un paralēlo saskarni:
- vienkāršojas OS ārējo ierīču aparatūras un programmu konfigurēšana uz viena unificēta standarta izmantošanas rēķina;
 - vienkāršojas sistēmas aparatūras konfigurācija, jo mikroshēmos vairs nav nepieciešamības iekļaut mazražīgās seriālās un paralēlās saskarnes moduļus;
 - lietotājiem vieglāk uzstādīt, konfigurēt un izmantot ierīces;
 - parādās jaunas iespējas datora aparatūras līdzekļu paplašināšanai;
 - būtiski pieaug datu apmaiņas ātrums (līdz 12 Mb/s), kas nodrošina iespēju multivides (skaņa, grafika un video) datu pārraidi reālā laika režīmā;
 - no aparatūras realizācijas viedokļa saskarne ir vienkārša, kas nodrošina jaunu ārējo ierīču kļāšu izstrādāšanu.


USB saskarne sastāv no hostdatora kontrollera (*host-controller*) un vairākām ierīcēm, kas savienotas ziedlapķēdē (*daisy-chained*). Šajā ķēdē var iekļaut papildus USB ierīces, kas var izveidot kokveida struktūru ar maksimāli 5 iekļaušanas līmeņiem vienam kontrollerim. USB kopnes topoloģija parādīta 11.1. attēlā.

Hostdatora kontrolleris tiek pieslēgts PCI kopnei pēc standarta shēmas. OS mijiedarbojas ar hostdatoru ar I/O pieslēgvietu palīdzību vai ar atmiņā attēlojamajiem reģistriem. Pats hostdatora kontrolleris vada hierarhisko struktūru, kas sastāv no USB ierīcēm, kā parādīts 11.1. attēlā. Šajā hierarhijā ir speciāli elementi – koncentratori jeb centrmezgli (*hubs*), kas izpilda mezglu funkcijas (*endpoints*), pie kuriem savukārt tiek pieslēgtas citas USB ierīces.

 USB ierīces (peles, printeri, skeneri utt.) USB terminoloģijā tiek sauktas par funkcijām. USB hierarhijā iekļauts tikai viens saknes centrmezgls (*root hub*), uz kura pamata tiek veidota ziedlapķēde.

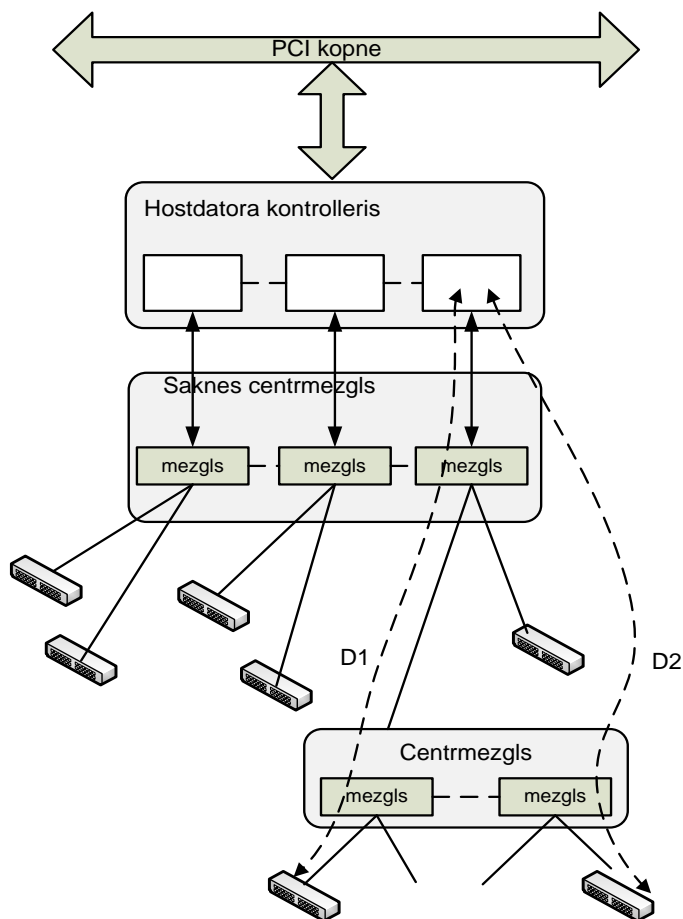
11.1. attēlā parādītas divas ziedlapķēdes (D1 un D2), kas sastāv no diviem posmiem: saknes centrmezgla un parastā centrmezgla. Hostdatora kontrolleris un saknes centrmezgls veido vienu loģisko ierīci, lai gan fiziski tie var tikt realizēti atsevišķās ierīcēs. Piemēram, hostdatora kontrolleris 11.1. attēlā satur trīs fiziskas ierīces – katra no tām vada vienu no saknes centrmezgla fiziskajiem kanāliem.

Tā kā hostdatora kontrolleris satur vairākas fiziskas ierīces, tas ļauj veikt datu no dažādām USB ierīcēm paralēlu apstrādi, jo katrai no tām izdalīts savs atmiņas diapazons un pārtraukumu līnija. Tādējādi ierīces draiveris pārtraukumu procedūrās var veikt datu asinhronu apstrādi.

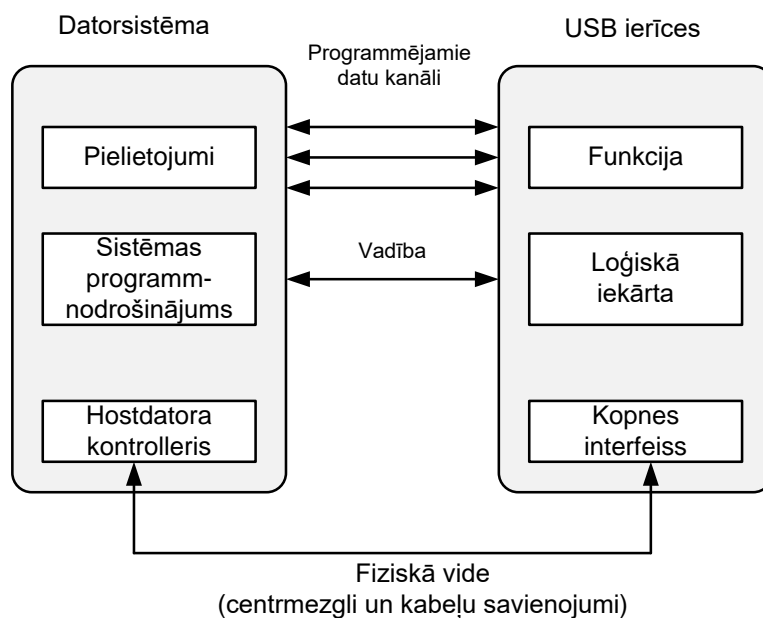
 Pie viena hostdatora kontrollera pieslēdzamo ierīču skaits nedrīkst pārsniegt 127, turklāt USB ierīču savienojuma kabeļi nav terminēti (atšķirībā no citiem saskarnēm, piemēram, SCSI, kur vairāku ierīču pieslēgšanai nepieciešami gala terminatori).

Viszemākajā modeļa līmenī mijiedarbība tiek veikta starp hostdatora kontrolleri, kas pieslēgts PCI kopnei, un fiziskās ierīces kopnes saskarni. Otrajā līmenī mijiedarbojas sistēmas programnodrošinājums un loģiskā ierīce. Visbeidzot, trešajā līmenī notiek datu apmaiņa starp klienta lietojumprogrammām un dotās ierīces funkciju.

USB ierīču mijiedarbības modelis ar OS un lietojumprogrammām parādīts 11.2. attēlā.



11.1. att. USB kopnes hierarhija



11.2. att. USB informatīvais modelis

Faktiski visa informatīvā plūsma notiek caur fizisko kanālu pašā zemākajā līmenī, tāpēc informatīvo plūsmu attēlojums augšējos divos līmeņos parādīts apmaiņas specifikas labākai izpratnei.

OS *Windows* vidē zemākā līmeņa informatīvais modelis pārstāvēts ar ierīču draiveru grupu. Šajā grupā ietilpst hostdatora kontrolleris (parasti tas ir *usbuhci.sys*), saknes centrmezgla draiveris

(*usbhub.sys*) un dinamiskā bibliotēka (*usbui.dll*), kuru izmanto sistēma un klienta draiveri (*usbd.sys*). Kopā šie draiveri vada aparatūras savienojumus, dodot iespēju realizēt programmu kanālus mijiedarbībai daudz augstākos līmeņos.



USB standarts nosaka 4 tipu datu pārraidi:

- *Control* – vadības signālu pārraide un saņemšana (tiek izmantoti pievienojamās ierīces konfigurēšanai). Šis tips garantē datu apmaiņu bez zaudējumiem, turklāt datu apjoms var būt mazāks vai vienāds ar 8, 16, 32 vai 64 baitiem;
- *Bulk* – nelielu nestrukturētu datu pakešu pārraide un saņemšana. Šis tips garantē apmaiņu bez zaudējumiem, turklāt datu apjoms var būt mazāks vai vienāds ar 8, 16, 32 vai 64 baitiem. Pārsvarā tiek realizēts datu apmaiņai ar printeriem un skeneriem;
- *Interrupt* – noteiktu simbolu informācijas pārraide, kas ļauj saņemt zināmu atbildes reakciju. Šis tips garantē datu apmaiņu bez zaudējumiem, turklāt datu apjoms var būt mazāks vai vienāds ar 64 baitiem;
- *Isochronous* – liela apjoma nestrukturētu datu pārraide vai saņemšana ar iepriekš iestatītu noteiktu periodiskumu. Šis tips negarantē to, ka nebūs zaudējumu datu pārraidē. Datu apjoms var būt mazāks vai vienāds ar 1023 baitiem (šī datu tipa piemērs ir audio informācija).

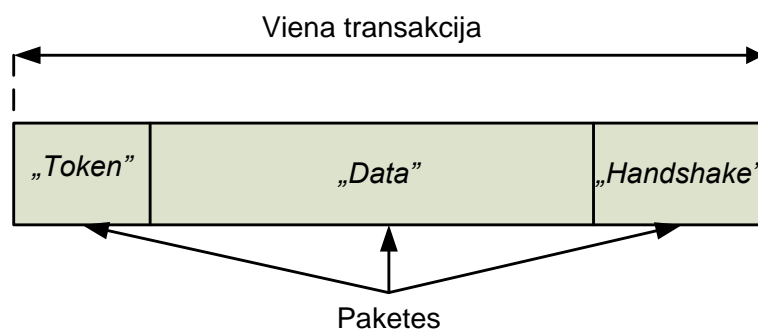
Katrs datu pārraides kanāls darbojas tikai ar vienu no uzrādītajiem datu tiptiem.

Kad lietojumprogramma sūta vai saņem datus caur USB programmu kanālu, tiek izsaukta atbilstoša WinAPI funkcija, kas ar OS I/O pārvaldnieka palīdzību vēršas pie ierīces draivera. I/O pārvaldnieks veido pieprasījuma paketi IRP ievadizvadei (*I/O Request Packet*) un nosūta to draiverim. Draivera pamatfunkcija ir datu nodošana USB ierīcei caur programmu kanālu, turklāt viena tipa dati tiek pārraidīti kā transakcijas.

Informācija mikrokadru (*microframes*) veidā ik pēc 125 mks tiek pārraidīta pa USB 2.0 kopni (vai ik pēc milisekundes priekš USB 1.0). Katra transakcija savukārt tiek sadalīta fāzēs (vienā vai vairākās). Katra fāze var tikt uzdots ar vienu no paketēm:

- *Token* – marķiera pakete (vai vienkārši marķieris) tiek pārraidīta no hostdatora kontrollera visām kopnē nokonfigurētajām ierīcēm. Marķieris ietver ierīces adresi un daudzos gadījumos arī mezgla numuru. Tikai ierīce, kas atpazīs savu adresi, turpinās datu apmaiņu;
- *Data* – datu pakete tiek pārraidīta no hostdatora kontrollera ierīcei (datu ierakstīšana) vai tiek saņemta (datu nolasišana);
- *Handshake* – pakete, kurā ierakstās statusa informācija par apmaiņas stāvokli. To izvieto USB kopnē vai nu hostdatora kontrolleris, vai ierīce. Piemēram, sekmīgas informācijas saņemšanas gadījumā ierīce izvieto kopnē paketi *ACK*. Ja ierīce aizņemta, tiek izvietota pakete *NAK*. Ja dati veiksmīgi pieņemti, bet kaut kādu iemeslu dēļ traucēta apmaiņas loģika, tad iekārta iestata paketi *STALL*.

Vienas transakcijas shēmas piemērs datu apmaiņā parādīts 11.3. attēlā.



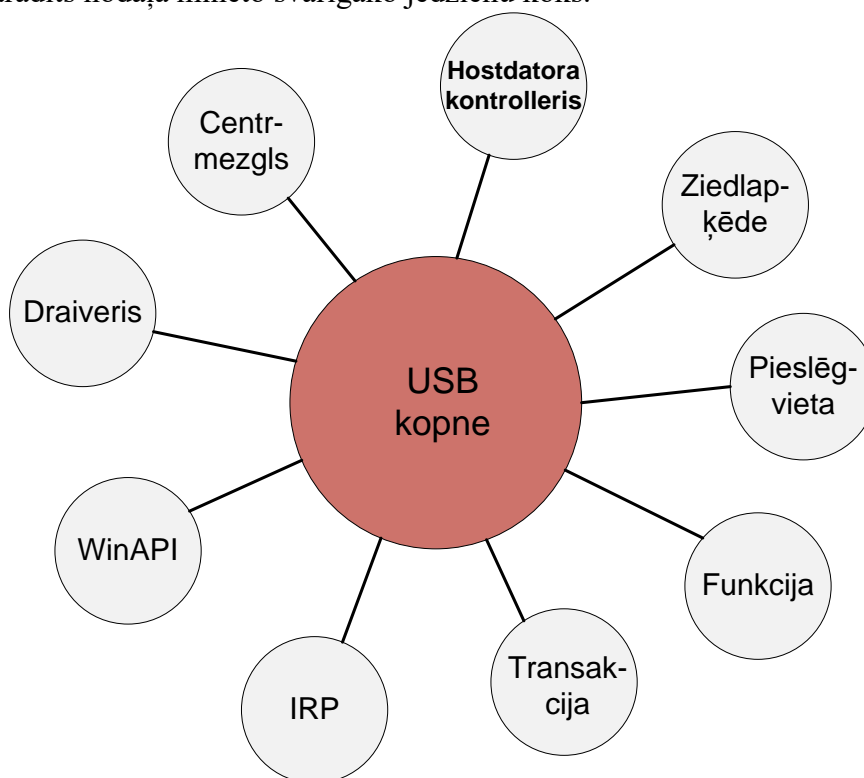
11.3. att. Vienas transakcijas piemērs

Paketē *handshake* netiek detalizēti aprakstīts kļūdas raksturs kopnē datu pārraides laikā. Izrādās, ka hostdatora kontrolleris vai USB ierīce neanalizē kļūdas – ja tādas ir, tad transakcija tiek veikta atkārtoti.

NODAĻAS KOPSAVILKUMS

- USB iepļānota kā universāls standarts, kas ļautu izmantot unificētu metodi ārējo ierīču pieslēgšanai, nomainot seriālās un paralēlās saskarnes.
- USB kopnei piemīt vesela virkne priekšrocību salīdzinājumā ar seriālo un paralēlo saskarni.
- USB saskarnei piemīt asimetriska loģika, jo USB sastāv no hostdatora kontrollera un vairākām ierīcēm, kas savienotas ziedlapķēdē.
- Tā kā hostdatora kontrolleris satur vairākas fiziskas ierīces, tas ļauj veikt datu no dažādām USB ierīcēm paralēlu apstrādi, jo katrai no tām izdalīts savs atmiņas diapazons un pārtraukumu līnija.
- Pie viena hostdatora kontrollera pieslēdzamo ierīču skaits nedrīkst pārsniegt 127.
- Kad lietojumprogramma sūta vai saņem datus caur USB programmu kanālu, tiek izsaukta atbilstoša WinAPI funkcija, kas vēršas pie ierīces draivera.
- Draivera pamatfunkcija ir datu nodošana USB ierīcei caur programmu kanālu, turklāt viena tipa dati tiek pārraidīti kā transakcijas.

11.4. attēlā parādīts nodaļā minēto svarīgāko jēdzienu koks.



11.4. att. Vienpadsmitajā nodaļā minēto terminu koks




Uzdevumi un jautājumi patstāvīgam darbam

1. Uzzīmēt iespējamo USB kopnes hierarhiju ar 3 līmeņiem.
2. Kādā veidā tiek realizēta daudzo pieslēgto USB ierīču paralēlā darbība?
3. Cik informācijas mikrokadru var pārraidīt pa USB 2.0 kopni 29 sekunžu laikā?
4. Paskaidrot transakcijas paketes marķiera funkcijas.
5. Iztirzāt transakciju rokspiešanas principa darbību.
6. Kā lietojumprogrammas nodod datus USB ierīcei?

12. DATORSISTĒMU VEIKTSPĒJAS ANALĪZE

Projektētājiem ir svarīgi novērtēt datorsistēmu veiktspēju. Šādu vērtējumu iegūšanai ir divas vispārpieņemtas metodes: imitācijas modelēšana un modeļa analītiskā izpēte. Modelēšanas rezultātā var tikt iegūts kāds vispārīgs veiktspējas raksturlielums, bet dažādu arhitektūras parametru rādītāji dažkārt paliek slēpti. Analītiskie modeļi savukārt satur atsevišķu arhitektūras komponentu raksturlielumu analīzi. Nodaļā tiks apskatīti daži datorsistēmu veiktspējas analītisko modeļu teorētiskie aspekti, jo tie ļauj novērtēt atsevišķu parametru ietekmi uz datorsistēmas veiktspēju kopumā.

 Veiktspēja (*performance*) - sistēmas vai tās komponentu spēja izpildīt paredzētās funkcijas. Kā veiktspējas kvantitatīvie kritēriji parasti ir atbildes laiks, caurlaidspēja, izmantojamība u.c. [1,4,7].

Agrīnais priekšstats

Skaitļotāju arhitektūras izstrādē liela vērība tiek veltīta tās veiktspējas problēmai. Faktiski to ir grūti paredzēt un izmērīt, bieži šis jēdziens tiek nepareizi interpretēts. Galvenais iemesls ir tas, ka veiktspēju var analizēt makrolīmenī un mikrolīmenī. Tomēr lielākā izstrādātāju daļa koncentrējas tikai uz mikrolīmeni. Par to liecina tādu veiktspējas novērtēšanas rādītāju izmantošana kā „*miljons operāciju skaits sekundē*”, „*saskaitīšanas vai reizināšanas operāciju izpildes laiks*” u.c. Piemēram, novērtējumam „*operāciju skaits sekundē*” ir jēga tikai gadījumā, kad tiek salīdzinātas vienas un tās pašas arhitektūras divas realizācijas.

Termins „*veiktspēja*” tiek traktēts nepareizi arī tāpēc, ka to bieži attiecina uz programmu izpildes ātrumu. Korektāk būtu to pielīdzināt uzdevumu risināšanas ražīgumam jeb produktivitātei (*productivity*).



12.1. piemērs

Sistēma A izpilda programmu 1 minūtē, bet nepieciešamas 10 cilvēkdienas programmēšanai. Sistēma B izpilda to pašu programmu 2 minūtēs, bet nepieciešamas 2 cilvēkdienas programmēšanai. Kāda sistēma ir produktīvāka?

Risinājums

Mikrolīmenī sistēma A ir divas reizes ātrāka par B. Makrolīmenī sistēmas A produktivitāte ir piecas reizes mazāka par B.

Iepriekš teiktais nenozīmē to, ka programmu izpildes laikam nav būtiskas nozīmes. Svarīgi ir saprast, ka programmu ātrdarbība atkarīga no tehnoloģiskās bāzes (ietekmē datu apstrādes ātrumu) un skaitļotāja arhitektūras (ietekmē izpildāmo darbu apjomu). Teorētiski ir iespējams skaitļotājs tikai ar 3 komandām:

- vārda palielināšana par 1;
- vārda samazināšana par 1;
- pāreja, ja vārda vērtība ir 0.

Nav grūti pamanīt, ka kopējais darbu apjoms (izlase, dekodēšana un komandu izpilde) vairāku reizināšanas operācijas realizēšanas gadījumā būs ievērojami lielāks tajā skaitļotājā, kuram komandu kopā nebūs iekļauta reizināšanas operācija.

Arhitektūru efektivitātes novērtējumu pēc programmu izpildes ātruma (ja, piemēram, ir jāsalīdzina hipotētiskā sistēma A un esoša sistēma B) nevar veikt šādu iemeslu dēļ:

- var izstrādāt sistēmas A imitāciju modeli, bet nav skaidrs, kādus apstrādes datus izmantot modelēšanā, un nav informācijas par sistēmas A aparatūras realizācijas īpatnībām;

- pat ja tiek realizēta arhitektūra A, nav skaidra abu arhitektūru salīdzināšanas iespējamība pēc tādiem kritērijiem kā elektronisko shēmu daudzums, to ātrdarbība un izmaksas. Vienmēr paliek neskaidrs jautājums, vai abi realizācijas gadījumi ir optimāli attiecībā pret arhitektūru.

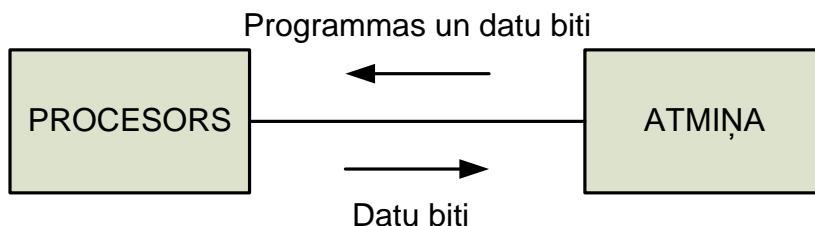
Acīmredzot ir nepieciešams vienkāršs un no aparatūras realizācijas neatkarīgs divu dažādu arhitektūru salīdzināšanas kritērijs pēc informācijas apstrādes efektivitātes.

Analīze var tikt veikta, izmantojot šādus parametrus (sk. modeli 12.1. att.):

S – programmas izmēru;

M – pārsūtāmo bitu skaitu starp procesoru un atmiņu (reģistru līmenī) izpildes laikā;

R – pārsūtāmo bitu skaitu starp procesora reģistriem programmas izpildes laikā.



12.1. att. Datu apmaiņas modelis starp procesoru un atmiņu

Parametrs S ir programmas izmēra skaitliskais novērtējums, kas atkarīgs no komandu garuma, datu izvietošanas atmiņas apjoma utt.

Parametrs M nosaka pārsūtāmo informācijas bitu daudzumu starp procesoru un atmiņu. Šis parametrs atkarīgs no kopnes starp procesoru un atmiņu caurlaidspējas (*bandwidth*). No šāda aspekta parametrs M pietiekami labi raksturo dotās arhitektūras efektivitāti.

Ja, piemēram, arhitektūrā A kādas programmas izpildei nepieciešams pārsūtīt 6×10^6 bitu informāciju, bet arhitektūrā B – 2×10^6 bitu informāciju, tad pirmajā tuvinājumā var teikt, ka arhitektūra B nodrošina $3 \times$ lielāku efektivitāti nekā arhitektūra A.

Parametrs R raksturo tos rādītājus, ko neņem vērā M – datu apmaiņas apjomu procesora iekšienē. Acīmredzams, ka šis parametrs būtiski atkarīgs no procesora aparatūras realizācijas (procesora funkcijas, ALU utt.). Dažādu arhitektūru salīdzināšanā parametru R bieži neizmanto šādu iemeslu dēļ:

- parametrs R būtiski atkarīgs no aparatūras realizācijas;
- testētāji patvaļīgi traktē šī parametra rādītājus;
- tajos pašos testos ir parādīts, ka M ir svarīgāks raksturlielums.

Dažādu arhitektūru salīdzinošā analīze parāda pietiekami augstu korelāciju starp S, M un R (skat 12.1. tabulu).

12.1. tabula

Parametru S, M un R mērījumi 12 testu programmām

Arhitektūra	S	M	R
<i>PDP-11</i>	1,00	0,93	0,94
<i>System 370</i>	1,21	1,27	1,29
<i>Interdata 8/32</i>	0,83	0,85	0,83
<i>IBM PC</i>	nav datu	nav datu	nav datu

Tādējādi jau skaitļošanas tehnikas ēras sākumā arhitektūras veiktspējas izvērtēšanai tika pievērsta liela uzmanība [10].

Mūsdienu priekšstati

1) Veiktspējas modeļi

Lai projektējams novērtētu datorsistēmas vai atsevišķu apakšsistēmu veiktspēju dažādu projekta variantu izvērtēšanā, būtiski ir divi rādītāji:

- uzdevuma izpildes laiks;
- uzdevuma izpildes ātrums.

CPI jēdziena izmantošana

Galvenais laika raksturlielums, ko pielieto procesoru novērtēšanā, ir taktu skaits komandā jeb CPI (*Cycles Per Instruction*). Šī vērtība nosaka laiku taktīs, ko procesors patērē vidējās komandas izpildei:

$$CPI = \frac{\text{Taktu skaits}}{\text{Izpildīto komandu skaits}} \quad (1)$$

CPI apgrieztā vērtība – komandu skaits taktī jeb IPC (*Instructions Per Cycles*) nosaka komandu izpildes ātrumu un atsevišķos gadījumos kalpo par ērtu veiktspējas rādītāju.

CPI izskaitļošanas piemērs: uzdevumam, kam nepieciešamas 1×10^6 taktis 5×10^5 komandu izpildei, CPI ir 2,0. Mazākas CPI vērtības norāda uz lielāku veiktspēju nekā lielākas. Daudziem procesoriem dažādām komandām nepieciešams dažāds taktu skaits, tāpēc CPI ir vidējais raksturlielums pa visām izpildītajām komandām.

CPI tiek izmantots arī projektu modifikācijas lietderības novērtēšanā. Pieņem, ka projekts A tiek modificēts projektā B. Vai B būs labāks par A? Atbildi var iegūt, izmantojot ātrdarbības palielināšanas jēdzienu (*speedup*):

$$\text{Ātrdarbības palielinājums} = \frac{\text{Laiks A}}{\text{Laiks B}} = \frac{CPI A}{CPI B} = \frac{\text{Ātrums B}}{\text{Ātrums A}} = \frac{IPC B}{IPC A} \quad (2)$$

Ja projekts B ir labāks par A, ātrdarbības palielinājums būs >1 . Ja projekts B samazina ātrdarbību, tad ātrdarbības palielinājums būs <1 . Ja ātrdarbības palielinājums ir vienāds ar 1, tad veiktspēja nav mainījusies.

Līdzīgi tiek rēķināta divu sistēmu veiktspējas starpība procentos. Pieņemot, ka sistēma A ir ātrāka par sistēmu B un ka A laiks ir mazāks par B, var izmantot formulas:

$$\begin{aligned} \text{A ir ātrāks nekā B par } x\% (\text{ātruma ziņā}) &= 100x \frac{\text{Ātrums A} - \text{Ātrums B}}{\text{Ātrums B}} = 100 \left(\frac{\text{Ātrums A}}{\text{Ātrums B}} - 1 \right) \\ &= 100(\text{Ātrdarbības palielinājums} - 1) \end{aligned} \quad (3)$$

$$\begin{aligned} \text{A ir ātrāks nekā B par } x\% (\text{laika ziņā}) &= 100x \frac{\text{Laiks B} - \text{Laiks A}}{\text{Laiks B}} = 100 \left(1 - \frac{\text{Laiks A}}{\text{Laiks B}} \right) \\ &= 100 \left(1 - \frac{1}{\text{Ātrdarbības palielinājums}} \right) \end{aligned} \quad (4)$$



12.2. piemērs

Procesors A izpilda doto uzdevumu 1 sekundes laikā, procesors B – 2 sekundēs. Par cik % A ir ātrāks nekā B?

Risinājums

Procesora A ātrdarbības palielinājums attiecībā pret B ir $2/1=2$. Tātad procesors A ir par $100(1-1/2)=50\%$ ātrāks nekā B.



12.3. piemērs

Procesors A izpilda doto uzdevumu ar ātrumu 1 uzdevums sekundē, procesors B – 0,5 uzdevumi sekundē. Par cik % A ir ātrāks nekā B?

Risinājums

Ātrdarbības palielinājums sastāda $1/0,5=2$. Tātad procesors A ir par $100x(2-1)=100\%$ ātrāks nekā B.



12.4. piemērs

Procesors izpilda 100M operācijas sekundē ($M=10^6$). Uzlabotas atmiņas izmantošana dod ātruma pieaugumu līdz 125M operācijām sekundē. Kāds procentuāli ir veikspējas pieaugums?

Risinājums

Ātrdarbības palielinājums sastāda $125M/100M=1,25$. Tātad veikspējas pieaugums procentos ir $100x(1,25-1)=25\%$.



12.5. piemērs

Tam pašam procesoram nepieciešamas $1x10^{-8}$ sekundes operācijai. Pēc atmiņas modifikācijas šis lielums uzlabojas līdz $8x10^{-9}$. Kāds procentuāli ir veikspējas pieaugums?

Risinājums

Ātrdarbības palielinājums sastāda $1x10^{-8}/(8x10^{-9})=1,25$. Tātad veikspējas pieaugums procentos ir $100x(1,25-1)=25\%$.

Vidējo vērtību izmantošana

Loģiska ir vēlme izteikt datorsistēmas veikspēju ar vienu skaitli. Attiecībā uz datorsistēmām var veikt triju veidu mērījumus:

- laiks, kas nepieciešams uzdevuma izpildei;
- ātrums, ar kādu tiek izpildīts uzdevums;
- ātruma vai laika attiecība veikspējas novērtēšanai.

Aritmētisko vidējo laika mērījumos izmanto vidējās vērtības izskaitļošanai mainīgajiem ar vienādiem svāriem (*weights*). Aritmētisko vidējo vienam notikumam aprēķina pēc formulas:

$$\text{Aritmētiskais vidējais} = \frac{1}{n} \sum_{i=1}^n T_i \quad (5)$$



12.6. piemērs

Katra mēneša pirmajā datumā ar datoru tiek veikts uzdevums. Šī uzdevuma izpildes laiks 4 mēnešos ir: 2 stundas, 2,2 stundas, 1,9 stundas un 2,3 stundas. Kāds ir vidējais uzdevuma izpildes laiks?

Risinājums

Visiem datiem ir vienāds svārs, jo katru mēnesi tiek izpildīts viens un tas pats uzdevums. Tāpēc vidējās vērtības aprēķināšanai izmanto aritmētiskā vidējā vērtības formulu (5): $(2,0 + 2,2 + 1,9 + 2,3)/4=2,1$ stundas uzdevuma veikšanai.

Daudzos gadījumos aritmētiskā vidējā ar vienādiem svāriem pielietošana izraisa kļūdas. Jānodala gadījumi, kad atsevišķi notikumi izpildās ar dažādām varbūtībām un dažādos laika intervālos. Viena notikuma svērtais laiks arī ir svērtais aritmētiskais vidējais, kuru aprēķina pēc formulas:

$$\text{Svērtais aritmētiskais vidējais} = \frac{1}{n} \sum_{i=1}^n W_i T_i \quad (6)$$



12.7. piemērs

Procesora komandu kopa satur divas komandu klases: klases A komandām izpildei nepieciešamas 2 taktis, bet klases B komandām – 3 taktis. No visām izpildītajām komandām 75% pieder klasei A un 25% - klasei B. Kāda ir procesora CPI vērtība?

Risinājums

Komandas tips	Svari	Taktis	Reizinājums
A	0,75	2	1,50
B	0,25	3	0,75

Procesora CPI aprēķina kā svērto aritmētisko vidējo: $CPI = W_A CPI_A = W_B CPI_B$.
 $CPI = 0,75 \times 2 + 0,25 \times 3 = 2,25$ taktis uz komandu.

Veiktspēju bieži mēra ātruma vienībās. Piemēram, dators var izpildīt 0,5 IPC komandas takts laikā. Ja apstrādājami notikumi ir dažādi ātrumi, tad vidējās vērtības tiek iegūtas ar harmoniskā vidējā palīdzību:

$$\text{Harmoniskais vidējais} = \frac{1}{\frac{1}{n} \sum_{i=1}^n \frac{1}{R_i}} = \frac{n}{\sum_{i=1}^n \frac{1}{R_i}} \quad (7)$$



12.8. piemērs

Tiek turpināti 12.6. piemērā aprakstītā datora novērojumi. Pieņem, ka ir iegūti fakti: 0,5, 0,45, 0,53 un 0,43 uzdevumi/stundā. Kāds ir šo mērījumu vidējais rādītājs – uzdevumi/stundā?

Risinājums

Vidējo vērtību iegūst ar harmoniskā vidējā palīdzību:

$$\text{Uzdevumu skaits stundā} = \frac{4}{\frac{1}{0,5} + \frac{1}{0,45} + \frac{1}{0,53} + \frac{1}{0,43}} = \frac{4}{2 + 2,2 + 1,9 + 2,3} = 0,476$$

(0,476 uzdevumi/stundā ir agrāk izrēķinātā apgrieztā vidējā aritmētiskā vērtība, t.i., $1/2,1 = 0,476$).

Ja novērojamās ātrumu vērtības ir svērtas, pielieto svērtā harmoniskā vidējā aprēķināšanas formulu:

$$\text{Svērtais harmoniskais vidējais} = \frac{1}{\sum_{i=1}^n \frac{W_i}{R_i}} \quad (8)$$



12.9. piemērs

Programma izpildās divos režīmos: 40% komandu izpildās ar ātrumu 10 miljonu komandu/sekundē, bet 60% - ar ātrumu 5 miljoni komandu/sekundē. Kāds ir vidējais svērtais komandu izpildes ātrums – miljons komandu/sekundē?

Risinājums

Datus uzdod tabulas veidā:

Svars	Ātrums	Dalījums
0,4	10 mlj/s	0,4/10=0,04
0,6	5 mlj/s	0,6/5=0,12

Vidējais komandu izpildes ātrums = $1/(0,04 + 0,12)=6,25$ miljons komandu/sekundē.

Atsevišķi novērojumu rezultāti var būt laika vai ātruma attiecības. Piemēram, var veikt kontroles uzdevumus divos datoros un iegūt izpildes laiku attiecības. Tādos gadījumos par vidējās vērtības mēru kalpo ģeometriskā vidējā vērtība:

$$\text{Ģeometriskais vidējais} = \sqrt[n]{\prod_{i=1}^n \text{attiecība}_i} \quad (9)$$



12.10. piemērs

Procesora projektētāji var samazināt takts periodu par 50% (attiecībā 0,5) un samazināt CPI par 25% (attiecībā 0,75). Komandas izpildes laiku aprēķina kā CPI un takts perioda reizinājumu. Kāds ir vidējais samazinājums procentos pēc šādām izmaiņām?

Risinājums

Pielietojot (9) formulu iegūst:

$$\text{Vidējais samazinājums uz vienu modifikāciju} = \sqrt{0,5 \times 0,75} = 0,612 = 38,7\%$$



12.11. piemērs

Divi datori veic programmas 4 ciklus noteiktā taktu skaitā (sk. tabulu). Kāda ir vidējā ciklu ātrdarbības palielinājuma vērtība?

Risinājums

Tabulā parādīts ciklu izpildes laiks (taktis):

Cikls	Dators A	Dators B	Ātrdarbības palielinājums
1	39	20	1,95
2	53	27	1,96
3	27	13	2,08
4	31	13	2,38

Par novērojumiem kalpo taktis, t.i., laika mērījumiem un notikumiem ir vienādi svāri – vidējo ātrdarbības palielinājumu četriem cikliem var izrēķināt ar vidējā ģeometriskā palīdzību (iegūtajam rezultātam nav mērvienības):

$$\text{Vidējais ātrdarbības palielinājums} = \sqrt[4]{1,95 \times 1,96 \times 2,08 \times 2,38} = \sqrt[4]{18,09} = 2,08.$$

Gadījumos, ja novērojamajām attiecībām ir noteikts sadalījums (tās ir svērtas), tad vidējās vērtības iegūšanai pielieto svērtā ģeometriskā vidējā formulu:

$$\text{Svērtais ģeometriskais vidējais} = \prod_{i=1}^n \text{attiecība}^{\omega_i} \quad (10)$$



12.12. piemērs

Divi datori veic programmas 4 ciklus noteiktā taktu skaitā (sk. tabulu). Pieņem, ka 1. cikls izpildās 20 reizes, 2. cikls – 30 reizes, 3. cikls – 50 reizes un 4. cikls – 100 reizes. Kāda ir vidējā ciklu ātrdarbības palielinājuma vērtība?

Risinājums

Tabulā parādīts ciklu izpildes laiks (taktīs):

Cikls	Dators A	Dators B	Ātrdarbības palielinājums	Izpildes reizes
1	39	20	1,95	20
2	53	27	1,96	30
3	27	13	2,08	50
4	31	13	2,38	100

Pirmā cikla svars ir $20/200=0,1$, 2. cikla svars – $0,15$, 3. cikla svars – $0,25$, 4. cikla svars – $0,5$. Svērtā ģeometriskā vidējā vērtība dod šādu rezultātu:

$$\begin{aligned} \text{Vidējais cikla ātrdarbības palielinājums} &= 1,95^{0,1} \times 1,96^{0,15} \times 2,08^{0,25} \times 2,38^{0,5} = \\ &= 1,069 \times 1,106 \times 1,201 \times 1,543 = 2,19. \end{aligned}$$

Saliktais pieauguma temps

Datortehnoloģijā pastāv vēl viens svarīgs jēdziens - saliktais pieauguma temps. Ja, piemēram, katru gadu tiek izstrādāts procesors, kas ir divas reizes ātrāks par iepriekšējo, tad saliktais pieauguma temps būs 100% gadā. Veiktspējas dubultošanās katrus divus gadus dod pieaugumu 41% gadā. Saliktais pieaugums ir identisks saliktajiem procentiem naudas noguldījumu aprēķinos (katru gadu procenti tiek rēķināti no kapitāla summas un iepriekšējā gada procentiem).

Formulas saliktā pieauguma tempa aprēķināšanai:

$$\text{Gala vērtība} = \text{Sākotnējā vērtība} \left(1 + \frac{\text{Pieauguma temps}}{100}\right)^{\text{laika intervāli}} \quad (11)$$

$$\text{Pieauguma ātrums} = 100 \left(\sqrt[\text{laika intervāli}]{\frac{\text{Gala vērtība}}{\text{Sākotnējā vērtība}}} - 1 \right) \quad (12)$$



12.13. piemērs

1995.gadā transakciju serveris darbojās ar ātrumu 100 transakcijas sekundē. Ik gadu tas tika modernizēts, un 2000.gadā darbojās ar ātrumu 700 transakcijas sekundē. Ja visas transakcijas identiskas, tad a) kāds ir saliktais pieauguma temps transakcijās/sekundē par šo 5 gadu periodu? b) kāds bija pieaugums transakcijās/sekundē 2005. gadā, ja saliktais pieauguma temps ir līdzšinējais?

Risinājums

a) Ātrdarbības pieaugums sastāda $700/100=7$. Saliktais pieauguma temps ir $100 \times (\sqrt[5]{7} - 1) = 47,5\%$ gadā.

b) Šī servera pieauguma tempam 2005.gadā vajadzēja būt $700 \times 1,474^5 = 4887$ transakcijas/sekundē.

2) Amdāla likums

Vispārīgā gadījumā Amdāla likums (*Amdahl's law*) raksturo attiecību cena/veiktspēja. Reālajā dzīvē pastāv pretruna: augsta veiktspēja – zema cena. Amdāla likums ļauj novērtēt datora ātrdarbības palielināšanu, kad tiek aprakstīti divi objekti. Viena objekta darbība var tikt paātrināta, otra – nē. Tādējādi var izskaitļot katras lokālās izmaiņas ietekmi uz globālo sistēmas ātrdarbību.

Amdāla likuma vispārīgais pieraksts ir sekojošs:

$$Speedup_{overall} = \frac{Execution\ time_{old}}{Execution\ time_{new}} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}} \quad (13)$$

Vienkāršotā veidā to uzdod šādi: $Speedup = L_{old} / L_{new}$, kur L_{old} ir paātrinājums bez papildus objektiem (parasti tiek pieņemta normalizēta vērtība 1). L_{new} ir paātrinājums pēc kāda viena objekta ieviešanas.

Amdāla likums iztirzāts uz piemēra pamata.



12.14. piemērs. Amdāla likuma pielietojums

Pieņem, ka grib uzlabot savu datoru. Ir divas iespējas:

1 – pievienot FPU (matemātisko līdzprocesora akseleratoru). Tad operācijas ar peldošo punktu izpildīsies 200 reizes ātrāk (*speedup*);

2 – nomainīt disku. Jaunais disks ir 2 reizes ātrāks (*speedup*).

Kādu datora uzlabošanas variantu izvēlēties?

Risinājums

Amdāla likums nosaka, ka šī informācija ir nepietiekama, lai varētu pieņemt lēmumu par to, kurš uzlabojums ir labāks. Rodas daudzi kritiski jautājumi, piemēram, cik bieži katrs atjaunojamais komponents piedalās skaitļojumos vai kā katrs komponents iespaido datora kopējo veiktspēju (*overall*). Taču ar dažiem empīriskiem pieņēmumiem var mēģināt šo uzdevumu atrisināt.

Pieņem, ka lietojumprogrammas izmantos FPU vidēji 10% no kopējā izpildes laika, bet disku – 45% no kopējā laika. Pieņem, ka $L_{old}=1$ abiem variantiem.

Par FPU zināms: $FPUUsed\%=0,1$ un $FPU\ Speedup=200$:

$$S = \frac{L_{old}}{L_{new}} = \frac{1}{(FPUUsed\% / FPU\ Speedup) + (1 - FPUUsed\%)} = \frac{1}{(0,1/200) + (1 - 0,1)} = \frac{1}{0,0005 + 0,9} = 1,11$$

L_{new} zemsvītras izteiksme sastāv no divām daļām. Pirmais saskaitāmais ietver FPU izmantošanu, otrais – bez FPU. FPU izmanto 10% no visa laika, bet tā ir 200x ātrāka nekā 10%. Pārējos 90% dators darbojas ar veco ātrumu.

Secinājums: ar FPU dators ir par 11% ātrāks.

Par disku ir zināms: $DiskUsed\%=0,45$ un $Disk\ Speedup=2$:

$$S = \frac{L_{old}}{L_{new}} = \frac{1}{(DiskUsed\% / Disk\ Speedup) + (1 - DiskUsed\%)} = \frac{1}{(0,45/2) + (1 - 0,45)} = \frac{1}{0,225 + 0,55} = 1,29$$

Pirmais saskaitāmais ietver jaunā diska izmantošanu, otrais – vecā. Jauno disku izmanto 45% no visa laika un tas ir 2x ātrāks. Pārējos 55% dators stādā ar vecā diska ātrumu.

Secinājums: ar jauno disku dators ir par 29% ātrāks. Šajā gadījumā pareizais lēmums būtu iegādāties jaunu disku.

3) Stacionārā veikspēja

Stacionāro veikspēju var izmērīt vairākos veidos. Viena no populārākajām mērvienībām ir „miljons komandu sekundē” - MIPS (*millions of instructions per second*). Procesora veikspēju MIPS mērvienībās aprēķina pēc formulas:

$$MIPS = \frac{\text{Izpildīto komandu skaits}}{\text{Sekunžu skaits} \times 10^6} \quad (14)$$



12.15. piemērs

Konkrētais procesors izpilda 2 miljonus komandu 3 sekundēs. Kāda ir procesora veikspēja MIPS mērvienībās?

Risinājums

Pielietojot (14) formulu, iegūst:

$$MIPS = \frac{2 \times 10^6}{3 \times 10^6} = 0,66 \cdot$$

Vispārīgā gadījumā procesoram MIPS nosaka pēc šādas formulas:

$$MIPS = \frac{\text{Takts frekvence}}{CPI \times 10^6} \quad (15)$$

Veikspēju MIPS mērvienībās vislabāk pielietot procesoru salīdzināšanai ar vienādu instrukciju kopu (piemēram, *Pentium IV* un *Celeron*). Atšķirīgās instrukciju kopās novērtējums MIPS mērvienībās var būt nekorekts. Ja, piemēram, vienam no procesoriem ir iebūvēta reizināšanas komanda, bet otrs reizināšanu izdara ar apakšprogrammas palīdzību, tad procesoru salīdzinājums MIPS mērvienībās būs apšaubāms, jo faktiski tiks izpildītas dažādas komandas. Tādos gadījumos izmanto citu veikspējas novērtējuma metodi.

Šāda veikspējas novērtējuma metode izskatīto uzdevuma izpildes laiku pēc formulas:

$$\begin{aligned} \text{Uzdevuma izpildes laiks} &= \text{Izpildīto komandu skaits} \times CPI \times \text{Takts garums} = \\ &= \frac{\text{Izpildīto komandu skaits} \times CPI}{\text{Takts frekvence}} \end{aligned} \quad (16)$$

Dotā formula var tikt izmantota dažādos veidos. Piemēram, ir nepieciešams noteikt, kādā veidā konkrēta mezgla modifikācija ietekmē veikspēju, nemainot takts frekvenci. Šajā gadījumā vienīgais, kas ir jāņem vērā, ir CPI izmaiņas, jo komandu skaits un takts garums paliek nemainīgi.

Procesora veikspējas novērtēšanai dažādu zinātnisko pielietojumu risināšanā izmanto citu mēru – MFLOPS (*millions of floatingpoint operations per second*). Tiek novērtēta tikai operāciju ar peldošo punktu izpilde – pārējās dienesta komandas (ielāde, pārejas utt.) netiek ņemtas vērā.



12.16. piemērs

Konkrētais procesors izpilda 10 miljonus komandu ar peldošo punktu un 1 miljonu dienesta komandu 50 milisekundēs. Kāda ir šī procesora veikspēja MFLOPS mērvienībās?

Risinājums

$$MFLOPS = \frac{10 \times 10^6}{50 \times 10^{-3} \times 10^6} = 200 \cdot$$

4) Citi veikspēju raksturojošie rādītāji

Datortehnoloģijā pastāv divas iespējas veiktspējas palielināšanai:

- izmantot labākus materiālus;
- lietot labāku arhitektūru.

To var ilustrēt ar piemēriem. Izmantojot uz silīcija pamata ražotas operatīvās atmiņas integrālshēmas, dators var nolasīt 32 bitu datus vai instrukcijas 80 nanosekundēs. Nomainot silīciju pret gallija arsenīdu, var panākt 40 nanosekundes. Tādējādi labāku materiālu tehnoloģijas izmantošana dubulto atmiņas ātrdarbību, bet kā sekas – pieaug izmaksas.

Labākas arhitektūras izmantošanā ilustrācijai var minēt šādu piemēru. Izmanto ātrāku atmiņu uz gallija arsenīda pamata tikai L1 kešatmiņai. Uzlabo programmatūru tā, lai 90% gadījumu programmu piekļuve atmiņai notiktu L1, nevis operatīvajai atmiņai. Ieguvums: piekļuve operatīvajai atmiņai – 80 ns, pieeja L1 – 10 ns. Tādējādi arhitektūras izmaiņa bez radikālas materiālu nomaiņas veicina pie jūtamu ātrdarbības palielināšanos.

Datorsistēmu veiktspējas novērtēšanā izmanto arī caurlaidspējas un reakcijas laika jēdzienus. Caurlaidspēja ir cieši saistīta ar reakcijas jeb atbildes (*respond*) laiku. Piemēram, ja dators M programmu A izpilda 4 sekundēs un tiek veiktas tehnoloģijas un arhitektūras izmaiņas, kā rezultātā programma tiek izpildīta 2 sekundēs, tad var teikt, ka datora caurlaidspēja ir dubultojusies.

M reakcijas laiks uz programmu A ir samazinājies uz pusi, t.i., dators M izpilda programmu divās sekundēs. Līdz ar to izpildes laiks no A sākuma un beigām ir M reakcijas laiks uz A izpildi.



12.17. piemērs

Dators M izpilda programmas A1 un A2 2 stundās (1 stunda katrai programmai), t.i., M reakcija uz katru no programmām ir 1 stunda. Pieņem, ka šajā gadījumā M ir dators, kurā programmas izpildās viena pēc otras (*sequential*).

Ja turpmāk pieņem, ka M ir paralēlās darbības dators un izpilda abas programmas 1 stundas laikā (katra no tām atsevišķi joprojām darbojas 1 stundu). Tātad M reakcijas laiks ir 1 stunda.

Var secināt, ka paralēlisma lietošana palielina reakcijas laiku un caurlaidspēju.

CPU ir būtisks komponents datoru arhitektūrā, kam ir liela nozīme veiktspējas novērtēšanā.

CPU reakciju var izskaitļot pēc formulas (identiska (16) formulai): $L=CPI \times IC \times CT$, kur

CPI – taktu skaits uz komandu;

IC – komandu skaits (*Instruction Count*);

CT – takts garums (*Cycle Time*).

Kad dators M izpilda programmas A komandas, tad kopējais CPU laiks ir vienāds ar CPU reakciju.

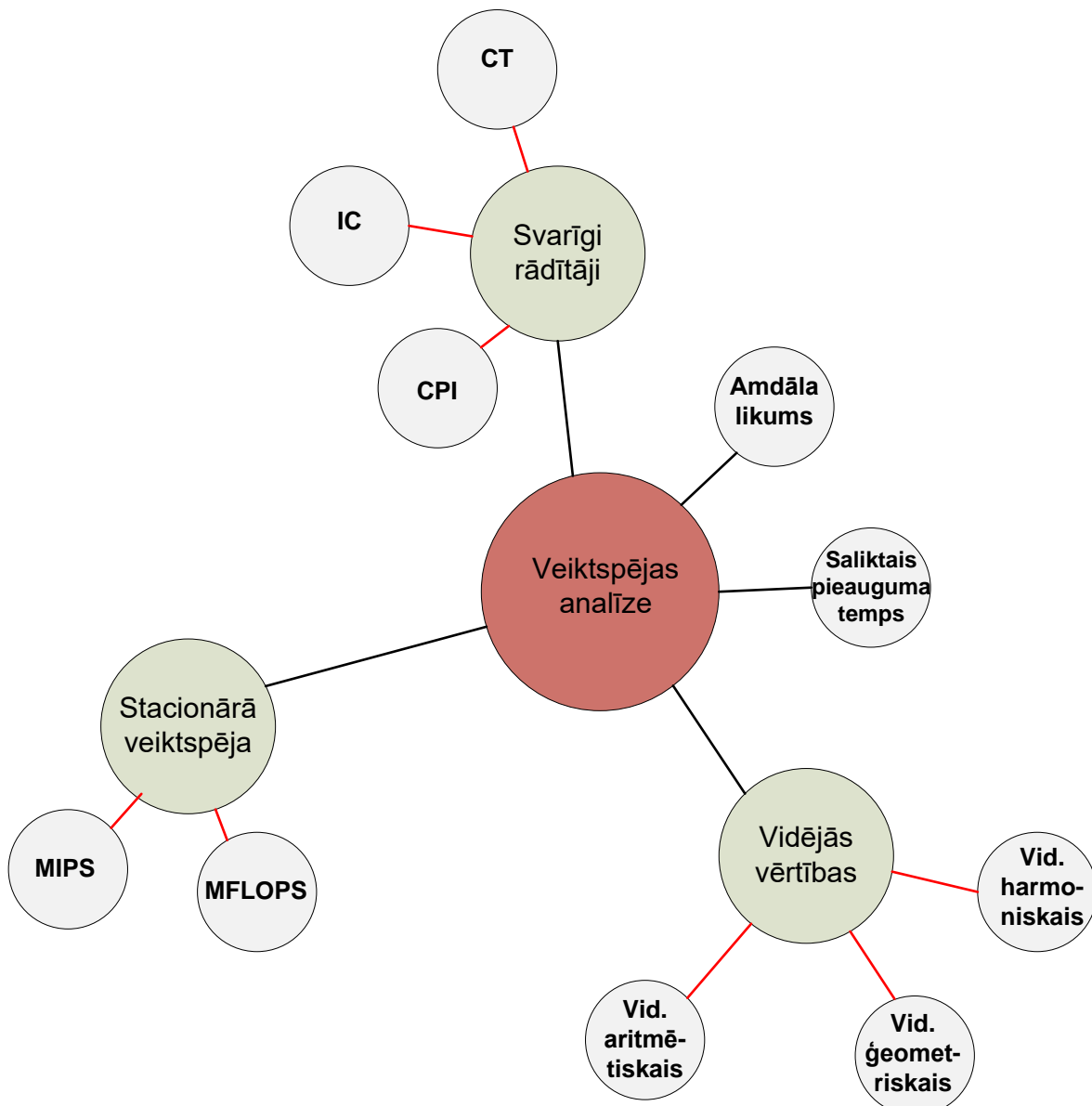
Atsevišķos gadījumos praksē tiek pielietoti arī Mūra (*Moore G.*) un Groša (*Grosch H.*) likumi [1].

NODAĻAS KOPSAVILKUMS

- Veiktspēja - sistēmas vai tās komponentu spēja izpildīt paredzētās funkcijas. Kā veiktspējas kvantitatīvie kritēriji parasti tiek izmantoti atbildes laiks, caurlaidspēja un izmantojamība.
- Datortehnoloģijā pastāv divas iespējas veiktspējas palielināšanai:
 - izmantot labākus materiālus;
 - lietot labāku arhitektūru.
- Datorsistēmu novērtēšanā būtiski ir divi rādītāji:
 - uzdevuma izpildes laiks;
 - uzdevuma izpildes ātrums.

- Galvenais laika raksturlielums, ko pielieto procesoru novērtēšanā, ir taktu skaits komandā jeb CPI. Šī vērtība nosaka laiku taktīs, ko procesors vidēji patērē komandas izpildei.
- Loģiska ir vēlme izteikt datorsistēmas veiktspēju ar vienu skaitli. Attiecībā uz datorsistēmām var veikt triju veidu mērījumus:
 - laiks, kas nepieciešams uzdevuma izpildei;
 - ātrums, ar kādu izpildās uzdevums;
 - ātruma vai laika attiecība veiktspējas novērtēšanai.
- Kvantitatīvie novērtējumi:
 - aritmētiskais vidējais un svērtais aritmētiskais vidējais;
 - ģeometriskais vidējais un svērtais ģeometriskais vidējais;
 - harmoniskais vidējais un svērtais harmoniskais vidējais.
- Procesora stacionāro veiktspēju var uzdot MIPS mērvienībās.
- Operāciju ar peldošo punktu novērtēšanā izmanto MFLOPS mērvienību.
- Datorsistēmu veiktspējas novērtēšanā izmanto arī caurlaidspējas un reakcijas laika jēdzienus.

12.2.attēlā parādīts pēdējā nodaļā minēto svarīgāko jēdzienu koks.



12.2. att. Nodaļā minēto svarīgāko jēdzienu koks

☒ Uzdevumi patstāvīgajam darbam

1. Datorsistēma S1 maksā Ls 10 000, datorsistēma S2 - Ls 15 000.
Tās izpilda 2 programmas:

	Izpildes laiks sistēmā1	Izpildes laiks sistēmā2
<i>Programma1</i>	10 sec.	5 sec.
<i>Programma2</i>	3 sec.	4 sec.

Kad *programma1* izpildās sistēmā1, sistēma1 veic 20×10^6 instrukcijas.
Kad *programma1* izpildās sistēmā2, sistēma2 veic 16×10^6 instrukcijas.

- Ja apskata tikai *programmu1*, kura sistēma un kāpēc ir vairāk efektīga cenas ziņā?
- Izskaitļot MIPS *programmai1* sistēmā1 un sistēmā2?
- Pieņem, ka sistēmai1 takts frekvence ir 20 MHz, bet sistēmai2 - 30 MHz.
Kāds ir CPI katrai sistēmai, izpildot *programmu1*?

2. Ir dotas sistēmas:

Sistēma M1 (takts frekvence: 50 MHz)	
Instrukciju klases	CPI katrai klasei
A	1
B	2
C	3
D	4

Sistēma M2 (takts frekvence: 75 MHz)	
Instrukciju klases	CPI katrai klasei
A	2
B	2
C	4
D	4

- Salīdzināt sistēmu M1 un M2 veiktspējas.
- Pieņem, ka M1 un M2 izpilda programmu P, kuras instrukcijas sadalās šādi:
25% -klasei A, 25% - klasei B, 25% - klasei C, 25% - klasei D.

Kurai no sistēmām būs labāka veiktspēja?

3. Doti N lati datora modernizācijai. Var iegādāties 2 iekārtas veiktspējas palielināšanai:
Iekārta1: ja tiks izmantota, vidēji 25% no kopējā laika tā palielinās datora veiktspēju par faktoru ar vērtību 200 (piem., 200x ātrāks)
Iekārta2: ja tiks izmantota, vidēji 60% no kopējā laika tā palielinās datora veiktspēju par faktoru ar vērtību 10 (piem., 10x ātrāks)
Kādu iekārtu būtu vērts iegādāties un kāpēc?

4. Pieņem, ka programma ar 400 instrukcijām tiek izpildīta 1,8GHz datorā. Instrukciju izpildes biežums un ciklu skaits vienai instrukcijai dots tabulā:

Operācija	Biežums	CPI
ALU operācijas	40%	1
Ielāde	20%	2
Ieraksts	15%	2
Cikli	20%	3
I/O	5%	4

- Kāds ir šo instrukciju kopējais CPI?
- Kāds ir programmas CPU laiks?
- Kāds ir šīs arhitektūras MIPS?

5. Trīs datori veic divus uzdevumus tabulā uzrādītajos laika intervālos:

Uzdevums	Dators A	Dators B	Dators C
1	1	10	20
2	1000	100	20

Izskaitļot $x\%$ vērtību, ja

- 1) A par $x\%$ ātrāks nekā B uzdevumam Nr. 1;
- 2) A par $x\%$ ātrāks nekā C uzdevumam Nr. 1;
- 3) B par $x\%$ ātrāks nekā C uzdevumam Nr. 1;
- 4) B par $x\%$ ātrāks nekā A uzdevumam Nr. 2;
- 5) C par $x\%$ ātrāks nekā A uzdevumam Nr. 2;
- 6) C par $x\%$ ātrāks nekā B uzdevumam Nr. 2;
- 7) B par $x\%$ ātrāks nekā A abiem uzdevumiem;
- 8) C par $x\%$ ātrāks nekā B abiem uzdevumiem;
- 9) C par $x\%$ ātrāks nekā A abiem uzdevumiem.

IETEICAMĀS LITERATŪRAS SARAKSTS

I Obligātā literatūra

1. Hennesy J.L., Patterson D.A. Computer Architecture-A Quantitative Approach. –Fourth Edition. –Morgan Kaufman Publishers, 2007.
2. Орловский Г. Введение в архитектуру микропроцессора 80386 / Г. Орловский. - СПб. : Сеанс-Пресс Ltd : Центр инфотехнологии ИНФОКОН, 1992. - 240 с.
3. Соломенчук Валентин. Аппаратные средства персональных компьютеров : самоучитель / В.Соломенчук. - СПб. : БХВ-Петербург, 2003. - 502 с. : ил. - (Самоучитель). - Библиогр.: с.495 (10 назв.). - Предм.указ.: с.496-502.
4. Таненбаум Эндрю. Архитектура компьютера / Э. Таненбаум ; Пер. с англ. И Ткачева. - Изд. 4-е. - СПб. : Питер, 2005. - 698 с. : ил. - (Классика Computer Science). - Библиогр.: с.647-654. - Алфавит. список лит. : с.654-664. - Алфавит.указ.: с.685-698.
5. Томпсон Роберт. Железо ПК : энциклопедия / Р.Томпсон ; Б.Томпсон ; Пер. с англ. Д.Солнышков. - Изд. 2-е. - СПб. : Питер, 2003. - 864 с. : ил. - Алфавит. указ. : с. 847-864.
6. Юров Виктор. Assembler : практикум (+ дискета) / В. Юров. - Санкт-Петербург : Питер, 2002. - 395 с. - Библиогр.: с.393 - 395 (48 назв.). - дискета.

II Ieteicamā literatūra

7. Stallings W. Computer Organisation & Architecture. Designing & Performance. – Fourth Edition.-Prentice-Hall International, 1996.
8. Джордейн Р. Справочник програмиста. – Москва: ФиС, 1991.
9. Магда Ю. Аппаратное обеспечение и эффективное программирование. – Питер, 2007.
10. Майерс Г. Архитектура современных ЭВМ. Том. 1,2. – Москва: МИР, 1988.
11. Мюллер Скотт. Модернизация и ремонт ПК / С. Мюллер ; Пер. с англ. под ред. А. Кушнира. - 13-е изд. - М.*СПб.*Киев : Вильямс, 2002. - 1180 с. : ил. - В приложении компакт-диск"Модернизация и ремонт ПК".

III Internet resursi

12. <http://asm.shadrinsk.net/>
13. <http://greenbox.ru/catalog/files/?id=30>
14. http://lv.wikipedia.org/wiki/Datora_arhitekt%C5%ABra
15. <http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-826Principles-of-Computer-SystemsSpring2002/CourseHome/index.htm>
16. <http://www.amd.com/gb-uk/>
17. <http://www.cs.princeton.edu/courses/archive/spring04/cos217/>
18. <http://www.cs.rtu.lv/Pubs/Cipa/Arhit2003/DArhSatur.htm>
19. <http://www.intel.com/>
20. <http://www.liis.lv/sktehves/neim1.htm>
21. http://www.lza.lv/lat/Akad_lekcijas/Padegs_041125/Padegs_Akad_lekc_041125_teksts.htm
22. <http://www.pcguides.com/topic.html>
23. <http://www.pcports.ru/>
24. <http://www.sandpile.org/>
25. <http://www.termini.lv/>

