1 Увод

У овој глави ћемо се упознати са разлозима за увођење паралелних рачунарских система, њиховом класификацијом, као и основама мерења и извештавања о перформансама рачунарских система.

1.1 Разлози увођења паралелних система

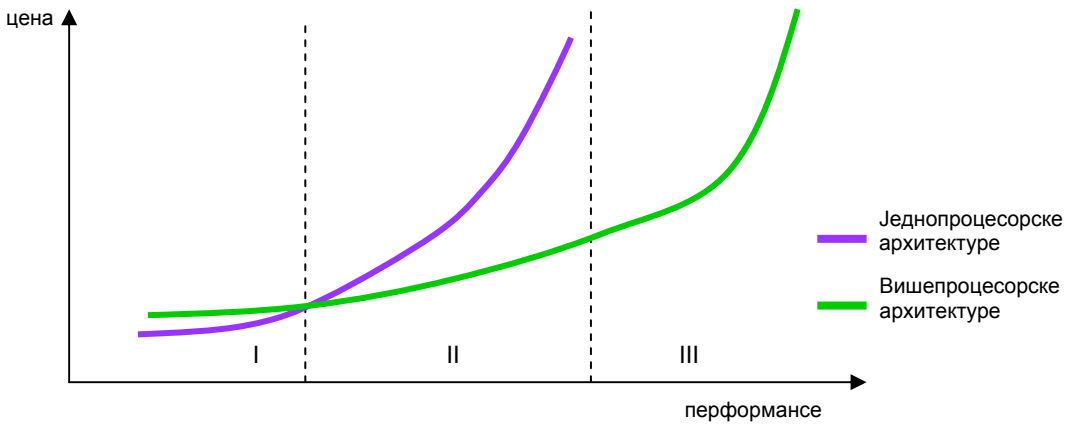
Све већи захтеви у многим областима примене рачунара намећу потребу за стварањем нових генерација рачунара који ће имати значајно боље перформансе а да при том цена остане у разумним границама. Од нових генерација рачунара тражи се флексибилност како архитектуре, тако и технологије градивних компоненти. Већина машина треба да има могућност конфигурисања за различите задатке (прорачуни опште намене, мрежни сервери, сервери за базе података, радне станице и др.). Нове генерације треба да буду што независније од технологије компонената како би се олакшала надоградња система коришћењем нових, бржих технологија. Битан аспект флексибилности је и проширљивост система.

Следећа битна карактеристика је могућност рада у мрежи обзиром да је време *stand-alone* система на измаку.

Кориснички интерфејс је незаобилазна карактеристика сваког од система, јер корисници желе стандардне оперативне системе како би се искористио постојећи софтвер.

Један од начина, или можда једини, да се испуне сви ови захтеви је да се приликом пројектовања изабере паралелна архитектура. Паралелни рачунари имају четири основне предности над секвенцијалним архитектурама:

1. **Перфомансе**: Колико год да је брз најбољи CPU, још већа брзина рада се постиже ако неколико таквих процесора ради у паралели.
2. **Однос цена/перфомансе**: Слика 1.1 показује како се креће однос цена и перформанси код једнопроцесорских и вишепроцесорских машина. На графику се могу уочити три области. Прва област је област ниских перформанси и такве захтеве једнопроцесорске машине могу да задовоље по нижој цени од вишепроцесорских. У другој области, где су перформансе знатно увећане, имамо да такве услове вишепроцесорске машине испуњавају по знатно нижој цени. Најзад, у трећој области су захтеви за перформансама које само вишепроцесорске машине могу да испуне.



Сл. 1.1. Однос цена/перформансе код једнопроцесорских и вишепроцесорских машина.

3. **Модуларна проширљивост**: Ова особина има директан утицај на успешност нових генерација рачунара.
4. **Увећана толерантност на грешке**: Вишепроцесорска архитектура је много толерантнија на грешке услед отказа појединих компонената, што уједно значи и већу поузданост.

1.2 Класификација паралелних система

Једна од најпознатијих класификација паралелних система је Флинова (Flynn) класификација. Поред ње постоје и друге, алтернативне класификације.

1.2.1 Флинова класификација

Најчешће коришћена класификација рачунарских система према њиховим могућностима паралелне обраде је Флинова класификација из 1972. године. Према Флину, паралелни системи се деле у четири категорије:

- **SISD** (*Single instruction, single data stream*). Један процесор извршава један низ инструкција над подацима запамћеним у јединствној меморији. Једнопроцесорске машине спадају у ову категорију.
- **SIMD** (*Single instruction, multiple data stream*). Једном машинском инструкцијом управља се симултаним радом извесног броја процесних елемената. Сваки процесни елемент има сопствену меморију за податке, тако да се свака инструкција извршава над различитим скупом података у различитим процесним елементима. Векторски процесори и процесорска поља спадају у ову категорију.
- **MISD** (*Multiple instruction, single data stream*). Низ података преноси се скупу процесора при чему сваки од њих извршава различиту секвенцу инструкција. Овакав тип машине није комерцијално имплементиран.
- **MIMD** (*Multiple instruction, multiple data stream*). Скуп процесора симултано извршава различите секвенце инструкција над различитим скуповима података. Симетрични мултипроцесори, кластери и NUMA системи припадају овој категорији.

На слици 1.2. приказана је ова класификација у облику стабла

Сл. 1.2. Класификација паралелних архитектура процесора.

1.2.2 Алтернативне класификације

Фенг (Tse-yun Feng) је предложио *степен паралелизма* као критеријум класификације рачунара. Максимални број битова који рачунар може да обради у јединици времена назива се *максимални*

степен паралелизма P. Нека је *Pi* број битова који се може обрадити у *i*-том тактном периоду и неке се посматра *T* тактних периода. Тада је *средњи степен паралелизма*

$$P_a = \frac{\sum_{i=1}^{T} P_i}{T}.$$

У општем случају је $P_i \le P$. Према томе, с*тепен искоришћења* μ рачунарског система у оквиру *T* тактних периода дефинише се као

$$\mu = \frac{P_a}{P} = \frac{\sum_{i=1}^{T} P_i}{T \cdot P}.$$

Ако је моћ израчунавања неког процесора потпуно искоришћена (паралелизам је у потпуности искоришћен), тада имамо да је $P_i = P$ за свако *i* а $\mu = 1$ за стопроцентну искоришћеност. Степен искоришћења зависи од апликационог програма који се извршава.

Даље, под *бит-одреском* подразумева се низ битова, по један из сваке речи, на истој вертикалној бит позицији. Нека је *n* дужина речи а *m* дужина бит-одреска. Максимални степен паралелизма $P(C)$ за дати систем *C* једнак је $P(C) = n \cdot m$. На основу овога постоје четири типа обраде:

- Бит-серијска (WSBS) – $n = m = 1$.
- Обрада бит-одреска (WPBS) – $n = 1$, $m > 1$.
- Обрада реч-одреска (WSBP) – $n > 1$, $m = 1$.
- Потпуна паралелна обрада (WPBP) - $n > 1$, $m > 1$.

Бит серијска обрада је постојала само код првих рачунара. Обрада реч-одреска се може наћи код већине данашњих рачунара, јер се једна реч од *n* битова обрађује истовремено. Најбржи начин обраде је потпуна паралелна обрада, јер се истовремено обрађује поље од $n \cdot m$ битова.

Хендлер (Wolfgang Händler) је предложио класификацију за идентификовање степена паралелизма и степена проточности уграђен у структуру хардвера рачунарског система. Он посматра паралелну и проточну обраду на три подсистемска нивоа:

- Управљачка јединица процесора (PCU).
- Аритметичко-логичка јединица (ALU).
- Кола на нивоу битова (BLC).

Функција PCU-а одговара једном CPU-у. BLC одговара комбинационој мрежи потребној да се обави једнобитна операција у ALU.

Рачунарски систем *C* карактерише се уређеном тројком која садржи шест независних ентитета

$$T(C) = \langle K \times K', D \times D', W \times W' \rangle$$

где је:

K – број процесора (PCU) у рачунару;
D – број ALU којима управља једна PCU;
W – дужина речи једне ALU;
K' – број PCU који могу бити проточни;
D' – број ALU које могу бити проточне;
W' – број проточних фаза у свим ALU.

1.3 Мерење и извештавање перформанси

1.3.1 Фактори који утичу на перформансе и њихово мерење

На шта мислимо када кажемо да је један рачунар бржи од другог? Корисник рачунара мисли да је рачунар бржи када се његов програм изврши за краће време. Управник неког рачунског центра сматра да је рачунар бржи ако изврши више послова за задато време. Корисник је заинтересован за смањење времена одзива (времена извршења) док је управник рачунског центра заинтересован за повећање пропусне моћи система (*throughput*).

Ако поредимо алтернативе у пројектовању често желимо да поредимо перформансе различитих машина, нпр. X и Y. Реченица "X је брже од Y" значи да је време одзива мање код машине X него код машине Y. Прецизније, "X је *n* пута брже од Y" значи

$$\text{Време извршења}_Y/\text{Време извршења}_X = n.$$

Како је време извршења реципрочно перформанси, важи

$$n = \text{Перформанса}_X/\text{Перформанса}_Y$$

Конкретно, реченица " X је 1.5 пута брже од Y" означава да је број задатака који се заврше у јединици времена на машини X 1.5 пута већи од броја задатака који су завршени на машини Y за исто време.

Како су перформанса и време извршења реципрочне, повећање перформансе смањује време извршења. Обично се уместо појмова *смањење* или *повећање* користе термини "побољшање перфоманси" или "побољшање времена извршења".

Без обзира да ли нас интересује пропусна моћ или време одзива, кључна мера је време. Рачунар који извршава исти износ посла за краће време је бржи. Разлика је само да ли се мерење врши на једном или више послова. Нажалост, време није једина мера која се наводи када се пореде перформансе. Постоји известан број и других популарних мера које су усвојене у жељи да се нађе разумљива универзална мера перформанси рачунара. Ипак, једина конзистентна и поуздана мера перформанси је време извршења стварних програма а све остале алтернативе могу да воде до погрешних закључака.

И време извршења може да се дефинише на различите начине у зависности од тога шта се узима у обзир. Најдиректнија дефиниција времена означава се као време које показује часовник или време одзива или протекло време и представља време потребно да се изврши задатак обухватајући више активности као што су:

- Време потрошено на приступе диску.
- Време потрошено на приступе меморији.
- Време посвећено У/И активностима.
- Време које се потроши на рад оперативног система, и др.

Код мултипрограмирања CPU ради на другом програму код чека да се обави У/И активност претходног програма и не мора обавезно да минимизира протекло време једног програма. Према томе, потребан нам је појам који узима у обзир ову активност! Тај појам је *CPU време* – време потрошено на CPU активност, дакле не и на чекање на обављање У/И операција или на извршавање других програма. Наравно, корисник осећа протекло време а не CPU време. CPU време се даље дели на *корисничко CPU време* (потрошено на програм) и *системско CPU време* (потрошено на рад оперативног система који обавља задатке захтеване од стране програма који се извршава).

1.3.2 Избор програма за процену перформанси

Корисник рачунара који свакодневно извршава исти програм био би одличан кандидат да изврши процену перформанси неке нове машине. Да би ту процену извршио потребно је само да упореди време извршења свог уобичајеног посла. Међутим, овакаве идеална ситуација се не јавља често. У већини случајева морамо се ослонити на друге методе у нади да ће оне на добар начин предвидети перформансе нове машине. Постоје четири нивоа програма који се користе за евалуацију рачунара (наведени по опадајућем редоследу тачности предвиђања).

1. **Стварни програми.** Купац ових програма не мора да зна који део времена се троши на извршење ових програма, али зна да неки корисници користе ове програме за решавање реалних проблема. Примери ових програма су компилатори, текст процесори, CAD алати и др.
2. **Кернели.** Било је више покушаја да се издвоје мали али кључни делови стварних програма па да се употребе за процену перформанси. Најпознатији примери су *Livermore Loops* и *Linpack*. Насупрот стварним програмима, ниједан корисник неће покренути неко њихово језгро, осим у сврху процене перформанси. Кернели се најбоље могу искористити за изоловање појединих карактеристка машине као пут за објашњење разлика у перформансама стварних програма.
3. **Бенчмарк играчке.** Ови програми типично заузимају између 10 и 100 линија кода и дају предвидљиве резултате. Међу најпопуларнијим су програми као што су Ератостеново сито, Puzzle и Quicksort, јер су мали и једноставни за коришћење на

сваком рачунару. Ови програми се најбоље могу употребити као почетнички програмерски задаци.

4. **Синтетички бенчмарк програми.** Слични по филозофији кернелима, ови програми се труде да погоде просечну фреквенцију операција и операнада великог скупа програма. Whetstone и Dhrystone су најпопуларнији међу њима. Корисници иначе не извршавају ове програме, јер они не израчунавају ништа што би им користило. У ствари, синтетички бенчмарк програми су чак и даље од реалности у односу на кернеле и нису чак ни делови стварних програма што кернели обично јесу.

У последње време постају популарне колекције бенчмарк програма које покушавају да измере перформансе процесора на различитим апликацијама (SPEC).

Што се тиче извештавања о процењеним перформансама, основни принцип треба да буде *репродуктивност*. Треба навести све што ће омогућити другом експериментатору да понови резултате. На жалост, поредећи рачунарске часописе и аутомобилске видимо да ови други садрже много више података о производу који приказују.

SPEC бенчмарк даје прилично исцрпан опис машине, опције компилатора, а даје и податке за тзв. *основну линију* мерења перформанси и за оптимизоване резултате.

1.4 Квантитативни принципи пројектовања рачунара

Један од најважнијих и најутицајних принципа у пројектовању рачунара је да се направи да се оно што се најчешће јавља учини брзим. Слично је код одлучивања о додели ресурса, јер је утицај фреквентнијх случајева на перформансе рачунара већи од случајева који се ретко јављају. Осим тога фреквентни случајеви су обично једноставнији па се могу и обрадити брже него што је то случај са онима који су мање фреквентни. На пример, ако сабирамо два броја можемо очекивати да се прекорачење ретко јавља па треба побољшати перформансе уобичајенијег случаја када се прекорачење не јавља. Такав избор ће успорити случај када се јави прекорачење али, како се то не дешава сувише често, утицај на перформансе није од значаја.

1.4.1 Амдалов закон

Побољшање перформанси које се добија побољшањем неког дела рачунара може се израчунати користећи Амдалов закон (Amdhal, 1967). Овај закон каже да је максимално убрзање које се добија побољшањем неког дела машине ограничено процентуалним уделом тог дела машине.

Амдалов закон дефинише убрзање које се добија побољшавањем одређеног дела машине. Под претпоставком да смо побољшали неки део машине, убрзање је однос перформанси машине са побољшањем и перформанси машине без побољшања. Алтернативно, то је однос времена извршења програма на непобољшаној машини и времена извршења на побољшаној машини.

Помоћу Амдаловог закона можемо да нађемо *убрзање* (*S*) добијено неким побољшањем на основу два фактора:

1. Дела времена израчунавања у оригиналној машини који се може побољшати (*F*e).
2. Убрзања које се добија само на побољшаном делу (*S*e).

Време извршења на побољшаној машини тада се може добити као:

$$ET_{new} = ET_{old} \times \left((1 - F_e) + \frac{F_e}{S_e} \right)$$

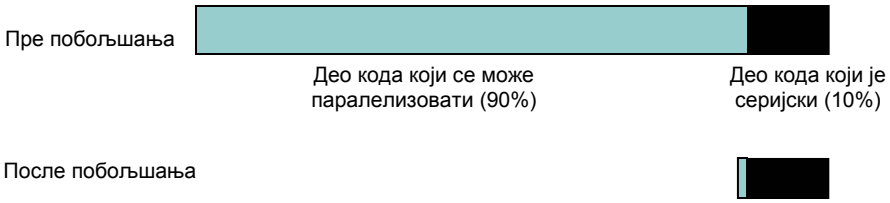
Укупно убрзање које се добија је однос времена извршења на старој и новој машини:

$$S = \frac{ET_{old}}{ET_{new}} = \frac{1}{(1 - F_e) + \frac{F_e}{S_e}}$$

Амдалов закон изражава закон смањења добитка: инкрементално побољшање само једног дела израчунавања опада како се то побољшање уводи. Важна последица је да ако је побољшање могуће само на делу посла онда не можемо убрзати посао више од $1/(1 - F_e)$ пута.

Пример 1.1. Нека имамо ситуацију да се 90% неког програма може паралелизовати, тј. извршавати симултано ако уведемо више процесора у систем, док је 10% програма строго секвенцијално.

Претпоставимо још да можемо неограничено додавати процесоре тако да се конкурентни део посла извршава за време које тежи нули. На слици 1.3. је графички илустрована ова ситуација.



Сл. 1.3. Ефекти Амдаловог закона.

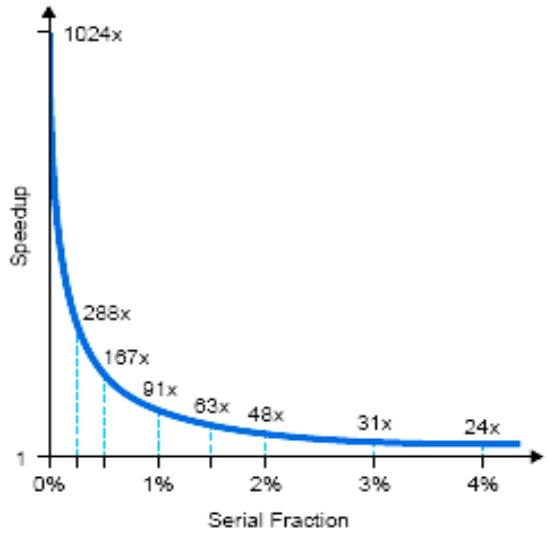
Видимо да, чак иако је број процесора неограничен, и убзање тог дела бесконачно, време извршења представља бар 10% од оригиналног времена и теоретски максимално убрзање је 10!

1.4.2 Густавсонов закон

У пракси величина проблема расте са бројем процесора. Према Амдаловом закону, ако је s време потрошено на серијски део програма (на серијском процесору) и p време потрошено на паралелни део програма (на серијском процесору), убрзање је

$$S = (s + p)/(s + p/N)$$
$$= 1/(s + p/N), \quad s + p = 1.$$

Приметимо да су s и p заправо процентуални удели серијског и паралаленог дела програма па је претпоставка да је $s + p = 1$ коректна.



Сл. 1.4. Убрзање према Амдаловом закону.

Густавсон примећује да су, у пракси, многи проблеми *скалабилни*, тј. да њихова величина расте са бројем процесора. Ако су s' и p' времена потрошена на серијски и паралелни део програма (на паралелном систему са N процесора), тада једном процесору треба време од $s' + p'N$ да изврши задатак.

Сада је *скалирано убрзање* једнако

$$S_{\text{scaled}} = (s' + p'N)/(s' + p')$$
$$= s' + p'N$$
$$= N + (1 - N)s', \; s' + p' = 1$$

Ова функција је линеарна и зависи од броја процесора! Тако је Густавсонов закон унео више оптимизма него што је то наговештено Амдаловим законом.

1.4.3 Једначине перформанси CPU-а

Већина рачунара је направљена тако да се све активности одвијају на основу тактног сигнала одређене фреквенције. Тактни сигнал је правоугаоног таласног облика а једна његова периода се назива и *циклусом*. Фреквенција тактног сигнала може се изразити као:

$$f_c = 1/\tau$$

где је τ период тактног сигнала. Корисничко CPU време може се изразити као број протеклих тактних периода помножен периодом тактног сигнала, тј.

$$T_{CPU} = N_c \tau = N_c / f_c$$

где је Nc број тактних периода потребних да са изврши програм.

Ако познајемо укупан број циклуса и број инструкција можемо да израчунамо средњи број циклуса по инструкцији (*clock Cycles Per Instruction* - CPI)

$$CPI = N_c /N,$$

где је N број инструкција у програму.

Сада је

$$N_c = CPI \cdot N,$$

односно

$$T_{CPU} = N_c \tau = N \cdot CPI \cdot \tau..$$

Можемо видети да корисничко CPU време зависи од три фактора: трајања периода тактног сигнала, средњег броја тактних циклуса по инструкцији и броја инструкција у програму. Та зависност је директно пропорционална, односно за колико се побољша било који од ова три фактора у тој мери се побољша и корисничко CPU време. На жалост, тешко је ове параметре мењати независно једне од других, јер постоје међузависности између фактора који утичу на поменута три параметра:

- На трајање периода утичу технологија и организација хардвера.
- На CPI утичу организација и скуп инструкција.
- На број инструкција утичу скуп инструкција и технологија компилатора.

На срећу, многе потенцијалне технике за побољшање перформанси у основи врше побољшање једне од компонената са малим и предвидљивим утицајем на друге две компоненте.

Треба имати у виду да је израз који смо добили за перформансу (корисничко CPU време) груба процена обзиром да број тактних периода по инструкцији може веома да варира од инструкције до инструкције. Стога се инструкције деле у групе са приближно истим бројем тактних периода по инструкцији. У том случају повољније је CPI израчунати на следећи начин. Укупан број тактних периода потребних за извршење програма је:

$$N_c = \sum_{i=1}^{n} (CPI_i \cdot N_i)$$

где је n број различитих типова инструкција у програму, N_i инструкција типа i које се јављају у програму и CPI_i средњи број циклуса потребан за извршење инструкција типа i. Даље имамо:

$$CPI = \frac{N_c}{N} = \frac{\sum_{i=1}^{n} (CPI_i \cdot N_i)}{N} = \sum_{i=1}^{n} (CPI_i \frac{N_i}{N})$$

Израз N_i/N се назива *фреквенција* (вероватноћа) појављивања инструкције типа i у програму. Сада се перформанса може изразити као:

$$T_{CPU} = \sum_{i=1}^{n} (CPI_i \cdot N_i) \cdot \tau = N \sum_{i=1}^{n} (CPI_i \cdot \frac{N_i}{N}) \cdot \tau .$$

Алтернативна мера је MIPS (Milion Instructions Per Second).

$$\begin{aligned} \text{MIPS} &= N/(T_{\text{CPU}} \cdot 10^6) = \\ &= N/(N \cdot CPI \cdot \tau \cdot 10^6) = \\ &= fc/(CPI \cdot 10^6) \end{aligned}$$

$$T_{\text{CPU}} = N/(\text{MIPS} \cdot 10^6)$$

Нажалост, ова мера није увек добра процена перформанси рачунар! Осим MIPS-а користи се и MFLOPS (Milion Floating Point Opearations Per Second).

$$\text{MFLOPS} = N_{FP}/(T_{\text{CPU}} \cdot 10^6)$$

Литература

[1] J. L. Gustafson, *Reevaluating Amdhal's Law*, Communications of the ACM, **31**:5, (May 1988).

[2] K. Hwang, F. A. Briggs, *Computer Architecture and Parallel Processing*, New York, McGraw-Hill, 1984.

[3] D. A. Patterson, J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, 2/e, Morgan Kaufmann Publishers, inc. San Francisko, California, 1996.

[4] W. Stallings, *Computer Organization and Architecture*, 6/e, Prentice Hall, 2003.

[5] J. Till, *Computer System Architecture*, Electron Des. (USA), vol. **37**, no. 1, pp 50-63 (12. Jan 1989).

2 Проточност инструкција

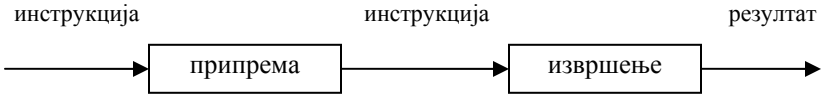
Током еволуције рачунарских система боље перформансе су постизане коришћењем побољшања у технологији али и побошањима у организацији. Један од таквих организационих приступа је *проточност*.

2.1 Стратегија проточности

Проточност инструкција је слична покретној траци у фабрици, где производ пролази кроз различите фазе производње. На производима у различитим фазама се може радити симултано. Овај процес се назива проточност, јер се, као у неком цевоводу, нови улази прихватају на једном крају пре него што се претходно прихваћени улази појаве на другом крају.

Ради једноставности, замислимо да се инструкције састоје од две фазе: припреме инструкције и извршења инструкције. Постоји време током извршења инструкције када се не приступа меморији. То време се може искористити за прибављање наредне инструкције паралелно са извршењем текуће.

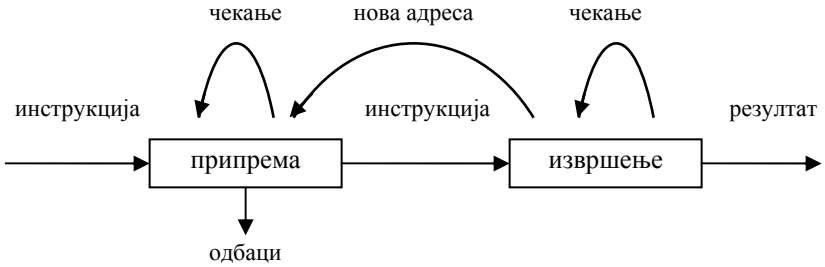
Проточни систем у овом случају има два независна степена (слика 2.1). Први степен прибавља инструкцију и баферује је. Када се други степен ослободи, први степен му предаје баферовану инструкцију. Док други степен извршава инструкцију, први користи чињеницу да постоји неискоришћен меморијски циклус и прибавља у бафер следећу инструкцију. Ово се назива *предприпрема инструкције* или *преклапање фазе припреме*.



Сл. 2.1. Двостепени проточни систем.

Двостепени проточни систем убрзава извршење инструкције. Ако су фазе припреме и извршења једнаког трајања, циклус инструкције биће преполовљен. Међутим ово удвостручење брзине није вероватно из два разлога:

1. Време извршења је у општем случају дуже од времена припреме.
2. Инструкције условног гранања чине да је адреса инструкције која ће се наредна прибавити у ствари непозната.



Сл. 2.2. Детаљнија представа двостепеног проточног система.

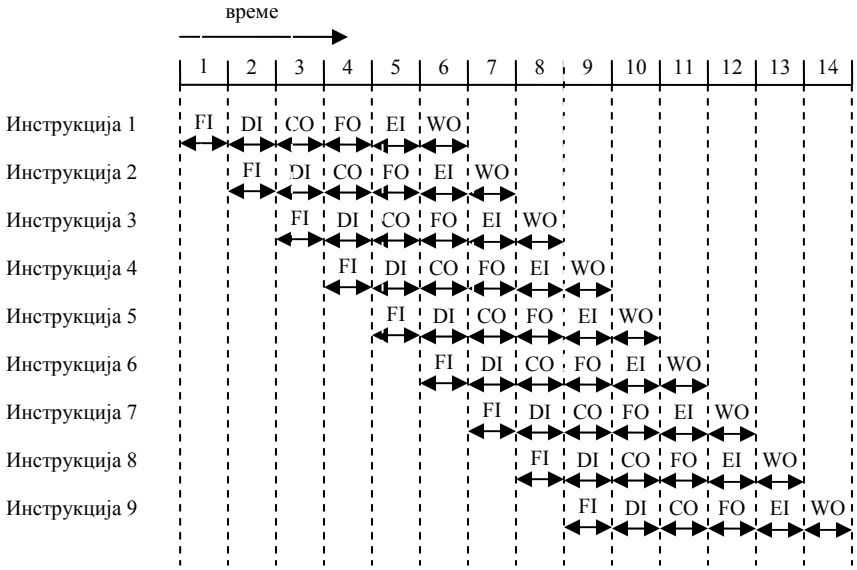
Нагађање може да смањи губитке настале због другог разлога. Треба поступити према једноставном правилу: када се инструкција условног гранања пренесе из степена припреме у степен извршења, степен припреме прибавља наредну инструкцију која је у меморији одмах иза

инструкције гранања. Према томе, ако се гранање не догоди нема губитка. У случају гранања, прибављена инструкција се мора одбацити и прибавити циљна инструкција гранања (слика 2.2).

Иако поменути фактори смањују могућу ефикасност двостепеног проточног система, ипак се јавља извесно убрзање. Како би добили још веће убрзање, извршење инструкције треба поделити на више фаза, што значи да и проточни систем треба да има више степени. Размотримо декомпозицију обраде једне инструкције на следеће фазе:

- Припрема инструкције (*Fetch Instruction* – **FI**). Врши се учитавање следеће очекиване инструкције у бафер.
- Декодирање инструкције (*Decode Instruction* - **DI**). Одређују се кôд операције и спецификатори операнада.
- Израчунавање операнада (*Calculate Operands* - **CO**). Израчунава се ефективна адреса сваког изворног операнда. То израчунавање може да обухвата и адресирање са размештајем, регистарско индиректно адресирање или било који други облик адресирања.
- Припрема операнада (*Fetch Operands* - **FO**). Сваки од операнада се прибавља из меморије. Операнди из регистара не морају да се прибављају.
- Извршење инструкције (*Execute Instruction* - **EI**). Извршава се назначена операција и смешта резултат, ако га има, у одређену локацију одредишног операнда.
- Упис операнда (*Write Operand* - **WO**). Уписује се резултат у меморију.

Са оваквом декомпозицијом ће различите фазе бити уједначеније по трајању. Ради једноставније илустрације, замислимо да су наведене фазе сасвим једнаке по трајању. Слика 2.3 приказује како се са шестостепеним проточним системом време извршења 9 инструкција може смањити са 54 на 14 јединица времена.

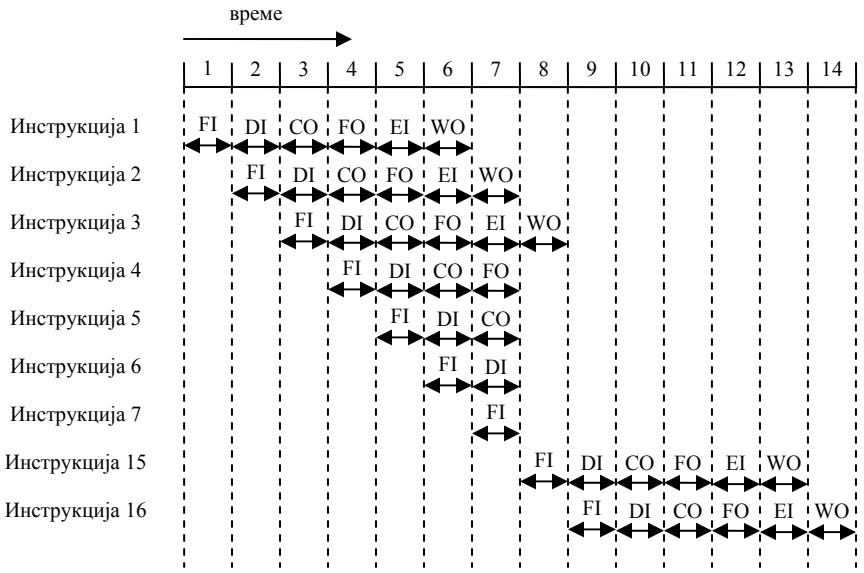


Сл. 2.3. Временски дијаграм рада проточног система.

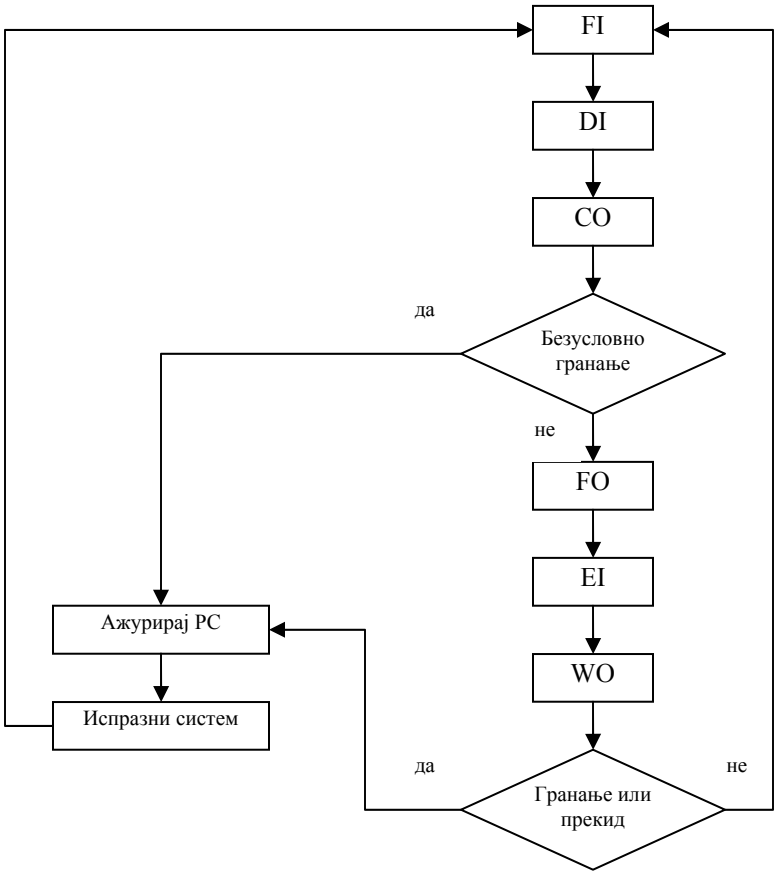
Дијаграм на слици 2.3 претпоставља да свака од инструкција пролази кроз свих шест степени проточног система. То, у ствари, није увек тако. Примера ради, инструкцији *load* не треба фаза WO. Ипак, ради једноставнијег хардвера у проточном систему, тајминг је тако постављен као да свака инструкција пролази све проточне степене. Осим тога, дијаграм је урађен тако као да се свака од фаза може извршавати паралелно, тј. да нема меморијских конфликата. Међутим, могуће је да фазе FI, FO и WO подразумевају и приступ меморији. Код већине система у пракси истовремени приступ меморији не би био дозвољен. Но, ако су потребне вредности већ у кешу, онда меморијски конфликти неће успорити проточни систем.

Неколико других фактора ограничава побољшање перформанси. Уколико шест проточних фаза нису истог трајања то ће изазвати извесно чекање које смо већ видели на примеру двостепеног система. Још један проблем представљају инструкције условног гранања које могу да обезвреде

неколико прибављених исинтрукција. Сличан непредвидљив догађај је прекид. Слика 2.4 илуструје ефекат условног гранања за програм са слике 2.3 али уз претпоставку да инструкција 3 представља условно гранање на инструкцију 15.



Сл. 2.4. Ефекат условног гранања на рад проточног система.



Сл. 2.5. Логика потребна да се обраде прекиди и гранања код шестостепеног проточног система.

Док се инструкција условног гранања извршава нема начина да се одреди која инструкција заиста треба да се изврши наредна. Систем једноставно прибавља следећу инструкцију у низу (инструкција 4) и наставља са радом. На слици 2.3 гранање се није догодило и имали смо пуну искоришћеност проточног система. На слици 2.4 гранање се догађа али се то не може знати пре окончања интервала 7. Сада проточни систем мора да се испразни обзиром да инструкције које су у њему нису оне које треба да се изврше. У току интервала 8 инструкција 15 улази у проточни систем и ниједна инструкција неће комплетирати своје извршење између интервала 9 и 12. То представља губитак у перформансама који се јавио јер нисмо могли да знамо хоће ли бити гранања или не. Слика 2.5. указује на логику потребну у проточном систему када се узму у обзир гранања и прекиди.

Овде се јављају и други проблеми који нису постојали у двостепеном проточном систему. Фаза CO може да зависи од садржаја регистара који је промењен од стране претходне инструкције која је још увек у проточном систему. И други слични регистарски или меморијски конфликти могу да се јаве. Систем мора да поседује логику која ће да разреши ове конфликте.

Из досадашње дискусије може да произађе да што је већи број степени у проточном систему, то је и брзина извршења већа. Међутим, постоје фактори који то оповргавају и који се морају узети у обзир:

1. У сваком степену постоји неки губитак који се састоји у преносу података од бафера до бафера и извршавању различитих функција припреме и испоруке. Овај губитак значајно продужава време извршења једне инструкције. Ово је нарочити значајно када су секвенцијалне инструкције логички зависне, било због честог гранања било због зависности у вези приступа меморији.
2. Износ управљачке логике који управља меморијским и регистарским зависностима и оптимизује употребу проточног система расте енормно са порастом броја степени. Ово води ка ситуацији где је логика која управља преносом између степени комплекснија од самих проточних степени којима управља.

2.2 Перформансе проточног система

Време циклуса τ проточног система је време потребно за напредовање једног скупа инструкција за један степен кроз систем. Оно може бити израчунато као:

$$\tau = \max[\tau_i] + d = \tau_m + d,\ 1 \le i \le k$$

где је τ_m максимално кашњење кроз степен, k број проточних степени и d кашњење лечева потребно за напредовање сигнала и података од једног до другог степена. У општем случају, кашњење d је еквивалентно периоду тактног сигнала док је $\tau_m >> d$.

Сада претпоставимо да се обрађује n инструкција без гранања. Укупно потребно време T_k да се изврши свих n инструкција је

$$T_k = [k + (n\text{-}1)]\,\tau.$$

Укупно k циклуса је потребно за довршење извршења прве инструкције и још n-1 за преосталих n-1 инструкција (време циклуса је једино једнако максималној вредности од τ када су сви проточни степени пуни, док на почетку оно може бити мање за првих неколико циклуса). Ово се лако може проверити на слици 2.3. Свих 9 инструкција је комплетирано у тренутку 14:

$$14 = [6+(9\text{-}1)]$$

Убрзање у односу на непроточни систем је

$$S_k = \frac{T_1}{T_k} = \frac{nk\tau}{[k+(n-1)]\tau} = \frac{nk}{k+(n-1)}.$$

Што је већи број проточних степени то је веће и потенцијално убрзање, али је већа и цена, кашњење између степени, а пражњење система услед гранања има већи негативан ефекат на перформансе.

2.3 Гранање код проточних система

Један од главних проблема код пројектовања проточног система је обезбеђивање стабилног тока инструкција у почетни степен система. Основну сметњу представљају инструкције условног

гранања, јер у току извршења инструкције није могуће одредити хоће ли се гранање извршити или не. Постоји више приступа за руковање инструкцијама условног гранања:

- Вишеструки токови.
- Прибављање циља гранања унапред.
- Бафер петљи.
- Предвиђање гранања.
- Закашњено гранање.

2.3.1 Вишеструки токови

Код обичног проточног система се плаћа цена за инструкције гранања, јер се мора одабрати једна од инструкције које ће се прибавити а при том се може направити погрешан избор. Приступ грубе силе је да се копира почетни део проточног система и омогући систему да прибави обе инструкције формирајући тако два низа.

Постоје два проблема код овог приступа:

- Постоји кашњење због надметања у приступу регистрима и меморији.
- Друга инструкција гранања може да уђе у систем (у оба тока) пре него што се одлука о гранању донесе. Свака нова инструкција гранања захтева додатне токове.

Упркос овим недостацима ова стратегија може да унапреди паралелизам. Примери машина које користе два проточна тока су IBM 370/168 и IBM 3033.

2.3.2 Прибављање циља гранања унапред.

Када се препозна условно гранање, циљ гранања се унапред прибавља, поред прибављања инструкције која следи иза гранања.

Циљна инструкција се памти док се инструкција не изврши. Ако се деси гранање, циљна инструкција је већ прибављена.

Овај приступ користи рачунар IBM 360/91.

2.3.3 Бафер петљи.

Ради се о малој брзој меморији којом управља степен припреме инструкције и која садржи *n* најскорије прибављених инструкција у секвенци.

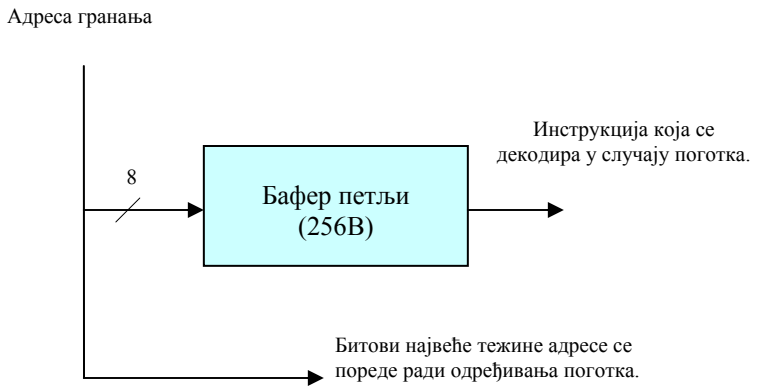
Ако се деси гранање, хардвер најпре проверава да ли је циљна инструкција у баферу. Ако јесте, следећа инструкција се прибавља из бафера.

Бафер петљи има три предности:

1. Када се користи прибављање унапред, бафер петљи ће садржати неке инструкције које су секвенцијално испред инструкције која се прибавља. Дакле, инструкције прибављене у секвенци су расположиве без коришћења уобичајеног времена за приступ меморији.
2. Ако је циљна инструкција свега неколико локација испред инструкције гранања, она је онда већ у баферу. То је чест случај код IF-THEN и IF-THEN-ELSE секвенци.
3. Ова стратегија је нарочито погодна за рад се петљама (отуда и назив бафер петљи), нарочито ако је бафер довољно велики да прими све инструкције из тела петље. У том случају се инструкције из тела петље прибављају из меморије само једном, приликом прве итерације. За наредне итерације, инструкције из тела петље су расположиве у баферу.

Бафер петљи је у суштини сличан кешу инструкција. Једина разлика је у томе што садржи инструкције у секвенци и много је мањег капацитета а тиме и јефтинији. На слици 2.6 приказан је пример једног бафера петљи. Ако бафер садржи 256B и користи се адресирање бајтова, 8 битова најмање тежине адресе се користе као индекс у баферу. Преостали битови се проверавају да би се одредило да ли се циљна инструкција гранања налази у оквиру бафера.

Пример машина које користе бафер петљи су неке од машина фирме CDC (Star-100, 6600, 7600) и CRAY-1. Специфични облик бафера петљи се среће и код микропроцесора Motorola 68010 за извршавање петљи од три инструкције.



Сл. 2.6. Бафер петљи.

2.3.4 Предвиђање гранања

Разне технике се користе за предвиђање хоће ли бити гранања или не. Најкоришћеније су:

- Предвиђање да се никад не врши гранање.
- Предвиђање да се увек врши гранање.
- Предвиђање на основу кôда операције.
- Коришћење прекидача има/нема гранања.
- Коришћење табеле историје гранања.

Прва три приступа су статичка. Они не зависе од историје извршења до тренутка када се наиђе на инструкцију условног гранања. Последња два приступа су динамичка и зависе од историје извршења.

Прва два приступа су најједноставнија. Ту се или претпоставља да се гранање увек не дешава и наставља се са прибављањем инструкција у секвенци, или се претпоставља да се гранање увек дешава и прибавља се циљна инструкција гранања. Motorola 68020 и VAX 11/780 користе приступ да се гранање никад не дешава. VAX 11/780 такође садржи средство за минимизацију ефекта погрешних одлука. Ако прибављање инструкције после инструкције гранања изазове промашај странице или нарушавање заштићеног дела меморије, процесор обуставља прибављање док не постане сигурно да инструкција треба да се прибави.

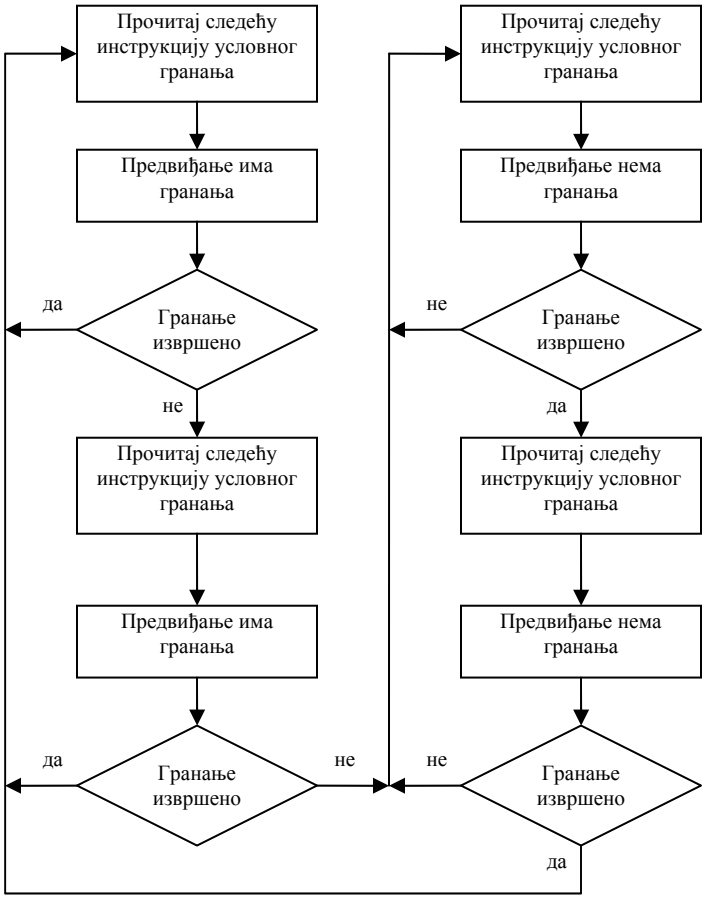
Анализа програма показала је да се условно гранање заиста одиграва у више од 50% случајева, па је, при истој цени прибављања једне од инструкција, боље прибављати циљну инструкцију гранања. Са друге стране, код машина са страничењем постоји већа вероватноћа да ће прибаљање циљне инструкције гранања изазвати промашај странице него што је то случај са прибављањем следеће инструкције у секвенци. Потребно је увести некакав механизам који би смањио губитке услед оваквог предвиђања.

Најзад, трећи од статичких приступа претпоставља доношење одлуке на основу кôда операције. Предвиђање на основу кôда операције састоји се у томе да процесор претпоставља да ће се гранање заиста обавити за неке кôдове операција док за друге не. Анализе показују да се у више од 75% случајева овакво предвиђање показује успешним.

Динамичке стратегије гранања покушавају да побољшају тачност предвиђања тако што памте историју инструкција условног гранања у програму. Примера ради, један или више битова се могу придружити свакој инструкцији условног гранања да опишу недавну прошлост инструкције. Ови битови се називају *прекидач има/нема гранања* и усмеравају процесор у доношењу одлуке када се наредни пут та инструкција буде извршавала. Обично ови битови нису придружени инструкцији у главној меморији већ се држе у привременој брзој меморији. Једна могућност је да се ови битови придруже свакој од инструкција условног гранања у кешу. Када се изврши замена инструкције (замена блока који је садржи) њена историја је изгубљена. Друга могућност је да се креира мала табела недавно извршених инструкција условног гранања са једним или више битова по свакој ставци. Процесор таквој табели може да приступа асоцијативно, као што је случај код кеша, или да користи ниже битове адресе инструкције гранања.

Са једним битом се једино може запамтити да ли је последње извршење дотичне инструкције условног гранања заиста довело до гранања или не. Недостатак код коришћења само једног бита се јавља у случају инструкције условног гранања које се скоро увек догађа, као што је то код петљи. Са само једним битом историје грешка у предвиђању ће се јавити два пута за свако извршавање петље: једном на уласку у петљу, једном приликом њеног напуштања.

Ако се користе два бита онда они могу да упамте резултат два последња извршења инструкције којој су придружени или да упамте стање на неки други начин. Слика 2.7 приказује типичан приступ. Претпоставимо да алгоритам почиње од горњег левог угла дијаграма тока. Док год се гранање код узастопних инструкција гранања заиста догађа процес одлучивања даје предвиђање да ће се наредно гранање заиста и догодити. Ако се једно предвиђање покаже нетачним, алгоритам наставља да предвиђа да ће се следеће гранање догодити. Једино два узастопна погрешна предвиђања (тј. два узастопна гранања која се нису догодила) преводе алгоритам у десну страну дијаграма тока.



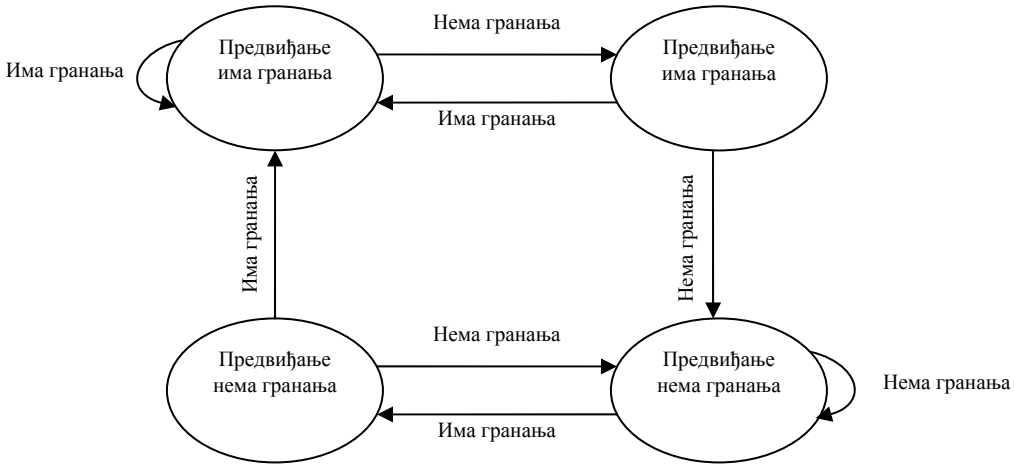
Сл. 2.7. Дијаграм тока предвиђања гранања.

После тога, алгоритам ће предвиђати да се гранање неће догодити све до појаве два узастопна погрешна предвиђања. Према томе, за промену карактера предвиђања потребне су две узастопне погрешне одлуке.

Процес одлучивања може се представити компактније у облику графа коначног аутомата који је приказан на слици 2.8.

Коришћење битова историје гранања има један недостатак. Ако је услов гранања испуњен циљна инструкција се не може прибавити пре него се декодира циљна адреса. Већа ефикасност се може постићи ако би се прибављање инструкције започело чим се донесе одлука о гранању. Да би

ово било могуће мора да се памти више информација а та шема је позната као бафер циљева гранања или табела историје гранања.



Сл. 2.8. Дијаграм стања предвиђања гранања.

Табела историје гранања је мала кеш меморија придружена степену припреме инструкције. Свака ставка у табели се састоји од три елемента:

- адресе инструкције гранања,
- извесног броја битова историје који памте стање употребе те инструкције, и
- информације о циљној инструкцији.

У већини имплементација треће поље садржи адресу циљне инструкције. Друга могућност је да то поље садржи циљну инструкцију. Памћење циљне адресе даје мању табелу али дуже време припреме инструкције него када се памти сама циљна инструкција!

Последња динамичка шема састоји се од преуређивања инструкција програма тако да се инструкција гранања јавља касније него што је стварна намера (закашњено гранање). О закашњеном гранању биће више речи касније.

Литература

[1] W. Stallings, *Computer Organization and Architecture*, 6/e, Prentice Hall, 2003.

3 RISC процесори

Још од развоја првог рачунара са запамћеним програмом, негде око 1950. године, било је мало стварних иновација у области архитектуре и организације рачунара. Неке од главних иновација у току развоја рачунарства су:

- **Концепт фамилије.** Увео га је IBM са својим System/360 током 1964. године, а убрзо га је пратио и DEC са својим PDP-8. Концепт фамилије раздваја архитектуру машине од њене имплементације. Тржишту се нуди скуп рачунара са различитим цена/перформанса карактеристикама, који представља исту архитектуру. Разлика у цени и перформансама јавља се због различите имплементације исте архитектуре.
- **Микропрограмирана управљачка јединица.** Предложио ју је Морис Вилкс 1951. године, а први пут је уведена у линију рачунара IBM System/360 1964. године. Микропрограмирање олакшава посао пројектовања и имплементације управљачке јединице и пружа подршку концепту фамилије.
- **Кеш меморија**. Прва комерцијална употреба била је код рачунара IBM S/360 Model 5 1968. године. Увођењем кеш меморија драстично су поправљене перформансе.
- **Проточност.** Начин да се уведе паралелизам у програм на машинском језику који је у суштини секвенцијалан. Примери овога су проточност инструкција и векторска обрада.
- **Вишеструки процесори.** Ова категорија покрива известан број различитих органинзација и циљева.

Овој листи треба додати најинтересантнију и потенцијално најважнију иновацију: RISC (*Reduced Instruction Set Computer*) архитектуру. RISC архитектура представља драматично скретање од историјских трендова у архитектури процесора. Анализа RISC архитектуре фокусира многа важна питања организације и архитектуре рачунара.

Мада су RISC системи дефинисани и пројектовани на различите начине, већина њих дели следеће кључне карактеристике:

- Велики број регистара опште намене (32 и више) и/или употреба компилаторских технологија за оптимизацију употребе регистара.
- Ограничен и једноставан скуп инструкција.
- Нагласак на оптимизацији проточности инструкција.

3.1 Употреба великог броја регистара

Пожељан је брз приступ операндима. Како је велики удео наредби доделе у програмима, а многе од њих су сасвим једноставне, и како постоји значајан број приступа операндима који су највећем броју случајева локални скалари, код RISC процесора се намеће значајно ослањање на велики скуп регистара. Разлог за овакав приступ је очигледан обзиром да су регистри процесора бржа меморија и од кеша и од главне меморије. Релативно мали број регистара (а он је много мањи од броја меморијских локација, чак и у случају RISC процесора) повлачи и краће адресе. На тај начин ова стратегија омогућује фреквентнији приступ операндима који се држе у регистрима а тиме се смањује број операција типа регистар-меморија.

Да би се ово омогућило постоје два приступа: софтверски и хардверски. Софтверски приступ се ослања на рад компилатора који треба да максимизирају коришћење регистара. Компилатор ће покушати да додели регистре оним променљивама које ће се највише користити у датом периоду времена. Овај приступ захтева употребу софостицираних алгоритама за анализу програма.

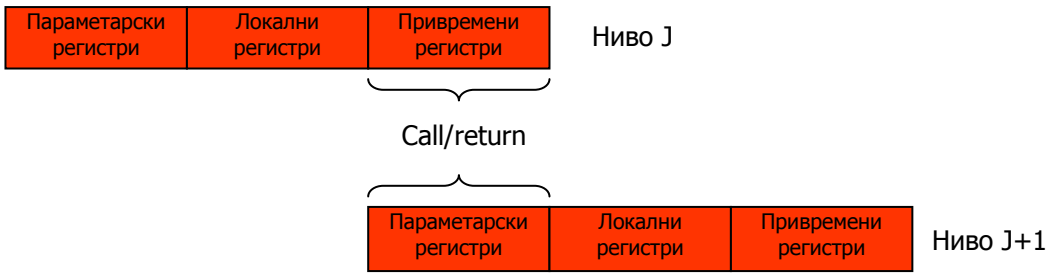
Хардверски приступ састоји је једноставно у коришћењу већег броја регистара како би се већи брј променљивих што дуже држао у њима.

3.1.1 Регистарски прозори

Већина операнада су локални скалари па је очигледан приступ њихово смештање у регистар уз неколико регистара резервисаних за глобалне променљиве. Проблем је што се дефиниција локалног мења са сваким позивом процедуре или повратком из процедуре а те операције се јављају често.

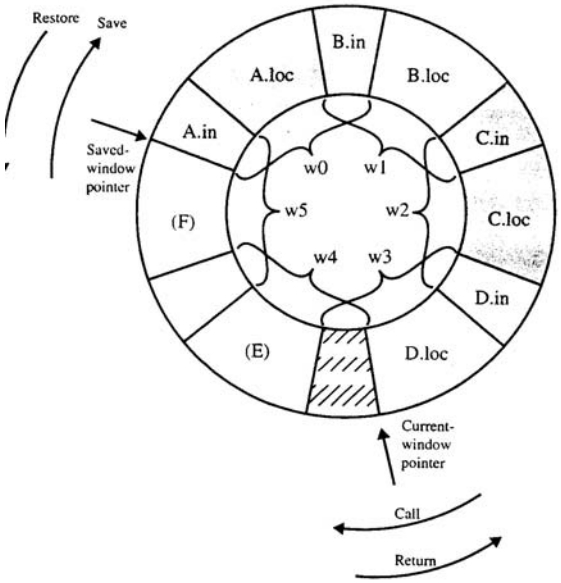
При сваком позиву се локалне променљиве морају из регистара пребацити у меморију како би се омогућило коришћење тих регистара у позваном програму. Осим овога морају се преносити параметри. При повратку, променљиве морају поново да се напуне у регистре као и да се позивајућем програму врати резултат позваног програма.

Решење се заснива на томе да типична процедура захтева свега неколико параметара и локалних променљивих. Осим тога, дубина позива процедура се креће у релативно уском опсегу. Да би се ова својства искористила користи се више малих скупова регистара, који се називају *регистарски прозори*, при чему се сваки додељује различитој процедури. Позив процедуре аутоматски пребацује процесор да користи други регистарски прозор фиксне величине уместо да врши памћење садржаја регистара у меморију. Прозори суседних процедура се преклапају како би се омогућио пренос параметара (слика 3.1).



Сл. 3.1. Преклапање регистарских прозора.

Да би се обрадила свака могућа комбинација позива и повратака, број регистарских прозора мора да буде неограничен! Уместо тога, регистарски прозори се користе тако да садрже неколико најскоријих активација процедура. Старије активације морају да се запамте у меморију а обнављају се када се смањи дубина позива. Према томе, стварна организација регистарског поља је кружни бафер са преклапајућим прозорима.



Сл. 3.2. Организација кружног бафера са преклапајућим прозорима.

На слици 3.2 описан је кружни бафер са шест прозора. Бафер је напуњен до дубине 4 (A позива B, B позива C, C позива D) при чему је активна процедура D. Указатељ текућег прозора (CWP) указује на прозор тренутно активне процедуре. Регистарске референце у оквиру машинских инструкција померене су за вредност тог поинтера како би се одредили стварни физички регистри. Указатељ запамћеног прозора идентификује прозор који је најскорије запамћен у меморију. Ако процедура D сада позове процедуру E, аргументи за E се смештају у привремен регистре прозора за процедуру D (преклапање између прозора w3 и прозора w4) и CWP се помера на наредни прозор.

Ако процедура E потом изврши позив процедуре F, то неће моћи да се уради обзиром на тренутно стање у баферу. Прозор за F би се преклопио са прозором за A, тачније привремени регистри за F преклопили би параметарске регистре за A (A.in). Према томе, када се CWP инкрементира (по модулу 6, за овај пример) он ће постати једнак указатељу SWP, што ће изазвати прекид ради памћења садржаја регистарског прозора који одговара позиву процедуре A. Неопходно је памтити само прва два дела регистарског прозора (A.in и A.loc). Потом се SWP инкрементира и наставља са позивом процедуре F. Сличан прекид се јавља приликом повратка из процедуре. На пример, пошто је извршена активација процедуре F низ повратака ће довести у једном моменту до повратка из процедуре B у позивајућу процедуру A. CWP се декрементира и постаје једнак као SWP. Ово ће изазвати прекид који за последицу има обнављање регистарског прозора за A.

Видимо да регистарско поље са N прозора може да прихвати N-1 активацију процедуре. Вредност за N не мора бити велика. Примера ради, испитивања су утврдила да је са 8 прозора памћење или обнављање регистарских прозора потребно само у 1% позива или повратака.

3.1.2 Глобалне променљиве

Шема са прозорима је ефикасна за памћење локалних скалара у регистре али не покрива памћење глобалних променљивих којима приступа више процедура. Намећу се две могућности. Прва је да компилатор глобалним променљивама додељује меморијске локације. Тако ће све машинске инструкције које референцирају ове променљиве користити један операнд у меморији. Ово је директан приступ и са становишта хардвера и са становишта софтвера. Међутим, код фреквентног приступа глобалним променљивама овај приступ је веома неефикасан.

Алтернатива је да се угради скуп глобалних регистара у процесор. Ови регистри су фиксирани по броју и доступни су свим процедурама. Да би се поједноставио формат инструкција, може се узети да су, рецимо регистри 0 до 7 глобални, док су регистри 8 до 31 намењени за регистарске прозоре. Да би се ова разлика у регистарском адресирању реализовала потребан је додатни хардвер. Осим тога компилатор мора да одлучи које глобалне променљиве треба доделити регистрима.

3.1.3 Поређење употребе великог регистарског поља и кеш меморије

Регистарско поље, организовано у прозоре, ради као мали, брзи бафер за држање подскупа свих променљивих које ће се највише користити. Са те тачке гледишта, регистарско поље подсећа на кеш меморију, мада је много брже. Питање које се намеће је када је једноставније и боље користити кеш и мало традиционално регистарско поље.

Табела 3.1 пореди карактеристике ова два приступа. Регистарско поље засновано на прозорима садржи све локалне променљиве (осим у ретком случају преклапања прозора) најскоријих N-1 позива процедура. Кеш садржи избор недавно коришћених скаларних променљивих. Регистарско поље штеди на времену, јер задржава све локалне променљиве. Са друге стране, кеш користи простор ефикасније, јер динамички реагује на ситуацију. Даље, кеш у општем случају третира све меморијске референце на сличан начин, укључујући и инструкције и друге типове података.

Регистарско поље може неефикасно да користи простор, јер није увек за све процедуре потребан читав простор прозора који им је додељен. Кеш пати од друге врсте неефикасности: подаци се читају у блоковима а можда нису сви потребни.

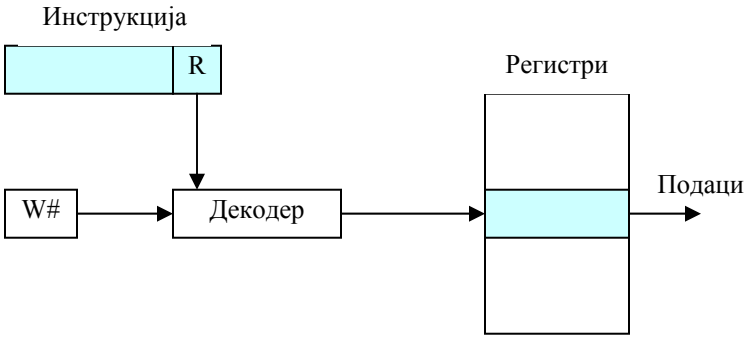
Кеш је у стању да ради са глобалним као и са локалним променљивама. Обично има доста глобалних скалара, али се само мањи број њих често користи. Кеш динамички открива те променљиве и садржи их. Ако је регистарско поље засновано на прозорима опремљено глобалним регистрима оно такође може да садржи неке глобалне скаларе, али је компилатору тешко да одреди који ће се често користити.

Код регистарског поља пренос података између регистара и меморије одређен је дубином угњежђавања процедура. Како се дубина гњежђења креће у релативно уском опсегу, коришћење меморије није толико често. Већина кеш меморија је скупно асоцијативна са малом величином скупа. Према томе, опасност је да ће други подаци или инструкције потиснути често коришћене промењиве.

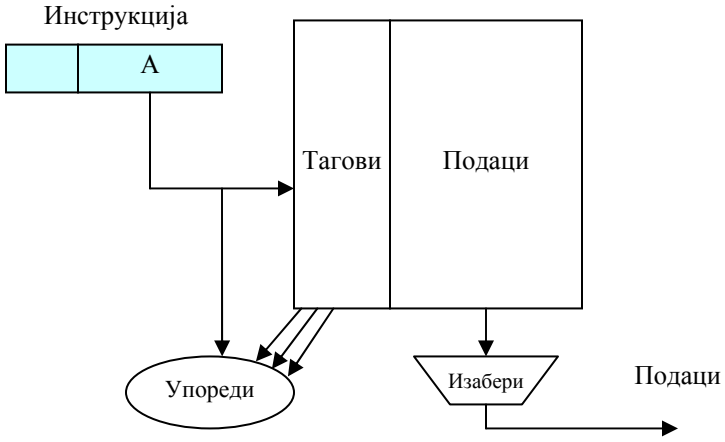
Таб. 3.1. Карактеристике организација са великим регистарским пољем и са кешом.

Велико регистарско поље	Кеш
Сви локални скалари	Најскорије коришћени скалари
Поједине промењиве	Блокови у меморији
Глобалне промењиве додељене од стране компилатора	Најскорије коришћене глобалне промењиве
Памћење и обнављање засновано на дубини гњежђења процедура	Памћење и обнављање засновано на алгоритмима замене
Регистарско адресирање	Меморијско адресирање

Из претходне дискусије није јасно који од ова два приступа је бољи. Међутим једна карактеристика по којој је регистарски приступ супериоран је да је систем заснован на кешу осетно спорији. Ова разлика се исказује у начину адресирања ова два приступа. Слика 3.3 илуструје ову разлику.



(а) Регистарско поље засновано на прозорима.



(б) Кеш.

Сл. 3.3. Референцирање скалара.

3.2 RISC архитектура

Код CISC машина је приметан тренд обогаћивања скупа инструкција који садржи већи број сложенијих инструкција. Два главна разлога су мотивисала настанак овог тренда: жеља да се поједностави компилација и жеља да се побољшају перформансе. Ови разлози су програмере водили ка HLL а архитекте да пројектују машине које боље подржавају HLL.

Први разлог је очигледан: што је више асемблерских инструкција које подсећају на исказе виших програмских језика, посао компилатора је лакши. Међутим истраживања су показала да је често тешко искористити сложене машинске инструкције, јер компилатор мора најпре да пронађе случајеве у којима оне тачно одговарају конструкцијама. Задатак оптимизације генерисаног кôда ради минимизације његове величине, смањења броја инструкција и побољшавања проточности је много тежи са сложеним скупом инструкција. Уосталом, емпиријски је потврђено да је већина инструкција у компилованом програму релативно једноставна.

Други разлог је што CISC машине дају мање и брже програме. Међутим, иако ће CISC програм најчешће имати мање инструкција од програма за RISC машину, то не значи да ће и заузимати мање битова. Осим тога, сложеније инструкције захтевају сложенију управљачку јединицу као и већу микропрограмску меморију што утиче на време извршења.

3.2.1 Карактеристике RISC архитектура

И поред различитих приступа код RISC архитектура, следеће карактеристике су заједничке за све њих:

- Једна инструкција по машинском циклусу.
- Операције типа регистар-регистар.
- Једноставни адресни начини рада.
- Једноставан формат инструкција.

Прва карактеристика је да се једна машинска инструкција изврашава за један машински циклус. Под машинским циклусом подразумевамо време потребно за припрему два регистарска операнда, обављање ALU операције и смештање резултата у неки од регистара. Према томе, RISC машине не би требало да буду компликованије од микроинструкција код CISC машина, а тиме би се и извршавале истом брзином. Са једноставним, једноциклусним инструкцијама постоји мала или никаква потреба за микрокôдом. Такве инструкције треба да се извршавају брже него одговарајуће инструкције код других машина, обзиром да не постоји потреба да се приступа микропрограмској меморији током извршења инструкције.

Друга карактеристика је да је већина операција типа регистар-регистар, уз једине две једноставне операције, типа *load* и *store*, које приступају меморији. Ова карактерситика поједностављује скуп инструкција а тиме и управљачку јединицу. Друга повољна околност је да таква архитектура стимулише оптимизацију коришћења регистара па су најчешће коришћени операнди увек у регистарској меморији.

Трећа карактеристика односи се на употребу једноставних адресних начина рада. Већина RISC инструкција користи једноставни регистарски формат. Додатни начини рада, као што је индиректно са размештајем или PC релативно, се такође могу јавити. Сложенији адресни начини рада могу се у софтверу постићи помоћу једноставнијих. И ова карактеристика поједностављује скуп инструкција и управљачку јединицу.

Најзад, заједничка карактеристика RISC машина је и једноставан формат инструкција. У општем случају, само један или свега неколико формата инструкција постоји код RISC машина. Дужина инструкција је фиксирана и поравната на границе речи. Локације поља у оквиру инструкција, нарочито кôда операције, су фикиране. Постоји више предности оваквог приступа. Уз фиксирана поља декодирање кôда операције и приступ регистарским операндима могу се јавити симултано. Такође, једноставнији формат значи и једноставнију управљачку јединицу. Припрема инструкција је поједностављена јер се увек добављају инструкције дужине једне речи. Поравнање на речи такође значи да једна инструкција не може да пређе границе странице.

3.3 Проточност код RISC архитектура

3.3.1 Проточност код обичних инструкција

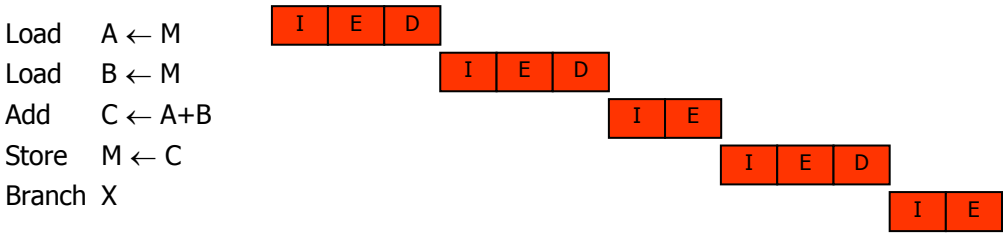
Већина инструкција код RISC архитектура је регистарског типа и захтева две фазе:

- **I** : Припрема инструкције.
- **E** : Извршење (обавља се ALU операција са регистарским улазом и излазом).

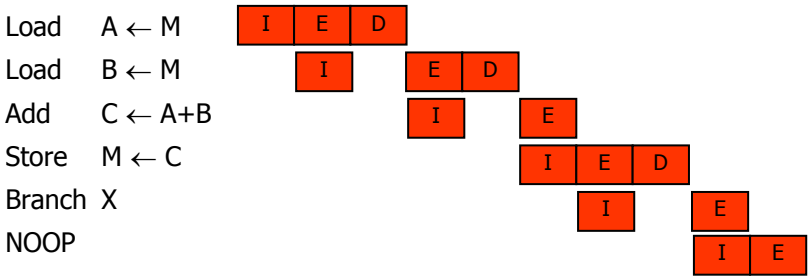
Инструкције *load* и *store* захтевају три фазе:

- **I** : Припрема инструкције.
- **E** : Извршење – израчунавају се меморијске адресе.
- **D** : Обраћање меморији – операције регистар у меморију или меморија у регистар.

Слика 3.4(а) описује тајминг за секвенцу инструкција када се не користи проточност, а на слици 3.4.(б) приказан је случај двостепеног проточног система где се фазе I и E различитих инструкција извршавају симултано.



(а) Секвенцијално извршење.



(а) Двостепени проточни систем.

Сл. 3.4. Ефекти проточности.

Два проблема спречавају постизање максималног двоструког убрзања:

- Претпоставка да је могућ само једноструки приступ меморији.
- Инструкција гранања ремети секвенцијални ток извршења. Да би се извршило одговарајуће прилагођавање компилатор убацује инструкцију NOOP.

Проточност се може побољшати дозвољавањем два приступа меморији по степену (слика 3.5). Сада се и до три инструкције могу преклапати. И у овом случају инструкција гранања не дозвољава постизање максималног убрзања које износи 3. Приметимо и да зависности по подацима такође имају свој утицај. Ако инструкцији треба операнд који треба да буде измењен неком од претходних инструкција, неопходно је одређено кашњење. И у овом случај се то може постићи убацивањем безефектних инструкција.

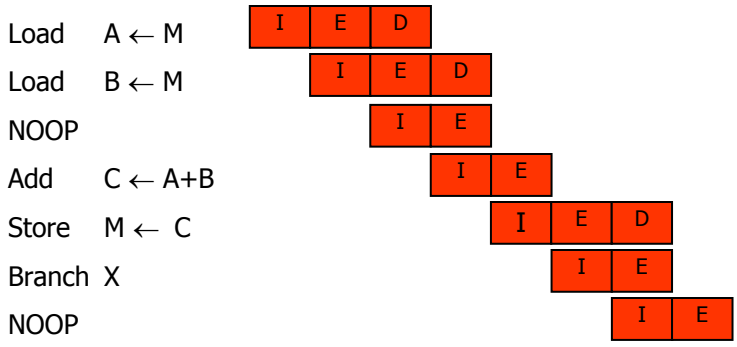
Проточни систем ће радити боље ако су све три фазе приближно једнаког трајања. Како фаза **E** обухвата рад ALU она је обично дужа од осталих, па је згодно да се подели на две фазе:

E_1: Читање регистара.

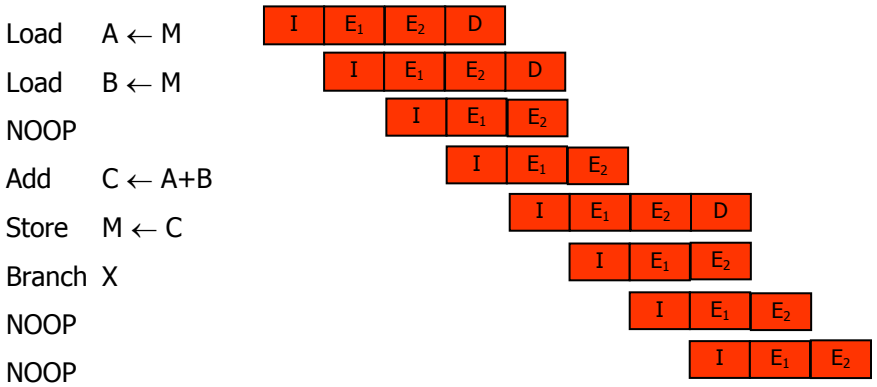
E_2: Рад ALU и упис у регистар.

Због једноставности и регуларности скупа инструкција код RISC машина, лако се пројектује подела инструкције на три или четири фазе. На слици 3.6. приказан је резултат када се уведе

четворостепени проточни систем. Максимално потенцијално убрзање је 4, јер се и до четири инструкције могу преклапати. Запазимо да још увек постоји потреба за безефектним инструкцијама због гранања и зависности по подацима.



Сл. 3.5. Ефекат проточности код тростепеног система.



Сл. 3.6. Ефекат проточности код четворостепеног система.

3.3.2 Оптимизација проточности

Зависности по подацима и гранања смањују брзину извршења и не дозвољавају да се до максимума искористи потенцијал проточности. Да би се ово компензовало развијене су технике за реорганизацију кôда.

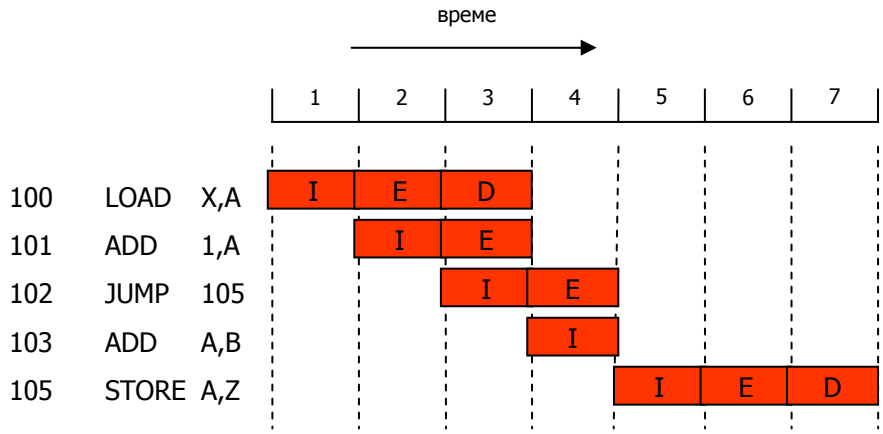
Размотримо најпре инструкције гранања. *Закашњено гранање* је техника код које гранање нема ефекта док се не изврши следећа инструкција. Локација инструкције која следи после гранања назива се *слот кашњења*. Ова техника је описана у табели 3.2. У колони "нормално гранање" приказане су инструкције неког асембелерског програма. После извршења инструкције са адресом 102, следећа инструкција је 105. Да би се нормализовао доток у проточни систем убацује се безефектна инструкција NOOP после инструкције гранања. Међутим, боље перформансе се постижу када инструкције 101 и 102 замене места.

Слике 3.7 – 3.9 приказују добијене резултате. Слика 3.7 приказује традиционални приступ проточности. Инструкција JUMP се прибавља у интревалу 3. У интервалу 4 се она извршава док се у исто време прибавља инструкција са адресом 103 (инструкција ADD). Због тога што се јавила инструкција JUMP која мења садржај програмског бројача, из проточног система мора да се испразни инструкција 103 и да се у интервалу 5 прибави инструкција 105 која је циљ гранања. На слици 3.8 приказан је ефекат у проточном систему типичне RISC машине. Тајминг је исти, али због убацивања NOOP инструкције не треба нам посебна логика за пражњење система. На слици 3.9 приказан је ефекат закашњеног гранања. Инструкција JUMP се прибавља у интервалу 2, пре инструкције ADD која се прибавља у интервалу 3. Приметимо да је инструкција ADD прибављена

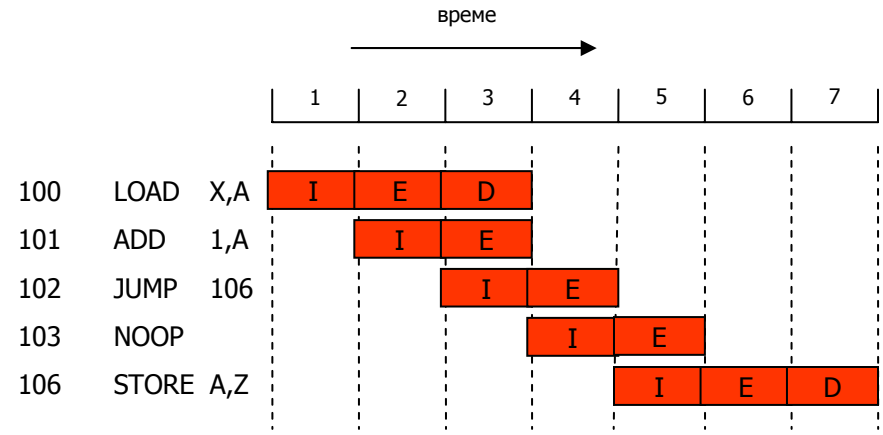
пре него што је инструкција JUMP имала прилику да измени програмски бројач. Тако се током интервала 4 инструкција ADD извршава док се прибавља инструкција 105. Оригинална семантика програма је задржана али је један циклус мање потрошен на извршење.

Таб. 3.2. Нормално и закашњено гранање.

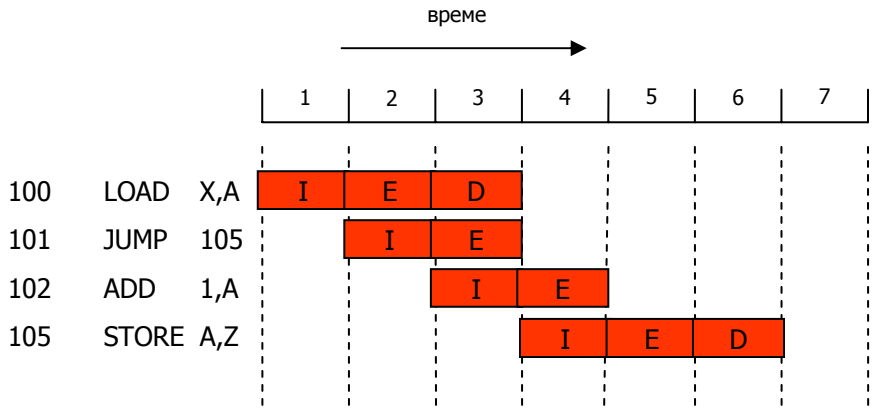
Адреса	Нормално гранање		Закашњено гранање		Оптимизирано закашњено гранање	
100	LOAD	X,A	LOAD	X,A	LOAD	X,A
101	ADD	1,A	ADD	1,A	JUMP	105
102	JUMP	105	JUMP	106	ADD	1,A
103	ADD	A,B	NOOP		ADD	A,B
104	SUB	C,B	ADD	A,B	SUB	C,B
105	STORE	A,Z	SUB	C,B	STORE	A,Z
106			STORE	A,Z		



Сл. 3.7. Традиционална проточност.



Сл. 3.8. RISC проточност са убаченим NOOP.



Сл. 3.9. Измењен редослед инструкција.

Измена редоследа инструкција успешно ради за безусловна гранања, позиве процедура и повратке из процедура. За условно гранање се овај поступак не може тек тако применити. Ако услов који се тестира ради гранања може да буде измењен инструкцијом која непосредно претходи, онда компилатор мора да се уздржи од измене редоследа и да убаци NOOP. Иначе, компилатор може да проба са убацивањем корисне инструкције после инструкције гранања.

Слична тактика, *закашњено пуњење*, може се применити код инструкције LOAD.

Литература

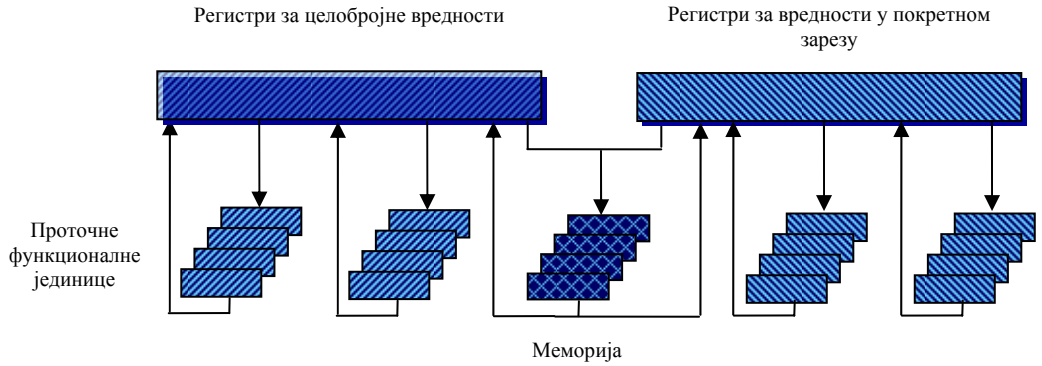
[1] W. Stallings, *Computer Organization and Architecture*, 6/e, Prentice Hall, 2003.

4 Паралелизам на нивоу инструкција и суперскаларни процесори

Суперскаларна имплементација архитектуре процесора претпоставља да се уобичајене инструкције, као што су аритметичке, *load*, *store*, гранања и др., могу започети симултано како би се независно извршавале. Таква имплементација повлачи настанак многих сложених пројектантских проблема везаних за проточност инструкција. Суперскаларно пројектовање је чврсто ступило на сцену са појавом RISC архитектура. Мада поједностављени скуп инструкција код RISC машина одговара примени суперскаларних техника, суперскаларни приступ може се користити и код CISC и код RISC архитектура.

4.1 Суперскалане и суперпроточне машине

Појам *суперскаларни* јавио се 1987. године и односи се на машину пројектовану да унапреди перформансе извршења скаларних инструкција. Суштина суперскаларног приступа је у способности независног извршења инструкција у различитим проточним системима. Овај концепт се даље може искористити тако што се дозвољава извршавање инструкција у редоследу различитом од оног у програму. На слици 4.1 илустрован је суперскаларни приступ.



Сл. 4.1. Општа суперскаларна организација

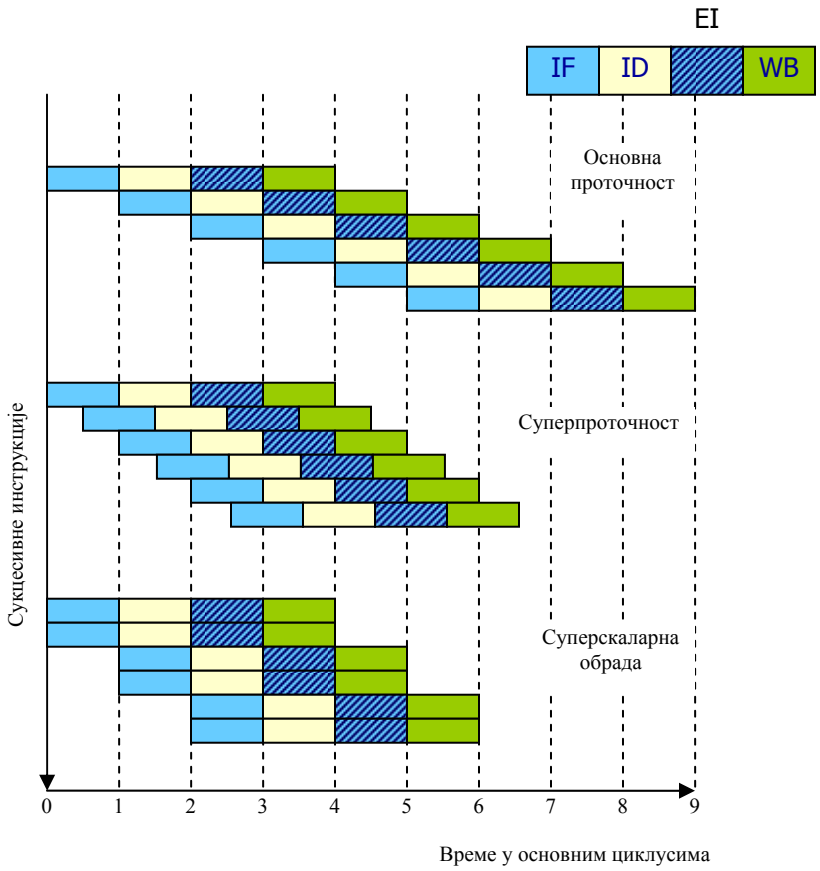
Алтернативни приступ за постизање бољих перформанси је *суперпроточност*. Овај појам је у употреби од 1988. године. Суперпроточност користи чињеницу да многи проточни степени обављају задатке који захтевају мање од једног тактног периода. Ако се удвостручи фреквенца интерног тактног сигнала могу се обављати по два задатка у једном тактном периоду.

На слици 4.2 приказано је поређење суперпроточног и суперскаларног извршења у односу на традиционалну проточну обраду. Основни проточни систем, који је овде дат ради поређења, издаје по једну инструкцију по циклусу и свака фаза инструкције се извршава за један циклус. Постоје четири проточна степена: припрема инструкције (IF), декодирање инструкције (ID), извршење инструкције (EI) и упис резултата (WB). Иако се извршење инструкција преклапа, само по једна инструкција може бити у истој фази, тј. у истом проточном степену.

У следећем делу слике 4.2 приказано је како суперпроточна имплементација може да извршава две проточне фазе по циклусу. Алтернативни начин да се ово сагледа је да претпоставимо да је функција коју обавља сваки од проточних степени подељена на два дела који се не преклапају а сваки од њих се извршава за пола циклуса. За суперпроточну имплементацију која ради на овакав

начин кажемо да је степена 2. Најзад, у најнижем делу дијаграма је приказана суперскаларна имплементација која може паралелно да извршава по два примерка сваке фазе инструкције. Могуће су имплементације већег степена од 2, како код суперпроточних, тако и код суперскаларних машина.

И суперскаларна и суперпроточна имплементација описане на слици 4.2 имају у стабилном стању исти број инструкција које се истовремено извршавају. Суперпроточни процесор заостаје за суперскаларним процесором на почетку програма и при сваком гранању.



Сл. 4.2. Поређење суперскаларног и суперпроточног приступа.

4.2 Ограничења суперскаларности

Суперскаларни приступ зависи од могућности да се више инструкција извршава паралелно. Појам *паралелизам на нивоу инструкција* односи се на степен до кога се, у просеку, инструкције програма могу извршавати паралелно. Користи се комбинација техника заснованих на компилатору и хардверских техника да би се максимизирао паралелизам на нивоу инструкција. Пре него што размотримо те технике које се користе код суперскаларних машина упознајмо се са основним ограмичењима паралелизма са којима се такви системи срећу:

- Праве зависности по подацима.
- Процедуралне зависности.
- Конфликти ресурса.
- Излазне зависности.
- Антизависности.

Најпре ћемо се упознати са прва три типа ограничења а нешто касније и са преостала два.

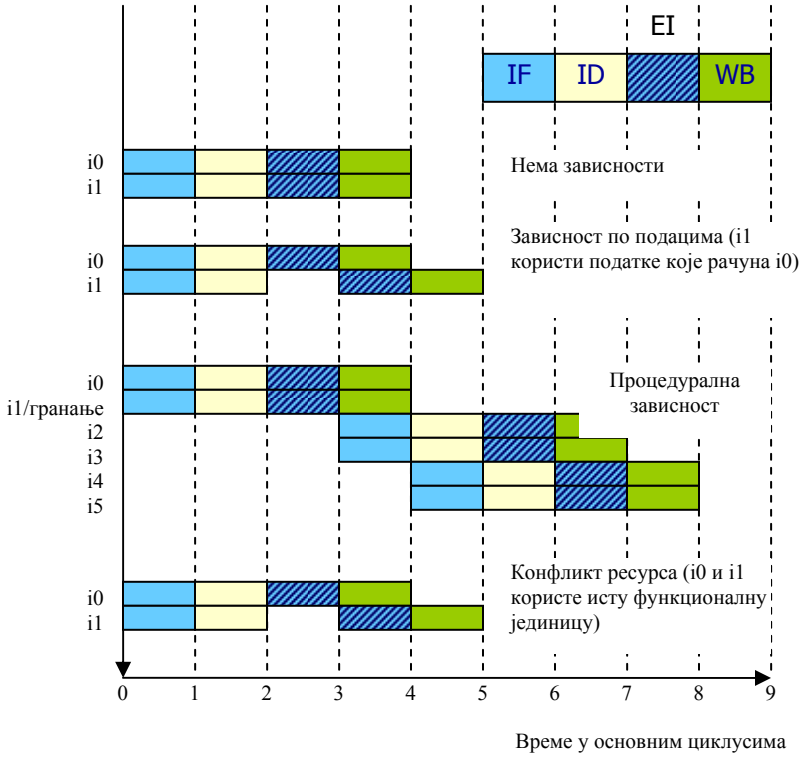
4.2.1 Праве зависности по подацима

Посматрајмо следећу секвенцу:

```
add     r1, r2    ; r1 ← r1+r2
move    r3, r1    ; r3 ← r1
```

Друга инструкција може да се прибави али не може да се изврши док се не изврши прва. Разлог овоме је што другој инструкцији требају подаци који су резултат прве инструкције. Ова ситуација се зове права зависност по подацима (још и зависност тока или *write-read* зависност).

На слици 4.3. илустроване су зависности код суперскаларне машине степена 2. Без зависности по две инструкције могу да буде паралелно прибављене и извршене. Ако постоје зависности по подацима између прве и друге инструкције, тада друга инструкција мора да одложи своје извршење онолико циклуса колико је неопходно да се отклони зависност. У општем случају, инструкција мора да се одложи онолико циклуса колико је потребно да се све вредности које су за њу улазне не произведу, тј. израчунају.



Сл. 4.3. Ефекти зависности.

У обичном скаларном проточном систему пређашња секвенца не би изазвала никакво кашњење. Међутим, размотримо следећу секвенцу где имамо инструкцију која има операнд у меморији уместо у регистру:

```
load    r1, eff   ; r1 ← M[eff]
move    r3, r1    ; r3 ← r1
```

Типичном RISC процесору треба 2 или више циклуса за читање меморије. Један од начина да се компензује кашњење је да компилатор преуреди инструкције тако да једна или више наредних инструкција које не зависе од читања из меморије могу уђу у проточни систем. Међутим ова шема је мање ефикасна код суперскаларних процесора. Независне инструкције које се извршавају током

извршења инструкције *load* ће се највероватније извршити током првог циклуса, остављајући процесор неупосленим док се читање из меморије не заврши.

4.2.2 Процедуралне зависности

Као што је већ речено, гранања компликују рад проточног система. Инструкције које следе иза инструкције гранања (без обзира да ли се оно заиста дешава или не) су процедурално зависне од инструкције гранања и не могу се извршити док се не изврши гранање. На слици 4.3 видимо илустрацију ефекта гранања на суперскаларни проточни систем степена 2.

Већ смо видели да овај тип зависности такође погађа и скаларни проточни систем. Код суперскаларних система овај проблем је још израженији, јер се са сваким одлагањем губи више.

Код инструкција променљиве дужине овај проблем је и већи, па је то разлог што се суперскаларност највише среће код RISC процесора који имају фиксну дужину инструкције.

4.2.3 Конфликти ресурса

Конфликт ресурса настаје када две или више инструкција захтева исти ресурс истовремено. Примери ресурса су меморија, кеш, магистрале, поља регистара и функционалне јединице.

Имајући у виду проточност може се рећи да се ова врста конфликата слично понаша као и зависности по подацима (види слику 4.3). Ипак, постоје и извесне разлике. Најпре, конфликти ресурса се могу ублажити повећањем броја ресурса у систему, док се зависности по подациме не могу избећи. Осим тога, ако је некој операцији потребно дуго време да би се обавила, конфликти ресурса се могу минимизирати ако се одговарајућа функционална јединица учини проточном.

4.3 Проблеми у пројектовању

4.3.1 Паралелизам на нивоу инструкција и машински паралелизам

Иако су ова два концепта блиска, између њих постоји важна разлика. *Паралелизам на нивоу инструкција* (ILP) постоји када су инструкције у секвенци независне и могу се извршавати паралелно преклапањем. Као пример за ILP концепт посматрајмо следеће две кôдне секвенце.

Load	R1 ← R2	Add	R3 ← R3, “1”
Add	R3 ← R3, “1”	Add	R4 ← R3, R2
Add	R4 ← R4, R2	Store	[R4] ← R0

Три инструкције са леве стране су независне и теоретски све три могу да се извршавају паралалено. Насупорт њима, три инструкције са десне стране се не могу извршавати паралелно, јер друга инструкција користи резултат прве, а трећа инструкција резултат друге.

ILP је одређен фреквенцијом стварних зависности по подацима и процедуралних зависности у кôду. Ови фактори, опет, зависе од архитектуре скупа инструкција и од његове примене. ILP је одређен оним што се назива кашњење операција: време док резултат инструкције није расположив за коришћење као операнд у следећој инструкцији. Ово кашњење одређује који износ кашњења ће зависности по подацима или процедуралне зависности изазвати.

Машински паралелизам је мера способности процесора да искористи паралелизам на нивоу инструкција. Машински паралелизам је одређен бројем инструкција које се могу прибавити и извршити у исто време (бројем паралелних проточних система) и брзином и софистицираношћу механизама које процесор користи за проналажење независних инструкција.

И ILP и машински паралелизам су важини фактори за побољшање перформанси. Неком програму може да недостаје довољно паралелизма на нивоу инструкција да би у потпуности искористио машински паралелизам. Коришћење инструкција фиксне дужине, као код RISC-ова, увећава ниво ILP-а. Са друге стране, ограничени машински паралелизам ограничава перформансе, без обзира на природу програма.

4.3.2 Политика издавања инструкција

Машински паралелизам није само питање поседовања више примерака сваког од проточних степена. Процесор мора да буде у стању да идентификује ILP и да укомпонује припрему,

декодирање и извршење инструкција у паралели. Термин *издавање инструкција* односи се на процес започињања извршења инструкције у функционалној јединици процесора, а појам *политика издавања инструкција* се односи на протокол који се користи за издавање инструкција.

У суштини, процесор се труди да гледа унапред у односу на текућу тачку извршења да би лоцирао инструкције које се могу довести у проточни систем и извршити. Три типа редоследа су овде значајна:

- Редослед у коме се инструкције прибављају.
- Редослед у коме се извршавају.
- Редослед у коме инструкције ажурирају садржаје регистара и меморијских локација.

Што је процесор софистициранији мање су стриктне границе између ових редоследа. Да би се оптимизирала употреба различитих проточних елемената процесор мора да мења један или више ових редоследа у односу на редослед који би био у стриктном секвенцијалном извршењу. Једино ограничење је да резултат мора да буде коректан. Према томе, процесор мора да се прилагођава различитим зависностима и конфликтима.

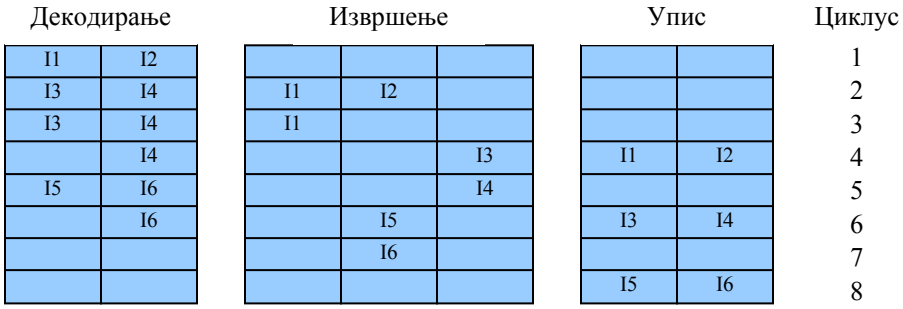
Уопштено говорећи, можемо груписати политике издавања суперскаларних инструкција у следеће категорије.

- Издавање у редоследу са завршетком у редоследу.
- Издавање у редоследу са завршетком ван редоследа.
- Издавање ван редоследа са завршетком ван редоследа.

Најједноставнија политика издавања инструкција је да се оне издају у тачном оном редоследу који се добија секвенцијалним извршењем и уписом резултата у том истом редоследу. Чак ни скаларни проточни системи не користе ову једноставну политику. Међутим, корисно је размотрити ову политику као пример за поређење са сложенијим политикама.

На слици 4.4 приказан је пример овакве политике. Претпоставимо да имамо суперскаларни проточни систем који је у стању да прибавља и декодира две инструкције истовремено и који има три различите функционалне јединице (нпр. две за целобројну аритметику и једну за аритметику у покретном зарезу) и два примерка проточних степени за упис резултата. Овај пример такође претпоставља следећа ограничења за кôдни сегмент са 6 инструкција:

- I1 захтева два циклуса.
- I3 и I4 долазе у конфликт због исте функционалне јединице.
- I5 зависи од вредности коју производи I4.
- I5 и I6 долазе у конфликт због исте функционалне јединице.



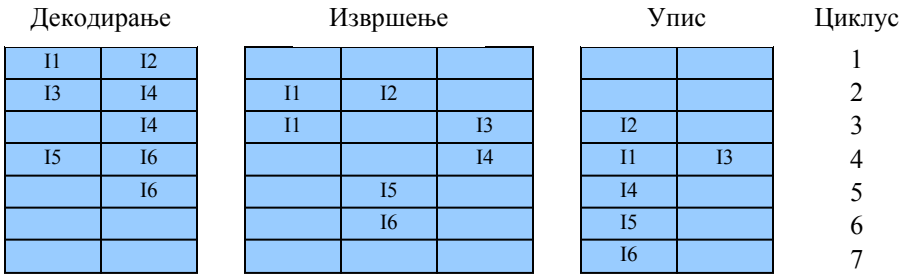
Сл. 4.4. Издавање у редоследу и завршетак у редоследу.

Инструкције се прибављају по две истовремени и пребацују јединици за декодирање. Због прибављања инструкција у пару, следеће две инструкције морају да чекају док се пар из степена за декодирање не помери у наредни степен. Да би се гарантовао завршетак у редоследу, тамо где постоји конфликт у вези функционалних јединица или када функционалној јединици треба више од једног циклуса за генерисање резултата, издавање инструкција се привремено обуставља.

Видимо да протекло време од декодирања прве две инструкције до уписа последњег резултата износи 8 циклуса.

Издавање ван редоследа се користи код скаларних RISC процесора да би се побољшале перформансе инструкција које захтевају више од једног циклуса. На слици 4.5 илустрована је

употреба ове политике код суперскаларног процесора. Инструкцији I2 је дозвољено да се изврши пре инструкције I1. Ова чињеница омогућава инструкцији I3 да се заврши раније што резултира укупном уштедом од једног циклуса.



Сл. 4.5. Издавање у редоследу са завршетком ван редоследа.

Са завршетком ван редоследа произвољан број инструкција може бити у степену за извршење у произвољном тренутку све до максималног степена машинског паралелизма функционалних јединица. Издавање инструкција може бити обустављено због конфликта ресурса, зависности по подацима или процедуралних зависности.

Уз поменута ограничења настаје и нова врста зависности који смо раније назвали излазном зависношћу (такође се зове и *write-write* зависност). Следећи фрагмент кôда илуструје ову зависност:

I1: R3 ← R3 op R5
I2: R4 ← R3 + 1
I3: R3 ← R5 + 1
I4: R7 ← R3 op R4

Инструкција I2 се не може извршити пре инструкције I1, јер јој је потребан резултат у регистру R3 који производи инструкција I1. Ово је пример праве зависности по подацима. Слично томе, I4 мора да чека на I3, јер она користи резултат који производи I3. Између I1 и I3 нема зависности по подацима али, ако се I3 изврши до краја пре I1, онда ће се погрешна вредност у регистру R3 прибавити за извршење инструкције I4. Према томе, I3 мора да се заврши после I1 да би произвела коректне излазне вредности. Да би се то осигурало издавање треће инструкције мора да се одложи ако постоји могућност да се преко њеног резултата упише резултат старије инструкције којој треба више времена да се заврши.

Завршавање изван редоследа захтева сложенију логику за издавање инструкција него што је то случај код завршетка у редоследу. Осим тога, теже се обрађују прекиди и изузеци. Када се јави прекид извршење инструкције се суспендује да би се касније обновило од тачке у којој је прекинуто. Процесор мора да обезбеди да се узме у обзир да је могуће да је инструкција испред оне која је изазвала прекид већ завршена.

Код издавања у редоследу процесор једино декодира инструкције све док се не јави нека зависност или конфликт. Док се конфликт не разреши додатне инструкције се не прибављају. Као резултат овога имамо да процесор не може да има увид унапред у односу на тачку у којој се конфликт јавио а наредне инструкције могу да буду независне од оних које су већ у проточном систему и могло би да буде корисно ако би се и оне увеле у проточни систем.

Да би се омогућило издавање ван редоследа неопходно је раздвојити степене за декодирање и извршење. То се постиже уз помоћ бафера који се назива *прозор инструкција*. Код овакве организације, када процесор заврши декодирање инструкције он је смешта у прозор инструкција. Док год се бафер не попуни процесор може да настави да прибавља и декодира нове инструкције. Када функционална јединица у степену извршења постане расположива, инструкције се из прозора инструкција предаје степену за извршење. Произвољна инструкција може бити издата под условом да је њој потребна функционална јединица расположива и да нема конфликата или зависности који би одложили извршење те инструкције.

Резултат овакве организације је да процесор има могућност увида у наредне инструкције како би идентификовао независне инструкције које се могу увести у степен за извршење. Инструкције се

издају из прозора инструкција без много обзира на њихов редослед у програму све докле то не ремети коректност извршења програма.

На слици 4.6 је илустрована ова политика. У сваком циклусу се прибављају по две инструкције. Такође се током сваког циклуса две инструкције из степена за декодирање премештају у прозор инструкција (наравно, уколико он није сасвим попуњен). У овом примеру могуће је издати инструкцију I6 пре инструкције I5 (сетимо се да I5 зависи од I4 али да то није случај са I6). Као што се са слике 4.7 види, уштедели смо један циклус у односу на претходно описану политику на слици 4.5.

Сл. 4.6. Издавање ван редоследа са завршетком ван редоследа.

Приметимо да прозор инструкција, иако је приказан на слици 4.6, није један од степени проточног система. Када је инструкција у прозору то значи да процесор има довољно информација о њој да би могао да одлучи о моменту њеног издавања.

Ова политика такође пати од ограничења која су раније описана. Инструкција се не може издати ако изазива зависност или конфликт. Разлика је у томе што је сада већи број инструкција расположив за издавање што смањује вероватноћу да ће доћи до застоја у проточном систему. Осим овога, овде се јавља нова врста зависности коју смо раније назвали антизависнот (зове се и *read-write* зависност). Следећа кôдна секвенца илуструје ову зависност.

I1: R3 ← R3 op R5
I2: R4 ← R3 + 1
I3: R3 ← R5 + 1
I4: R7 ← R3 op R4

Инструкција I3 не може да комплетира своје извршење пре него I2 почне да се извршава и прибави своје операнде. То је зато што I3 ажурира вредност у регистру R3 а та вредност је изворни операнд за инструкцију I2. Појам *антизависност* се користи јер је ово ограничење слично правој зависности по подацима, али у обрнутом смеру. Уместо да прва инструкција прозводи вредност коју користи друга, овде друга инструкција уништава вредност коју користи прва инструкција.

4.3.3 Преименовање регистара

Код политика издавања инструкција ван редоследа и/или завршетка ван редоследа видели смо да постоји могућност појаве излазних зависности и антизависности. Ове зависности се разликују од правих зависности по подацима и конфликата ресурса које одсликавају ток података кроз програм и секвенцу извршења. Са друге стране, излазне зависности и антизависности настају јер вредности у регистрима више не представљају секвенцу вредности коју диктира програм.

Када се инструкције издају и завршавају у редоследу могуће је специфицирати садржај сваког регистра у свакој тачки извршења програма. Када се користе технике ван редоследа, вредности у регистрима не могу да буду у потпуности познате у сваком тренутку само на основу посматрања низа инструкција у програму. У суштину, вредности су у конфликту у вези употребе регистара а процесор мора да реши те конфликте повременим застојима у проточном систему.

И антизависности и излазне зависности су примери меморијских конфликата. Више инструкција надмеће се у употреби истих регистара. Када се користе технике за оптимизацију коришћења регистара овај проблем је још израженији, јер компилаторске технике покушавају да максимизирају употребу регистара а тиме максимизирају и број оваквих конфликата.

Метод који се бави овим типом конфликата заснован је на традиционалној техници за разрешавање конфликата ресурса: удвостручавању ресурса. У овом контексту се ова техника назива *преименовање регистара*. У суштини, хардвер процесора динамички додељује регистре и придружују вредностима које су потребне у инструкцијама у разним временским тренуцима. Када се креира нова регистарска вредност (тј. када се изврши инструкција која има регистар као одредишни операнд), нови регистар се додељује тој вредности. Узастопне инструкције које приступају тој вредности као изворном операнду у том регистру морају да прођу кроз процес преименовања: регистарске референце у тим инструкцијама морају се изменити како би реферисале регистар који садржи потребну вредност. Према томе, иста оригинална регистарска референца у више различитих инструкција може се обраћати различитим стварним регистрима, уколико су потребне различите вредности.

Размотримо како се преименовање регистара користи код секвенце коју смо већ имали као пример:

I1: $R3_b \leftarrow R3_a$ op $R5_a$
I2: $R4_b \leftarrow R3_b + 1$
I3: $R3_c \leftarrow R5_a + 1$
I4: $R7_b \leftarrow R3_c$ op $R4_b$

Регистарске референце без индекса означавају логичке регистарске референце које се налазе у инструкцији. Регистарске референце са индексима означавају хардверске регистре додељене да садрже нову вредност. Када се начини нова додела за неки логички регистар, наредне референце у инструкцијама на тај логички регистар као изворни операнд се подешавају да би се односиле на најскорије додељени хардверски регистар.

У овом примеру, креирање регистра $R3_c$ у инструкцији I3 избегава антизависност са другом инструкцијом и излазну зависност са првом исинтрукцијом, а не утиче на коректну вредност којој приступа I4. Резултат је тај да I3 може бити издата одмах. Без преименовања, I3 не би могла да буде издата док I1 не заврши своје извршење и док се I2 не изда.

Литература

[1] W. Stallings, *Computer Organization and Architecture*, 6/e, Prentice Hall, 2003.

5 Векторски процесори

Већ смо видели да проточност и паралелизам на нивоу инструкција могу значајно да поправе перформансе. Међутим, постоје ограничења у побољшавању перформанси путем проточности. Ова ограничења јављају се због два фактора:

- **Период тактног сигнала.** Период тактног сигнала може се смањити тако што се повећа дубина проточног система, али веће дубина проточног система повећава зависности што резултује већим средњим бројем циклуса по инструкцији почев од неке дубине.
- **Брзина добављања и декодирања инструкција.** Ова препрека, често називана *Флиново уско грло*, отежава прибављање и издавање више инструкција по тактном периоду. Ово је разлог што је тешко направити процесор са високом фреквенцом тактног сигнала и веома високом брзином издавања инструкција.

Подједнако је тешко планирање проточног система n пута веће дубине као и планирање процесора који издаје n инструкција по тактном периоду.

Брзи проточни процесори су корисни за велике техничке апликације. Проточни процесори велике брзине корите кеш како би се смањило кашњење код инструкција које се обраћају меморији. На жалост, велики научни програми са дугачким временом извршавања често имају веома велике активне скупове података са мало локалности што изазива слабе перформансе хијерархијског меморијског система. Овај проблем може да се превазиђе ако се ове структуре података не кеширају уколико је могуће одредити шаблон приступа меморији и приступе меморији учинити проточним на ефикасан начин.

Новије кеш архитектуре и помоћ компилатора путем блокирања и прибављања унапред смањују ове проблеме меморијске хијерархије, али су они и даље изражени за неке апликације.

Векторски процесори пружају могућност операција на нивоу виших програмских језика које раде над векторима. Типична векторска инструкција еквивалентна је читавој петљи где се у једној итерацији обрађује један елемент вектора, ажурирају индекси и врши гранање на почетак тела петље.

Векторске инструкције имају неколико важних особина које решавају већину поменутих проблема:

- Израчунавање сваког резултата независно је од израчунавања претходног, што је могуће код проточног система велике дубине, али се овде не јављају хазарди по подацима. У основи, одсуство хазарда по подацима одређено је компилатором или од стране програмера који одлучује када се векторска инструкција може користити.
- Једна векторска инструкција специфицира велики износ посла који је еквивалентан извршењу целе петље. Према томе, захтев за ширином опсега инструкција је смањен а Флиново уско грло значајно ублажено.
- Векторске инструкције које приступају меморији имају познат шаблон приступа. Ако су елементи вектора сви суседни, тада добављање вектора из скупа веома преклопљених меморијских банки може да ради веома добро. Велико кашњење због иницирања приступа меморији у односу на приступ кешу је амортизовано, јер је један приступ иницран за цео вектор уместо за једну реч. Према томе, цена кашњења због приступа главној меморији се плаћа само једном за цео вектор уместо по једном за сваку реч у вектору.
- Како је цела петља замењена векторком инструкцијом чије је понашање унапред одређено, управљачки хазарди, који се иначе јављају због гранања у петљи, су непостојећи.

Из наведених разлога се векторске операције могу брже извршавати од секвенце скаларних операција над истим бројем података. Тако су пројектанти мотивисани да уводе векторске јединице уколико се такве инструкције често користе.

Код векторских процесора операције над појединим елементима вектора су проточне. Проточни систем не обухвата само аритметичке операције већ и приступ меморији и одређивање ефективне адресе. Осим овога, многи *high-end* процесори дозвољавају и да се више векторских операција извршава истовремено и тако уводи паралелизам између операција над различитим елементима.

5.1 Основна векторска архитектура

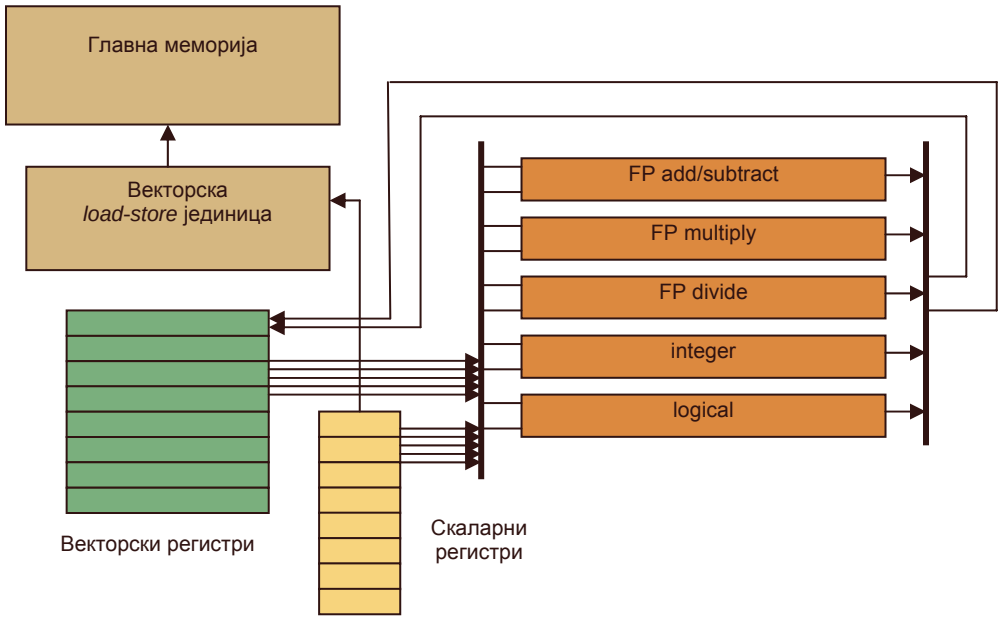
Векторски процесор се обично састоји од обичне проточне скаларне јединице и векторске јединице. Све функционалне јединице у оквиру векторске јединице имају кашњење од неколико тактних периода. Ово омогућава краћи период тактног сигнала и компатибилно је са векторским операцијама које се дуго извршавају и које се могу обрађивати проточним системом велике дубине без генерисања хазарда. Већина векторских процесора подржава рад са векторима целобројних вредности, логичких вредности или вредностима у покретном зарезу.

Постоје два основна типа векторских архитектура:

- Векторско-регистарски процесори.
- Меморија-меморија векторски процесори

У векторско-регистарским процесорима све операције, осим *load* и *store*, обављају се над векторским регистрима. Ове машине су векторски пандан већ познатих *load-store* архитектура. Сви велики векторски рачунари на тржишту од касних 80-тих су овог типа. Између осталих, то су процесори фирме Cray Research: CRAY-1, CRAY-2, X-MP, Y-MP, C-90; јапански суперрачунари NEC SX/2 и SX/3, Fujitsu VP200 i VP400, Hitachi S820 као и супер-мини рачунари Convex C-1 и C-2. У векторском процесору типа меморија-меморија све векторске операције су типа меморија-у-меморију. Први векторски рачунари били су овог типа (CDC фамилија). Ми ћемо се фокусирати надаље на векторско-регистарске архитектуре.

На слици 5.1 приказана је основна векторско-регистарска архитектура.



Сл. 5.1. Основна векторско-регистарска архитектура.

Основне компоненте ове архитектуре су:

- *Векторски регистри*. Сваки векторски регистар је фиксне дужине намењен да садржи један вектор. Сваки векторски регистар мора да има најмање два порта за читање и један порт за упис. Ова чињеница омогућава висок степен преклапања векторских

операција над различитим векторским регистрима. Улазни и излазни портови су повезани на улазе, односно излазе, функционалних јединица.

- *Векторске функционалне јединице.* Свака јединица је потпуно проточна и може да започне нову операцију после сваког тактног периода. Управљачка јединица мора да детектује хазарде, како структурне (конфликте над функционалним јединицама) тако и хазарде по подацима (конфликте у приступу регистрима). Претпостављаћемо да су функционалне јединице дељиве, али ћемо, због једноставности, игнорисати могуће конфликте.
- *Векторска load-store јединица.* То је векторска меморијска јединица која чита и уписује вектор у меморију. Ова јединица је у потпуности проточна тако да се пренос речи између меморије и регистара може, после почетног кашњења, вршити брзином од једне речи по тактном периоду. Поред векторских, ова меморијска јединица може да обавља и скаларне уписе или читање меморије.
- *Скуп скаларних регистара.* Њихови подаци могу бити улаз у векторске функционалне јединице а служе и за израчунавање адреса које се преносе векторској *load-store* једниници.

5.2 Векторско време извршења

Време извршења секвенце векторских операција првенствено зависи од три фактора:

- Дужина вектора.
- Структурни хазарди међу операцијама.
- Зависности по подацима.

За дату дужину вектора и *брзину иницијације* (брзина којом векторска јединица користи нове операнде и производи нове резултате) можемо одредити време потребно за једну векторску инструкцију. Брзина иницијације је обично једна инструкција по тактном периоду за индивидуалну операцију. Неки суперрачунари имају векторске јединице које производе два или више резултата по тактном периоду, док неки имају јединице које се не могу у потпуности учинити проточним. Због једноставности ћемо претпоставити да је брзина иницијације 1 по тактном периоду, тако да је време потребно за једну векторску инструкцију приближно једнако дужини вектора.

Да би поједноставили дискусију о векторском извршењу и одговарајућем тајмингу увешћемо појам *конвоја*. Под конвојом ћемо подразумевати скуп векторских инструкција које се могу потенцијално извршавати заједно у једном тактном периоду. Инструкције у конвоју не смеју садржати било који структурни хазард или хазард по подацима. Да би поједноставили анализу претпоставићемо да конвој инструкција треба да заврши своје извршење пре него што било која друга инструкција (скаларна или векторска) може да отпочне са извршавањем.

Појму конвоја придружићем појам *сагласја* који се може користити за процену перформанси векторске секвенце која се састоји од конвоја. Сагласје је приближна мера времена извршења за векторску секвенцу и независна је од дужине вектора. Тако ће векторска секвенца од m конвоја која се извршава у m сагласја, а за дужину вектора n, захтевати приближно $m \times n$ тактних периода.

Ако знамо број конвоја у векторској секвенци можемо да знамо време извршења изражено сагласјима.

5.3 Векторске *load-store* јединице и векторски меморијски системи

Понашање векторске *load-store* јединице је компликованије од понашања аритметичких функционалних јединица. *Време стартовања* за операцију *load* је време потребно да се прва реч смести у регистар. Ако се остатак вектора може добавити без застоја, брзина иницијације једнака је брзини којом се једна реч може прочитати или уписати. Насупрот једноставнијим функционалним јединицама, брзина иницијације није увек један по тактном периоду.

Типично време стартовања *load-store* једнице је и до 50 тактова (CRAY-1 и CRAY X-MP имају ово време између 9 и 17 тактних периода).

Да би се одржала брзина иницијације од једне речи по тактном периоду (прочитане или уписане), меморијски систем мора да буде у стању да произведе или прихвати тај износ података. Ово се обично постиже креирањем вишеструких меморијских банака.

Већина векторских процесора користи меморијске банке уместо обичног преклапања из два разлога:

1. Многи векторски рачунари подржавају вишеструка читања или уписе по истом тактном интервалу. Да би се подржао вишеструки симултани приступ меморијски систем мора да има вишеструке банке и да буде у стању да управља независним адресирањем банака.
2. Многи векторски процесори подржавају могућност читања и уписа података који нису секвенцијални. У таквим случајевима независно адресирање банака је пожељније од преклапања.

Број банака у меморијском систему и дубина проточности функционалних јединица су у суштини комплементарни појмови, јер одређују брзине иницијације операнада које користе ове јединице.

5.4 Дужина вектора

Векторско-регистарски процесори имају природну дужину вектора одређену бројем елемената у сваком од векторских регистара. Ова, унапред фиксирана, дужина вектора ретко ће одговарати стварној дужини вектора у програму. У реалним програмима, дужина појединих векторских операција често није позната у време компилације. Шта више, различити делови кôда могу да захтевају различите дужине вектора. Примера ради, размотримо следећу петљу:

```
      do 10 i = 1,n
10    Y(i) = a*X(i) + Y(i)
```

Величина свих векторских операција зависи од *n* које не мора да буде познато све до времена извршења програма. Вредност променљиве *n* може да буде и параметар процедуре која садржи горњу петљу па се тако може мењати током извршења програма.

Решење овог проблема је у креирању регистра дужине вектора (VLR – *vector length register*). Овај регистар би садржао дужину сваког вектора, а та дужина не би могла да буде већа од максималне дужине вектора (MVL) дефинисане од стране процесора.

Шта урадити ако *n* није познато у време компилације па се деси да буде веће од MVL? Да би се решио проблем који настаје када је стварна дужина вектора већа од максималне дозвољене користи се техника извлачења трака (*strip mining*). Ова техника претпоставља генерисање таквог кôда да се свака векторска операција обавља за онај део вектора који није већи од MVL.

Ево како ви изгледала горња петља преуређена коришћењем технике извлачења трака:

```
      low = 1
      VL = (n mod MVL)
      do 1 j = 0, (n / MVL)
            do 10 i = low, low+VL-1
                  Y(i) = a*X(i) + Y(i)
10          continue
            low = low + VL
            VL = MVL
 1    continue
```

Осим губитака при стартовању, морамо да урачунамо и губитке извршавања петље код које су извучене траке. Губици настали услед извлачења трака јесу они губици који се састоје у поновном иницирању векторске секвенце и постављању VLR-а.

5.5 Корак вектора

Наредни проблем односи се на то да позиција у меморији суседних елемената вектора не мора бити секвенцијална. Размотримо познати кôд за множење матрица:

```
      do 10 i = 1,100
            do 10 j = 1,100
                  A(i,j) = 0.0
                  do 10 k = 1,100
10                      A(i,j) = A(i,j) + B(i,k)*C(k,j)
```

У изразу са лабелом 10 може се векторисати множење сваке врсте матрице *B* са сваком колоном матрице *C* и вршити извлачење трака унутрашње петље којој је *k* бројач петље. Да би о урадили морамо да размотримо начин смештања елемената матрице у меморију. Као што је познато, вишедимензиона поља се приликом смештања у меморију линеаризују и смештање врши или по колонама или по врстама. Ово ће изазвати да елементи једне врсте, односно колоне, а што зависи од изабраног начина смештања, неће бити у суседним меморијским локацијама. Примера ради, FORTRAN врши смештање по колонама па ће у нашем случају множења матрица елементи матрице *B* којима се приступа у унутрашњој петљи бити раздвојени 800 бајтова, ако претпоставимо да се за смештање једног елемента користи 8B.

Растојање које одваја елементе који треба да буду у сакупљени у један векторски регистар назива се корак вектора (*vector stride*). За наш пример, корак за матрицу *C* је један (тј. 8B), а за матрицу *B* 100 (тј. 800B).

Када се вектор напуни у векторски регистар онда се са њим може радити као да су у питању логички суседни елементи. Према томе, векторско-регистарски процесори могу да обрађују нејединичне кораке вектора користећи само векторске *load* и *store* операције које имају могућност подешавања корака.

Треба имати у виду да до меморијских конфликата не може да дође уколико су корак вектора и број меморијских банака међусобно прости бројеви; у супротном конфликти су могући.

5.6 Технике за побољшање векторских перформанси

Размотрићемо три технике за побољшање перформанси векторских процесора. Прва се односи на убрзање операција над зависним векторима и назива се *ланчање*. Друге две технике се односе на проширивање класа петљи које се могу векторисати. Ту се решавају проблеми условног извршења и ретко посегнутих матрица.

5.6.1 Ланчање

Размотримо једноставну векторску секвенцу

```
MULTV       V1, V2, V3
ADDV        V4, V1, V5
```

Према досадашњим разматрањима ове две инструкције морају да буду у различитим конвојима, јер се ради о зависним инструкцијама. Са друге стране, ако се векторски регистар, у овом случају V1, третира не као једна вредност него као група индивидуалних регистара, тада можемо прослеђивања применимо на поједине елементе вектора. Оваква техника која омогућава да се ADDV започне раније назива се *ланчање*. Ланчање дозвољава да се векторска операција започне што је пре могуће, тј. чим поједини елементи вектора буду доступни. Резултат се из прве функционалне јединице у ланцу прослеђује наредној. У пракси се ланчање често имплементира тако што се омогући процесору да врши истовремено упис и читање регистра, али над различитим елементима вектора.

Прве имплементације ланчања радиле су по систему прослеђивања уз временска ограничења за изворне и одредишне инструкције у ланцу. Новије имплементације користе *флексибилно ланчање* које дозвољава векторској инструкцији да се уланчава са било којом другом активном векторском инструкцијом, под условом да то не изазива структурне хазарде. Овде је неопходно постојање више улазних и излазних портова за векторске регистре.

5.6.2 Условно извршење исказа

Два разлога умањују могућност векторизације петљи. То су условни искази унутар петљи и ретко посегнуте матрице. Програм који садржи **if** исказ у оквиру петље не може се извршавати у векторском режиму рада због појаве управљачке зависности у оквиру те петље.

Размотримо следећу петљу:

```
      do 100 i = 1, 64
            if (A(i) .ne. 0) then
                  A(i) = A(i) – B(i)
            endif
100   continue
```

Ова петља се не може векторисати због условног извршења тела петље. За решење проблема користи се *управљање маском вектора*. Ради се о вектору логичких вредности дужине MVL који управља извршењем векторских инструкција баш као што условне инструкције користе логичке изразе. Када је активирана маска вектора све векторске инструкције раде само са оним елементима вектора који одговарају елементима маске вектора са вредношћу 1. Елементи одредишног вектора који одговарају нулама у маски вектора остају неизмењени у току векторске операције. Ако се маска вектора поставља на основу неког услова, само елементи који задовољавају тај услов биће обухваћени операцијом. Брисање маске вектора поставља све њене елементе на 1 што ће изазвати да се наредна векторска операција односи на све елементе вектора.

Следећи кôд се може употребити зе претходну петљу под претпоставком да су адресе вектора A и B у регистрима Ra и Rb, респективно:

```
LV      V1, Ra        ; Ra adresa vektora A
LV      V2, Rb        ; Rb adresa vektora B
LD      F0, #0        ; puni FP nulu u F0
SNESV   F0, V1        ; postavlja VM(i) na 1 ako je V(i) <> F0
SUBV    V1, V1, V2    ; oduzimanje prema masci vektora
CVM                   ; postavlja masku na sve 1
SV      Ra, V1        ; smesta rezultat u A
```

Многи новији процесори подржавају управљање маском вектора. Неки од процесора дозвољавају употребу маске прекида само са подскупом векторских инструкција, док је код осталих могућа њена употреба код свих векторских инструкција.

Постоје и недостаци ове технике. Условне инструкције захтевају одређено време и када услов није испуњен. Ипак, елиминација гранања и управљачких зависности може ове инструкције да учини бржим чак иако не раде користан посао. Слично томе, векторске инструкције које се извршавају са маском вектора захтевају извесно време чак и за елементе којима је маска 0. Чак и са значајним бројем нула у масци, управљање маском вектора може да буде брже него скаларно извршење.

Код неких векторских процесора маска вектора служи само да онемогући памћење резулатата у назначени регистар док се стварна операција заиста извршава.

5.6.3 Ретко посегнуте матрице

Постоје технике које омогућавају да се програми који раде са ретко посегнутим матрицама извршавају у векторском начину рада. Код ретко посегнутих матрица елементи вектора су обично запамћени у неком компактном облику а приступа им се индиректно. Претпостављајући поједностављену ретко посегнуту структуру, посматрајмо следећи кôд:

```
      do 100 i = 1,n
100         A(K(i)) = A(K(i)) + C(M(i))
```

Овај кôд имплеметира сумирање ретко посегнутих вектора *A* и *C* користећи индексне векторе *K* и *M* за означавање ненултих елемената вектора *A* и *C* (вектори *A* и *C* морају да имају исти број ненултих елемената, у овом случају *n*). Још једна уобичајена репрезентација ретко посегнутих матрица користи бит вектор за означавање ненултих елемената и густ вектор који садржи ненулте елементе. Често оба типа репрезентација могу да постоје у истом програму. Ретко посегнуте

матрице се јављају у многим програмима и постоји више начина за њихову имплементацију, зависно од употребљених структура података.

Основни механизам подршке раду са ретко посегнутим матрицама су *распрши-скупи операције* које користи индексне векторе. Циљ таквих операција је прелазак са згуснуте репрезентације матрице на обичну репрезентацију. Операција *скупи* користи индексни вектор и добавља векторе чији се елементи на адресама које се добијају сабирањем базне адресе са офсетом датим у индексном вектору. Резултат је *густ* вектор у векторском регистру. Пошто се ти елементи обраде у густом облику, ретко посегнути вектор се може сместити у развијеном облику операцијом типа *распрши*.

У следећем кôду ћемо претпоставити да Ra, Rc, Rk и Rm садрже почетне адресе вектора из претходне петље:

```
LV      Vk, Rk          ; napuni K
LVI     Va, (Ra+Vk)     ; napuni A(K(I))
LV      Vm, Rm          ; napuni M
LVI     Vc, (Rc+Vm)     ; napuni C(M(I))
ADDV    Va, Va, Vc      ; saberi vektore
SVI     (Ra+Vk), Va     ; smesti A(K(I))
```

5.7 Програмски језици за векторске рачунаре

Ради се о мањим проширењима постојећих језика (нарочито FORTRAN-а) уз ослањање на векторске компилаторе који производе одговарајући кôд за циљну машину. Неки од језика су:

- CFT – Cray FORTRAN
- IVTRAN – Illiac IV FORTRAN
- CFD (налик FORTRAN-у) и Glypnir (налик ALGOL-у) за Illiac IV.
- Actus.

Пожељно је да ови језици буду високог нивоа, али се ту крије опасност да транспарентност архитектуре води ка слабијем искоришћавању њених предности. Разумно је очекивати да, ако програмски језик дозвољава програмеру да експлицитно представи паралелизам уграђен у алгоритам на начин који се лако може искористити на циљној машини, онда то олакшава компилацију.

Литература

[1] K. Hwang, F. A. Briggs, *Computer Architecture and Parallel Processing*, New York, McGraw-Hill, 1984.

[2] D. A. Patterson, J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, 2/e, Morgan Kaufmann publishers, inc. San Francisko, California, 1996.

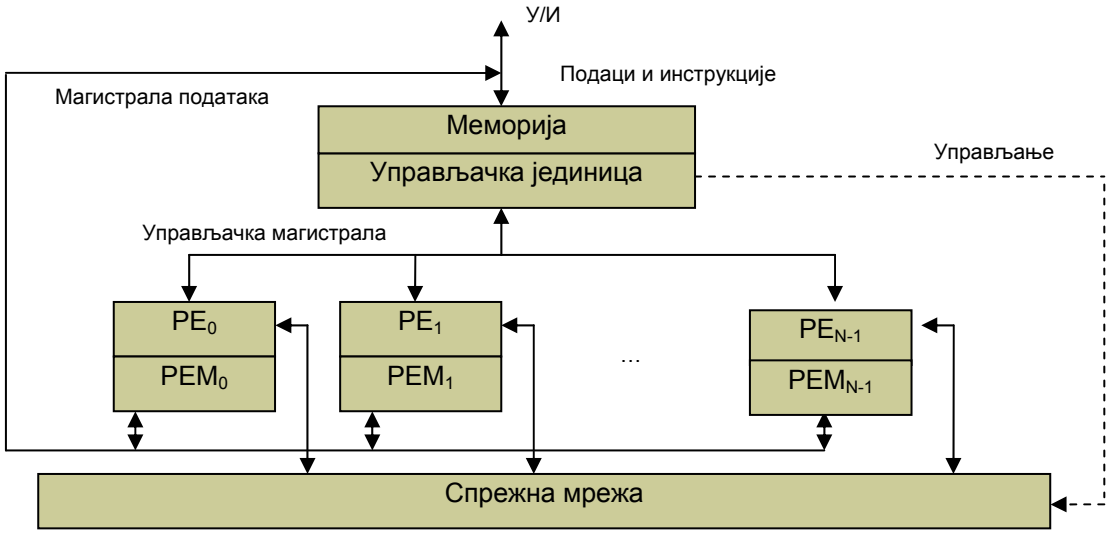
6 Процесорска поља

Синхроно поље паралелних процесора назива се *процесорско поље* и састоји се од већег броја процесних елемената (PE) којима управља једна управљачка јединица (CU). Процесорска поља раде са једним током инструкција и са више токова података па им је друго име SIMD рачунари. SIMD машине су пројектоване за обављање векторских операција над матрицама или пољима података.

SIMD рачунари се јављају у две основне архитектуре: процесорска поља која користе RAM и асоцијативни процесори који користе асоцијативну меморију. Ми ћемо се углавном позабавити првим од ова два типа машина.

6.1 Организација SIMD рачунара

Основна организација процесорског поља приказана је на слици 6.1.



Сл. 6.1. Организација процесорског поља.

Оваква организација се, између осталих, може срести код познатог рачунара Illiac-IV. Постоји *N* синхронизованих процесних елемената којима управља једна управљачка јединица (CU). Сваки PE је у ствари ALU којој су придружени локални регистри и локална меморија (PEM) у којој се смештају локални подаци. Управљачка јединица такође има своју меморију за смештање програма и она управља извршењем системских и корисничких програма. Кориснички програми се са спољних меморијских медијума пуне у меморију управљачке јединице. Функција управљачке јединице је да декодира све инструкције и да одреди где ће се декодиране инструкције извршавати. Инструкције скаларног и управљачког типа извршавају се директно у CU. Векторске инструкције се шаљу сваком од процесних елемената ради дистрибуираног извршења чиме се постиже просторни паралелизам путем умножених функционалних јединица.

Сви процесни елементи обављају исту функцију синхроно под командом управљачке јединице. Векторски операнди дистрибуирају се локалним меморијама процесних елемената пре паралелног извршења у процесним елементима. Дистрибуирани подаци се могу напунити из спољашњих извора преко системске магистрале или из управљачке јединице путем управљачке магистрале.

За управљање стањем сваког од процесних елемената током извршења векторских инструкција користе се шеме маскирања. Сваки од процесних елемената може бити активан или неактиван у току једног циклуса инструкције. У ту сврху се користи векор маскирања. Ради се о томе да не морају сви процесни елементи да учествују у извршењу неке векторске инструкције, већ само они којима је то дозвољено (наравно, некада су то сви постојећи процесни елементи).

Пренос података између процесних елемената одвија се преко спрежне мреже која обавља све неопходне функције манипулације и усмеравања података. Овом спрежном мрежом такође управља управљачка јединица.

Процесорско поље је обично повезано са хостом преко управљачке јединице. Хост рачунар је машина опште намене која управља читавим системом (систем се састоји од хоста и процесорског поља). Функције хоста обухватају управљање ресурсима и перифералима, као и управљање извршењем У/И операција. Управљачка јединица процесорског поља директно управља извршењем програма док хост машина обавља извршне и У/И функције са спољашњим окружењем. У том смислу процесорско поље је *back-end* рачунар.

Произвољни SIMD рачунар C (процесорско поље) формално се може окарактерисати следећим скупом параметара:

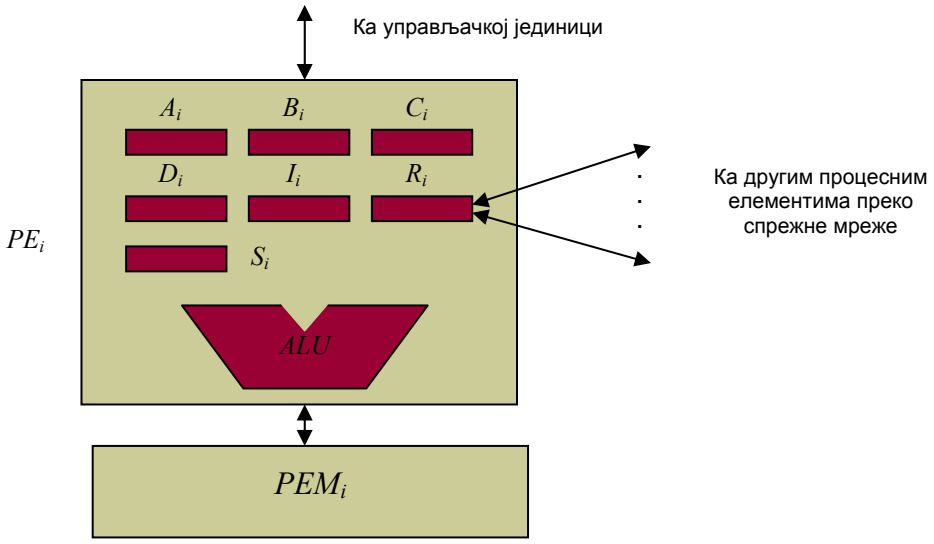
$$C = \langle N, F, I, M \rangle$$

где је N број процесних елемента у систему, F скуп функција за усмеравање података које пружа спрежна мрежа, I скуп машинских инструкција за скаларне и векторске операције, операција усмеравања и манипулација спрежном мрежом и M скуп шема маскирања.

Овај модел пружа општу основу за евалуацију различитих SIMD машина.

6.2 Структура процесних елемената

Сваки процесни елемент PE_i (слика 6.2) представља процесор са локалном меморијом PEM_i, скупом радних регистара A_i, B_i, C_i, статусним регистром S_i, локалним индексним регистром I_i, адресним регистром D_i и регистром за усмеравање R_i.



Сл. 6.2. Структура процесног елемента.

Регистар R_i сваког PE_i повезан је преко спрежне мреже са R_j регистрима других процесних елемената. Када се јави пренос података између процесних елемената садржај R_i регистара је тај који се преноси. Приметимо да има N процесних елемената означених са PE_i, $i = 0, 1, \ldots, N\text{-}1$, где је индекс i адреса од PE_i. Да би олакшали илустрацију претпоставићемо да је $N = 2^m$, тј. да је $m = \log_2 N$ битова потребно за кодирање адресе процесног елемента. Адресни регистар D_i садржи m-тобитну адресу просесног елемента.

Нека процесорска поља могу да имају по два регистра за усмеравање, један за улаз други за излаз. Ради једноставности ми ћемо претпоставити да постоји само по један такав регистар по процесном елементу а који може да служи у обе сврхе.

6.3 Технике маскирања процесних елемената

Сваки од процесних елемената може да буде у активном или неактивном стању током сваког циклуса инструкције. Ако је PE_i активан он извршава инструкцију коју је емитовала управљачка јединица. Ако је PE_i неактиван онда он не извршава ту инструкцију. Шеме за маскирање се користе за специфицирање статусног маркера S_i у процесном елементу PE_i. Обично се узима да је $S_i = 1$ за активан а $S_i = 0$ за неактиван процесни елемент. У управљачкој јединици постоји глобални индексни регистар I и регистар маске M. Регистар M има N битова при чему i-ти означавамо са M_i. Колекција маркера S_i формира регистар стања S за све процесне елементе. Шаблони битова у регистрима M и S могу се по налогу управљачке јединице разменити када се поставља маска.

6.4 Комуникација међу процесним елементима

Када се пројектује спрежна мрежа морају се донети одлуке које се тичу режима рада, стратегије управљања, метода пребацивања и топологије мреже.

Два типа комуникације, тј. р*ежима рада* могу да се идентификују: синхрони и асинхрони. *Синхрони* начин рада се јавља код свих SIMD машина и служи са синхроно успостављање комуникационих путева било за функцију манипулације подацима било за емисију инструкција. *Асинхрона* комуникација је неопходна код вишепроцесорских машина где се захтеви за повезивањем издају динамички. Могуће је и да систем има могућности и синхроног и асинхроног рада што онда представља *комбиновани* режим рада.

Типична спрежна мрежа састоји се од извесног броја прекидачких елемената и веза између њих. Функције везе реализују се погодним успостављањем управљања над прекидачким елементима. Управљање може да врши централизовани контролер или поједини прекидачки елемент. Стратегија управљања код које је управљање делегирано прекидачким елементима назива се *дистрибуирано управљање*. У случају централизованог контролера ради се о *централизованом управљању* и среће се код већине SIMD машина.

Два су основна метода пребацивања: пребацивање колима и пребацивање пакетима. Код *пребацивања колима* се успостављају физички путеви између извора и одредишта. Код *пребацивања пакетима* се подаци, организовани у пакете, усмеравају кроз спрежну мрежу без успостављања физичких путева. Код већине SIMD машина срећемо пребацивање колима док се мреже са пребацивањем пакетима више користе код MIMD машина.

Спрежна мрежа се може описати графом у коме чворови представљају прекидачке тачке а потези комуникационе линије. Топлогије мреже треба да су регуларне и могу се сврстати у две категорије: статичке и динамичке. Код *статичких топологија* везе између два процесора су пасивне посвећене магистрале које се не могу реконфигурисати ради директног повезивања са другим процесорима. Код *динамичке топологије* везе могу да се реконфигуришу, тј. садрже активне прекидачке елементе.

6.5 Спрежне мреже код SIMD рачунара

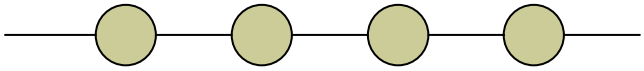
Топлошка структура SIMD процесора углавном је одређена спрежном мрежом између процесних елемената. Формално се таква мрежа може описати скупом спрежних функција. Ако адресе процесних елемената чине скуп

$$S = \{0, 1, 2, \ldots, N-1\},$$

свака спрежна функција је бијекција скупа S на S.

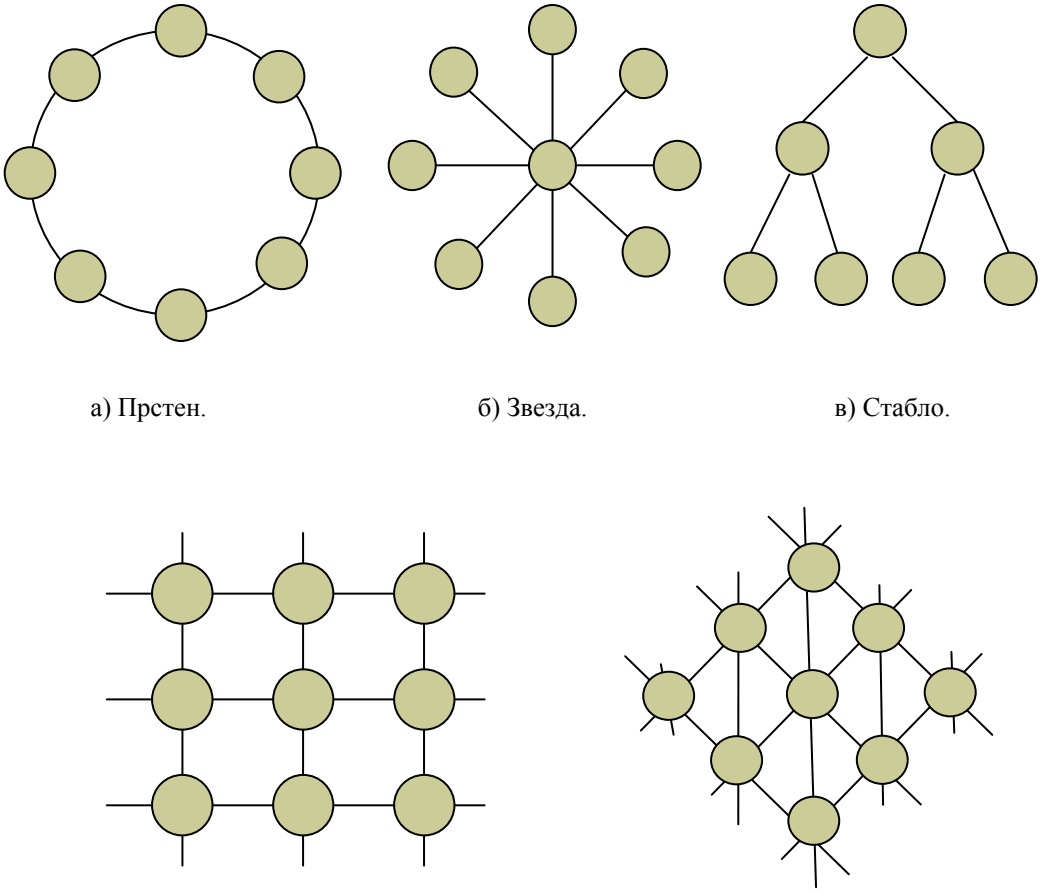
Када се извршава спрежна функција преко спрежне мреже, онда PE_i копира садржај свог R_i регистра у $R_{f(i)}$ регистар процесног елемента $PE_{f(i)}$. Овај пренос се обавља симултано за све активне елементе. Неактиван елемент може да прими податке али не може да их шаље. Ако два процесна елемента нису директно повезана, онда пренос података мора да иде преко других процесних елемената.

Спрежне мреже могу се поделити у две категорије: *статичке* и *динамичке*. Топологије статичких мрежа могу се класификовати према њиховом димензионом уређењу. Пример једнодимензионе топологије је *линеарни низ* или *листа* (слика 6.3).



Сл. 6.3. Линеарна листа.

Примери дводимензионалних топологија су *прстен*, *звезда*, *стабло*, *мрежа* и *систоличко поље*. Ови типови топлогија илустровани су на слици 6.4.

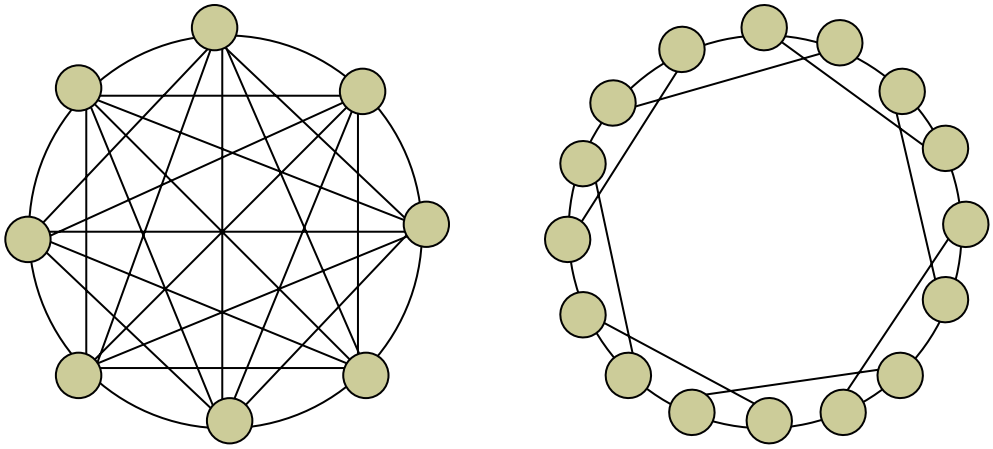


а) Прстен. б) Звезда. в) Стабло.

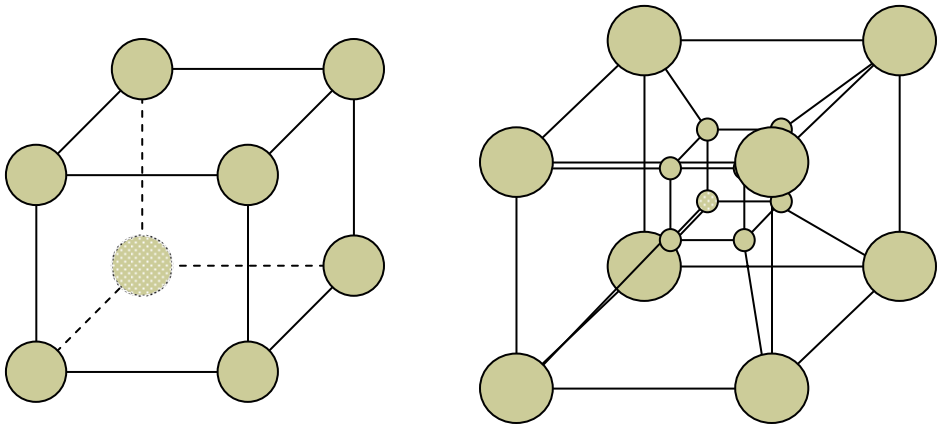
г) Мрежа. д) Систоличко поље.

Сл. 6.4. Примери дводимензионалних топологија спрежних мрежа.

Примери тродимензионалних топологија су: *потпуно повезани прстен*, *тетивни прстен*, *коцка* и *хиперкоцка*. На слици 6.5. илустроване су ове топологије.



а) Потпуно повезан прстен. б) Тетивни прстен.

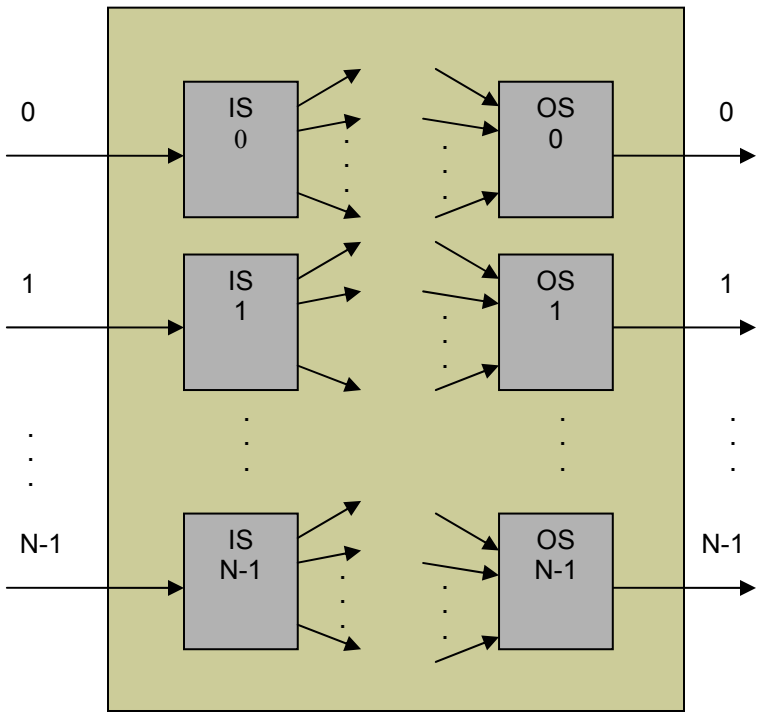


в) Коцка. г) Хиперкоцка.

Сл. 6.5. Примери тродимензионалних топологија спрежних мрежа.

Динамичке спрежне мреже могу бити *једностепене* и *вишестепене*. Једностепена мрежа је прекидачка мрежа са N улазних селектора (IS) и N излазних селектора (OS) као што је то приказано на слици 6.6. Сваки IS је у суштини представља 1-у-D демултиплексер док је сваки OS један M-у-1 мултиплексер, где је $1 \le D \le N$ и $1 \le M \le N$.

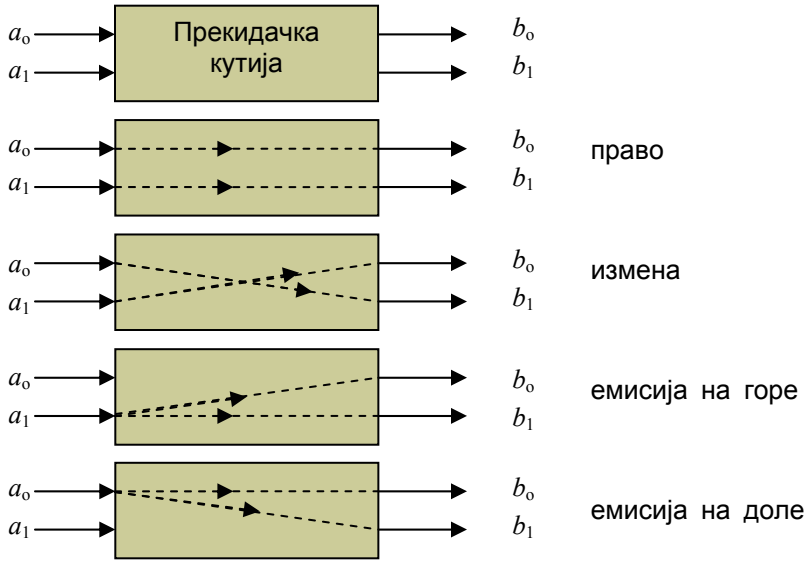
Прекидачка мрежа типа *crossbar* представља једностепену мрежу код које је $D = M = N$. Једностепене мреже се такође називају и *рециклулирајућим*, јер подаци могу да круже преко неколико степени пре него стигну до коначног одредишта. Колико ће таквих кружења бити, зависи од повезаности мреже. У општем случају, што је већа хардверска повезаност мањи је број рециркулација. Мрежа типа *crossbar* је екстремни случај код кога је само једна циркулација потребна да се успостави било која веза. Међутим потпуно повезана *crossbar* мрежа има цену која је $O(N^2)$ што може бити неприхватљиво за велике вредности N. Већина рециркулирајућих мрежа има цену реда $O(N \log N)$ или нижу што је много повољније за велике вредности N.



Сл. 6.6. Концептуални изглед једностепене спрежне мреже.

Вишестепене мреже садрже више степени међусобно повезаних прекидача. Ове мреже карактеришу се на основу три особине: *прекидачке кутије*, *топологије мреже* и *управљачке структуре*.

Вишестепене мреже садрже више прекидачких кутија које представљају прекидачки уређај који може да буде у једном од четири стања: *право*, *измена*, *емисија на горе* и *емисија на доле*. Прекидачке кутије и начин повезивања у сваком од стања дати су на слици 6.7.



Сл. 6.7. Прекидачке кутије два-у-два и њихова четири стања.

Вишестепена мрежа је у стању да повеже произвољни улазни терминал са произвољним излазним терминалом. Вишестепене мреже могу бити *једностране* и *двостране*.

Једностране мреже (потпуни прекидачи) имају У/И портове са исте стране. Двостране вишестепене мреже имају улазну страну и излазну страну и могу се сврстати у три класе: блокирајуће, реконфигурабилне и неблокирајуће.

Код *блокирајућих мрежа* симултано повезивање више од једног пара терминала може да резултује конфликтом над комуникационим линијама у мрежи. Мрежа се назива *реконфигурабилном* ако може да изведе све могуће везе између улаза и излаза путем промене конфигурације постојећих веза тако да се конекција за нови улазно-излазни пар увек може успопставити. Мрежа која може да успостави све могуће везе без појаве блокирања назива се *неблокирајућа*.

У општем случају, вишестепена мрежа састоји се од n степени где је $N = 2^n$ број улазних и излазних линија. Међутим, сваки степен може да користи $N/2$ прекидачких кутија. Шаблон међусобних веза од степена до степена одређује топологију мреже. Сваки степен је повезан се следећим са најмање N путева. Кашњење кроз мрежу пропроционално је броју степени n у мрежи. Цена вишестепене мреже пропорцинална је $N\log_2 N$. Управљачка структура мреже одређује начин на који се постављају стања прекидачких кутија. Постоје два типа управљачких структура које се користе у конструкцији мрежа: управљање индивидуалним стањима и управљање индивидуалним кутијама.

Управљање индивидуалним стањима користи исти управљачки сигнал да постави све прекидачке кутије у исто стање. Другим речима, све кутије у оквиру истог степена морају бити у истом стању. Према томе, потрено је n постављања управљачких сигнала за постављање стања свих n степени.

Код *управљања индивидуалним кутијама* посебан управљачки сигнал се користи да се постави стање сваке прекидачке кутије. Ово пружа већу флексибилност у постављању веза али захтева $n^2/2$ управљачка сигнала који ће значајно да повећају сложеност управљачких кола.

Компромис између ова два приступа присутан је код парцијалног управљања степенима где се у степену i, $0 \leq i \leq n$-1, користи i+1 управљачки сигнал.

Литература

[1] K. Hwang, F. A. Briggs, *Computer Architecture and Parallel Processing*, New York, McGraw-Hill, 1984.

7 Мултипроцесори

Већина првих вишепроцесорских архитектура биле су SIMD типа. Такође, током 80-тих година обновила се пажња посвећена овим машинама. Међутим, у новије време MIMD архитектуре постају водеће као архитектуре вишепроцесорских машина опште намене. Два су основна разлога одговорна за овај успон MIMD машина:

1. MIMD нуди флексибилност. Уз одговарајућу хардверску и софтверску подршку, MIMD машине могу да раде као једнокорисничке машине фокусирајући се на високе перформансе једне апликације, као мултипрограмиране машине које обављају више задатака симултано или као нека комбинација ових функција.
2. MIMD машине се могу градити на предностима везаним за однос цена/перформансе микропроцесора који су расположиви на тржишту. У ствари, скоро сви мултипроцесори који се данас праве користе исте микропроцесоре који се користе за радне станице и мале једнопроцесорске сервере.

Мултипроцесори морају да буду скалабилни. Хардвер и софтвер морају да буду пројектовани тако да могу да се продају са различитим бројем процесора који код неких машина варира и до 50 пута. Како је софтвер скалабилан, неки мултипроцесори могу да подржавају рад чак иако је неки хардверски део отказао, тј. ако је неки од n процесора отказао, систем непрекидно пружа услуге са n-1 процесором. Најзад, мултипроцесори могу да имају најбоље перформансе, тј. бржи су и од најбрже једнопроцесорске машине.

Мултипроцесори данас заиста имају своје упориште на тржишту. Како су микропроцесори са најбољим односом цена и перформанси, мултипроцесор састављен од више једнопроцесора на једном чипу је ефикаснији од високо перформансних једнопроцесорских машина. Готово сви фајл-сервери могу се наручити са више процесора тако да је индустрија база података стандардизована на вишепроцесорским машинама.

7.1 Класификација мултипроцесора

Код комерцијалних мултипроцесора високе перформансе обично значе високу пропусну моћ за независне задатке. Ова дефиниција је у супротности са извршавањем једног задатка на више процесора. Под *програмом са паралелном обрадом* подразумеваћемо један програм који се извршава на више процесора истовремено.

Кључна питања код пројектовања мултипроцесора су:

- Како паралелни процеси деле податке?
- Како се паралелни процеси координирају?
- Колико процесора употребити?

Као што ћемо видети, одговор на прво питање може бити двојак. Једна од могућности је да процесори деле јединствени адресни простор, делећи податке. Процесори комуницирају преко дељивих променљивих у меморији при чему је сваки процесор у стању да приступи произвољној меморијској локацији. Стога је потребна координација рада ових процеса коју називамо *синхронизација*.

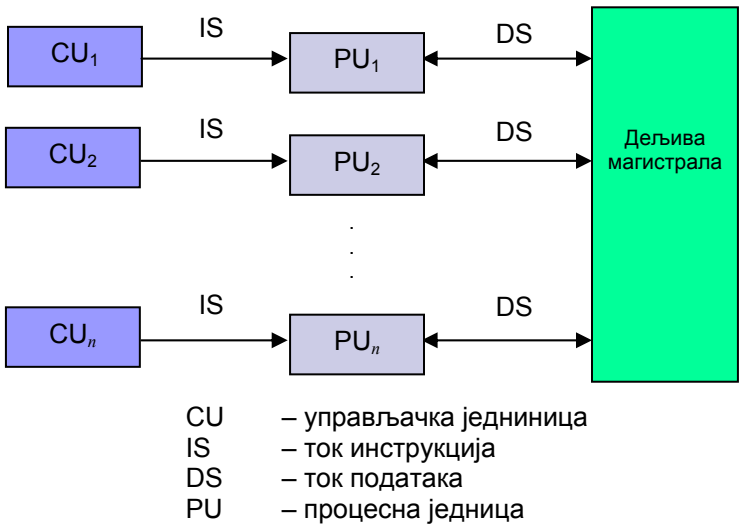
Код MIMD организације процесори су опште намене: сваки од њих је у стању да обради све инструкције неопходне за обављање жељене трансформације података. MIMD машине се могу поделити према броју употребљених процесора. Број процесора, опет, диктира организацију меморије и стратегију њиховог међусобног повезивања.

Постојеће MIMD машине сврставају се у две класе, зависно од броја употребљених процесора, који опет диктира организацију меморије и стратегију њиховог међусобног повезивања. Практично машине делимо према њиховој организацији меморије, јер се оно што данас представља мали или велики број процесора може лако променити током времена.

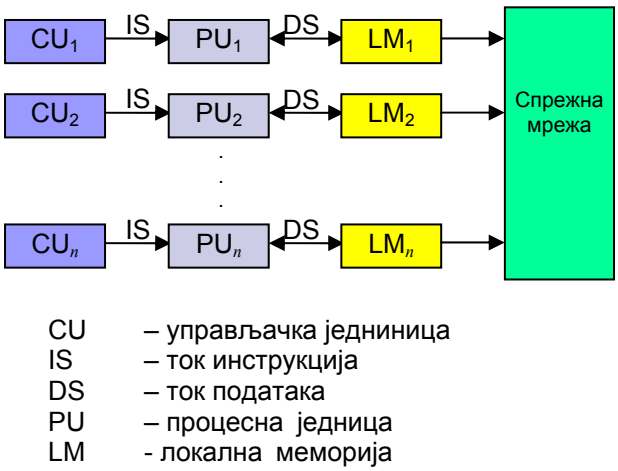
Ако процесори деле заједничку меморију, онда сваки процес приступа подацима у дељивој меморији а комуникација се одвија преко те меморије. Овакав тип организације процесора зове се и *јако спрегнути*. Овакве машине обично имају највише неколико десетина процесора (средином 90-тих година XX века).

Најуобичајенији облик ове организације су симетрични мултипроцесори (SMP – *Symmetric Multiprocessors*). SMP деле јединствену меморију преко заједничке магистрале или другог механизма за повезивање. Карактеристична особина је да је време приступа произвољној области у меморији приближно исто за сваки од процесора. Алтернативно име које се употребљава за овај тип мултипроцесора је UMA (*Uniform Memory Access*).

У новије време јавља се организација NUMA (*Non-Uniform Memory Access*), где је време приступа меморији различито за неке од процесора.



Сл. 7.1. MIMD са дељивом меморијом.



Сл. 7.2. MIMD са дистрибуираном меморијом.

Алтернативни начин је да процесори имају *приватне меморије* а да комуницирају *слањем порука*. Екстремни случај ове организације јесу колекције независних једнопроцесора или SMP-ова

који се могу повезати да чини кластер (*cluster* – гомила). Комуникација међу рачунарима иде преко фиксних путева или преко спрежне мреже која може бити и LAN. Оваква организација спада у *слабо спрегнуте*.

Уз два главна начина комуникације, мултипроцесори могу да буду конструисани у две основне организације:

- Процесори повезани на заједничку магистралу.
- Процесори повезани на мрежу.

Број употребљених процесора повезан је са избором једне од ове две организације. На слици 7.1 приказана је MIMD организација са дељивом меморијом а на слици 7.2 MIMD са дистрибуираном меморијом.

7.2 Симетрични мултипроцесори

До скора је сваки PC рачунар, као и већина радних станица, садржао само један микропроцесор. Са порастом захтева за перформансама и падом цена микропроцесора, на тржишту су се јавили системи са SMP организацијом. SMP се односи на архитектуру хардвера рачунара и на понашање оперативног система које одсликава ту архитектуру.

SMP се може дефинисати као самосталан рачунарски систем са следећим карактеристикама:

1. У систему постоје 2 или више процесора приближних могућности.
2. Сви процесори деле исту меморију и У/И систем а међусобно су повезани преко магистрале или друге интерне шеме за повезивање која омогућује приближно исто време приступа меморији за сваки од процесора.
3. Сви процесори деле приступ У/И уређајима било преко истих канала или различитих канала који омогућавају приступ истом уређају.
4. Сви процесори могу да обављају исте функције (отуда назив симетрични).
5. Системом управља интегрисани оперативни систем који пружа интеракцију између процесора и њихових програма на нивоу посла, задатка, датотеке и других нивоа елемената података.

Карактеристике 1-4 јасне су по себи. Карактеристика 5 илуструје један од контраста са слабо спрегнутим мултипроцесорима као што су кластери, где је јединица интеракције порука или читава датотека. Код SMP поједини елементи података могу да сачињавају ниво интеракције па може да постоји већи степен сарадње међу процесорима.

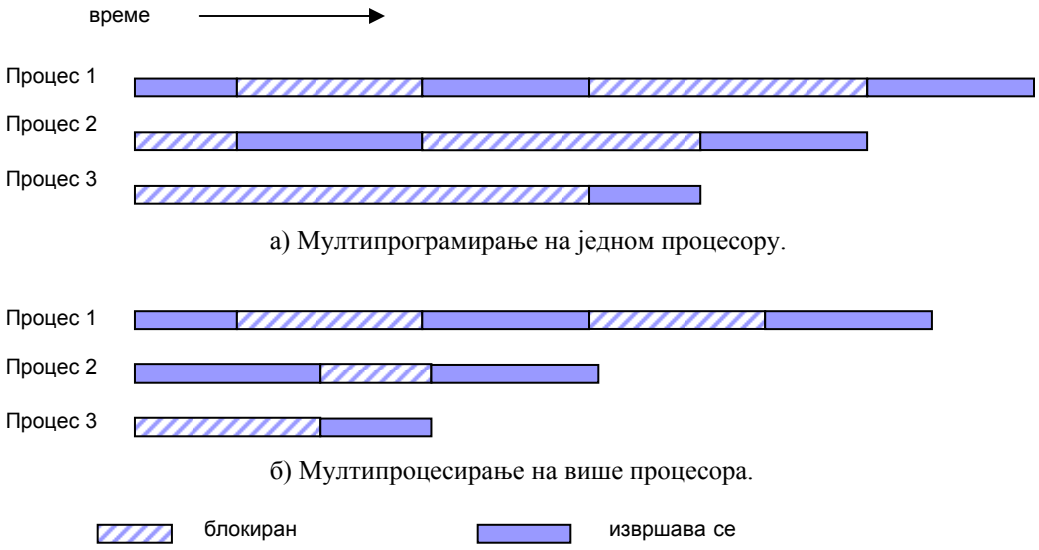
Оперативни систем једног SMP планира процесе или нити за све процесоре. SMP организација има известан број потенцијалних предности над једнопроцесорским организацијама међу којима су:

- **Перформансе**. Ако се посао који треба да се обави на рачунару организује тако да се делови посла могу извршавати паралелно, систем са више процесора даће боље перформансе од оних које постиже један процесор истог типа. (види слику 7.3)
- **Расположивост.** Код SMP-а сви процесори обављају исту функцију па у случају отказа једног од њих не долази до застоја целе машине. Систем тада наставља да ради са умањеним пеформансама.
- **Инкрементални раст.** Корисник може да побољшава перформансе система додавањем нових процесора.
- **Скалирање.** На тржишту може бити понуђен читав спектар производа са различитим ценама и перформансама у зависности од броја уграђених процесора.

Важно је запазити да су ово могуће а не гарантоване добити. Оперативни систем мора да обезбеди алате и функције за искоришћавање паралелизма SMP система.

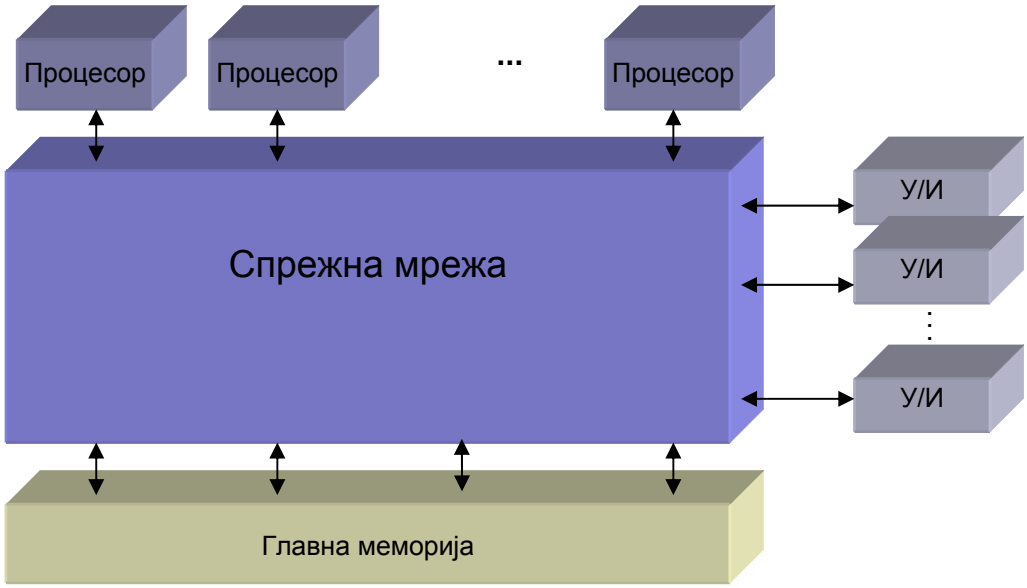
Атрактивна особина SMP-а је да је постојање више процесора транспарентно за корисника. Оперативни систем води рачуна о планирању нити или процеса на појединим процесорима као и о синхронизацији међу процесорима.

На слици 7.4 приказана је уопштена организација мултипроцесорског система. Као што се са слике 7.4 види, у систему постоје 2 или више процесора. Сваки процесор је самосталан и садржи управљачку јединицу, ALU, регистре и обично један или више нивоа кеша. Сваки процесор има приступ дељивој главној меморији и У/И уређајима преко механизма за повезивање. Процесори комуницирају преко главне меморије, а могуће је да измењују сигнале директно. Меморија је обично организована тако да је могуће извршити више симултаних приступа различитим блоковима.



Сл. 7.3. Ефекти мултипрограмирања и мултипроцесирања.

Код неких конфигурација сваки процесор може да има сопствену приватну меморију и У/И канале уз постојеће дељиве ресурсе.



Сл. 7.4. Општа шема чврсто спрегнутог мултипроцесора.

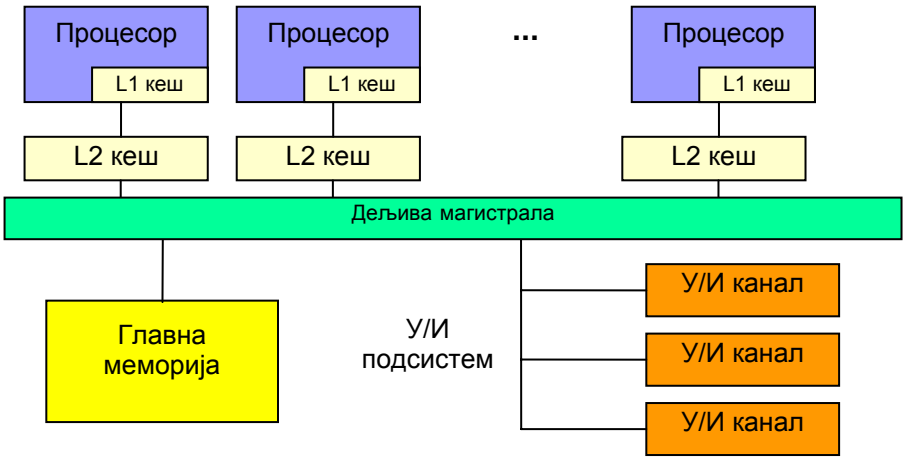
Организациони приступ за SMP могу се сврстати у три категорије:

- Дељива или заједничка магистрала.
- Вишепортна меморија.
- Централна управљачка јединица.

Сваки од ових приступа ћемо објаснити са нешто више детаља.

7.2.1 Магистрала дељива у времену

Дељива магистрала је најједноставнији механизам за конструкцију вишепроцесорског система. На слици 7.5 приказана је организација симетричног мултипроцесора са дељивом магистралом. Структура и интерфејси су у основи исти као и код једнопроцесорских система који користе јединствену магистралу. Магистрала се састоји од линија за податке, адресних и управљачких линија.



Сл. 7.5. Организација симетричног мултипроцесора са дељивом магистралом.

За омогућавање DMA преноса из У/И процесора обезбеђена су следећа средства:

- **Адресирање:** Мора бити могуће да се разликују модули на магистрали како би се одредио извор и одредиште података.
- **Арбитража:** У/И модул може привремено да ради као господар магистрале. Обезбеђен је механизам за арбитражу у случају више захтева за управљање над магистралом који користи неку врсту алгоритма са приоритетом.
- **Дељење у времену:** Док неки модул управља магистралом остали морају да чекају.

Ова средства карактеристична су за једнопроцесорске системе и директно су употребљива код SMP. У случају SMP постоји више процесора као и више У/И процесора при чему сви покушавају да преко магистрале приступе неком од меморијских модула.

Организација са заједничком магистралом има неколико предности у поређењу са другим приступима:

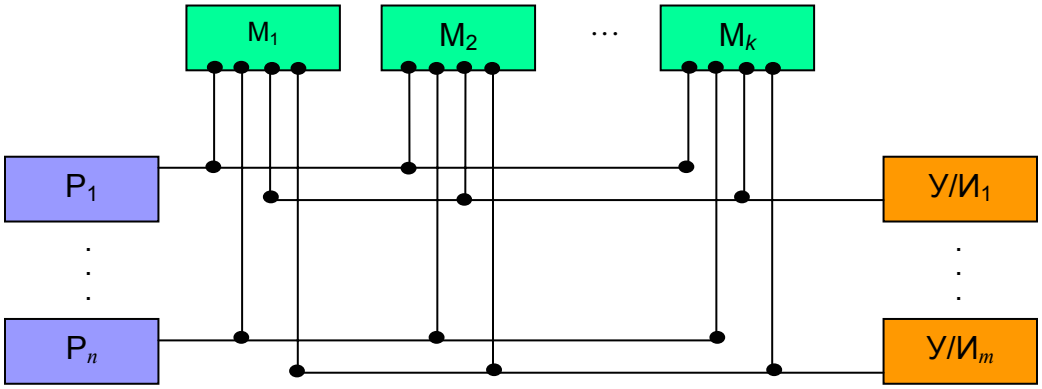
- **Једноставност:** Ово је најједноставнији приступ организацији мултипроцесора. Физички интерфејси, адресирање, арбитража и логика за дељење у времену сваког процесора исти су као и код једнопроцесорских система.
- **Флексибилност:** У општем случају је лако проширивати систем прикључивањем нових процесора на магистралу.
- **Поузданост:** Магистрала је у суштини пасивни медијум и отказ било ког од прикључених уређаја неће изазвати отказ читавог система.

Постоје и недостаци овакве организације а тичу се перформанси. Све меморијске референце пролазе преко заједничке магистрале. Према томе, време циклуса магистрале ограничава брзину система. Да би се перформансе побољшале пожељно је да се сваки процесор опреми кеш меморијом. Ово драстично смањује саобраћај на магистрали.

Међутим, употреба кеш меморија уноси нове проблеме. Сваки локални кеш садржи слику дела меморије. Ако се нека реч измени у неком кешу, потребно је упозорити друге процесоре да је дошло до ажурирања и да њихова копија података није више валидна. Овај проблем познат је као *кеш-кохеренција*. Ради се о претежно хардверском проблему и о њему ће бити више речи касније.

7.2.2 Вишепортне меморије

Приступ са вишепортним меморијама омогућује директан, независан приступ модулима главне меморије од стране сваког процесора и У/И модула. Ова организација је приказана на слици 7.6. Меморији се придружује логика која служи за разрешавање конфликата. Метод који се често користи за разрешавање конфликата је додељивање сталног приоритета сваком меморијском порту. Физички и електрични интерфејс сваког порта идентичан је онима код једнопортних меморијских модула. Очигледно је потребно мало или нимало модификација код сваког процесора или У/И модула да се прилагоди вишепортној меморији.



Сл. 7.5. Организација симетричног мултипроцесора са мултипортном меморијом.

Овај приступ је сложенији од приступа са заједничком магистралом и захтева да се у систем угради већи износ логике. Међутим, ово треба да обезбеди боље перформансе, јер сваки процесор има посвећен пут до сваког меморијског модула. Још једна предност овог приступа је да је могуће конфигурисати делове меморије као приватне за један или више процесора и/или У/И модула. Ова особина омогућава повећану безбедност од неауторизованих приступа.

За управљање кешом треба користити политику *write-through*, јер нема начина да се остали процесори упозоре на ажурирање садржаја меморије.

7.2.3 Централна управљачка јединица

Централна управљачка јединица врши каналисање различитих токова података у и ван различитих модула: процесора, меморије или У/И. Контролер баферује захтеве и врши функције арбитраже и синхронизације. Такође може да преноси стање и управљачке поруке између процесора и врши функцију упозорења о измени садржаја кеша.

Како је сва логика за координацију концентрисана у централној управљачкој јединици, интерфејси У/И модула, меморије и процесора остају у суштини недистрибуирани. Ово омогућује флексибилност и једноставност као код приступа са магистралом. Кључни недостатак је у сложености управљачке јединице која је уз то и потенцијално уско грло.

Овај приступ је раније био уобичајен код вишепроцесорских *mainframe* система (нпр. код фамилије IBM S/370) али се данас ретко среће.

7.3 Кеш кохеренција

У савременим вишепроцесорским системима је уобичајено да се сваком процесору придружи један или два нивоа кеш меморије. Оваква организација је од суштинске важности за постизање одговарајућих перформанси. Међутим, увођење кеш меморија изазива проблем који је познат као *проблем кеш кохеренције*. Суштина проблема је у следећем: више копија истог податка може да постоји истовремено у различитим кеш меморијама па ако процесор може да ажурира сопствене копије података као резултат се може јавити неконзистентан поглед на меморију.

Увођење кеш меморија је и код једнопроцесорских машина изазвало проблем кохеренције код У/И операција, јер је поглед на меморију преко кеша могао да буде различит од погледа добијеног преко У/И подсистема. Уобичајене политике уписа које се користе су:

- **write back**: операција уписа врши се само у кешу, а ажурирање меморије врши се само код замене линије кеша.
- **write through**: све операције уписа обављају се и у меморији и у кешу чиме је осигурано да су подаци у главној меморији увек ваљани.

Јасно је да је као резултат политике *write back* могућа неконзистентност. Ако два кеша садрже исту линију а та линија се измени само у једном од њих, други кеш ће и незнајући да има погрешну (тј. застарелу) вредност.

Код вишепроцесорских машина проблем кеш кохеренције је још израженији него код једнопроцесорских, јер се поглед на меморију два различита процесора одвија преко њихових индивидуалних кеш меморија. Како више копија истог података може постојати у различитим кешевима, а процесори слободно модификују своје копије, резултат тога може бити неконзистентност. У табели 7.1 приказана је илустрација овог проблема (подразумева се политика уписа *write through*).

Таб. 7.1. Проблем кеш кохеренције када два процесора модификују исту локацију.

Време	Догађај	Садржај кеша за CPU A	Садржај кеша за CPU B	Садржај меморијске локације X
0				1
1	CPU A чита X	1		1
2	CPU B чита X	1	1	1
3	CPU A уписује 0 у X	0	1	0

Неформално се може рећи да је меморијски систем *кохерентан* ако произвољно читање података даје најскорије уписану вредност тог податка. Ова дефиниција је поједностављена и садржи два аспекта понашања меморијског система, оба критична за писање коректних програма са дељивом меморијом. Први аспект се назива *кохеренција* и дефнише *која* вредност се враћа приликом читања. Други аспект је *конзистентност* и одређује *када* се уписана вредност враћа приликом читања.

Меморијски систем је кохерентан ако:

1. Читање локације X од стране процесора P које следи после уписа процесора P у локацију X, без уписа у X од стране других процесора између уписа и читања од стране P, увек враћа вредност уписану од стране P.
2. Читање локације X које следи иза уписа од стране другог процесора у ту локацију, враћа уписану вредност ако су читање и упис довољно раздвојени и не јављају се други уписи у локацију X између та два приступа.
3. Упис у исту локацију се *серијализује*, тј. два уписа у исту локацију од стране прозвољна два процесора виде се у истом редоследу од стране свих процесора.

Прво својство чува програмски редослед и треба да буде испуњено и код једнопроцесорских машина. Друго својство дефинише појам кохерентног погледа на меморију. Ако процесор може да непрекидно чита стару вредност онда је меоријски систем некохерентан.

Потреба за серијализацијом је суптилнија али једнако важна. Претпоставимо да нисмо извршили серијализацију уписа и процесор P1 уписује улокацију X иза чега следи упис процесора P2 у локацију X. Серијализација уписа обезбеђује да сваки процесор у истој тачки виду упис извршен од стране процесора P2. Да није извршена серијализација могло би се десити да неки процесор види најпре упис процесора P2 а затим упис процесора P1 задржавауђи на тај начин претходну вредност дотичне локације. Најједноставнији начин да се избегне овакав проблем је серијализација уписа тако да се сви уписи у исту локацију виде у истом редоследу.

Мада су ове три особине довољне да обезбеде кохеренцију, питање када ће уписана вредност бити виђена је такође важно. Да би разумели зашто је конзистенција сложена приметимо да не можемо да захтевамо да читање локације X моментално види вредност за X уписану од стране неког другог процесора. Ако, на пример, упис у X на неком процесору претходи читању X на неком другом процесору током кратког времена, вероватно неће бити могуће обезбедити да читање врати уписану вредност, јер она можда није још напустила процесор.

Питање када тачно уписана вредност мора да буде виђена од страна читалаца дефинише се као *модел меморијске конзистенције*. Кохеренција и конзистенција су комплементарне. Кохеренција дефинише понашање читалаца и писаца у исту меморијску локацију, док конзистенција дефинише понашање читалаца и писаца у односу на приступ другим меморијским локацијама.

7.4 Протоколи за обезбеђивање кеш кохеренције

Проблем кохеренције за мултипроцесоре и У/И, иако слични у пореклу, имају различите карактеристике које утичу на одговарајућа решења проблема. Насупрот У/И, где више копија једног истог податка представљају редак случај који треба избегнути кад год је то могуће, код мултипроцесора се захтева да у току рада програма постоје копије истих података у неколико кеш меморија придруженим различитим процесорима. У кохерентном мултипроцесору кеш меморије омогућавају и *миграцију* и *репликацију* дељивих података. Кохерентне кеш меморије омогућавају миграцију, јер се податак може ископорати у локални кеш и одатле користити на транспарентан начин што смањује кашњење код приступа дељивим подацима који се налазе у удаљеној меморији. Кохерентни кеш такође пружа и могућност репликације дељивих података који се симултано читају, јер кеш меморије праве копије дотичних података. Репликација смањује и кашњење код приступа и код надметања ради читања дељивих података.

Подршка миграцији и репликацији је критична за перформансе приступа дељивим подацима. Из тих разлога мултипроцесори са малим бројем процесора уместо софтверских решења овог проблема преферирају хардверска решења која користе протоколе за обезбеђивање кеш кохеренције.

7.4.1 Софтверска решења

Већ смо наговестили да решења проблема кеш кохеренције могу бити софтверска и хардверска. Софтверска решења покушавају да избегну потребу за додатним хардвером тако што се ослањају на компилатор и оперативни систем, тј. решење се тражи у могућностима системског софтвера да отклони проблем. Овакав приступ је атрактиван, јер се детекција потенцијалних проблема, уместо у време извршења, обавља у време комплације а сложеност пројектовања, уместо на хардвер, је пренета на софтвер. Са друге стране, софтверске технике у општем случају користе конзервативна решења која воде ка неефикасној употреби кеша.

Механизми кохеренције који се засновају на компилатору врше анализу кôда како би утврдили који подаци могу да буду несигурни за кеширање и на одговарајући начин их обележавају. Хардвер или оперативни систем даље спречавају кеширање таквих података.

Најједноставнији приступ је да се спречи кеширање било које од дељивих променљивих. Ово је исувише конзервативно, јер се дељиве структуре података могу користити током дужег периода и то само у сврху читања. Проблем кеш кохеренције се у ствари јавља само онда када бар један од процеса врши модификацију података и бар један од процеса приступа тим подацима.

Ефикаснији приступи анализирају кôд како би одредили безбедне периоде за дељиве променљиве. Компилатор тада убацује инструкције у генерисани кôд како би осигурао кеш кохеренцију током критичних периода. Постоји више техника које се баве оваквим анализама и одговарајућим генерисањем кôда.

7.4.2 Хардверска решења

Ова решења се у општем случају називају протоколима кеш кохеренције. Ова решења динамички, у току извршења програма, препознају услове могуће неконзистенције. Како се проблем решава тек када се јави, ова решења су ефикаснија по питању коришћења кеша и општих перформанси система. Осим тога, транспарентна су за програмера и компилатор растерећујући развој софтвера.

Хардверске шеме се разликују у извесном броју детаља, као што је место чувања информације о линијама података, како су те информације организоване, када се кохеренција намеће и каквим механизмима.

У општем случају хардверске шеме деле се у две категорије:

- Директоријумске протоколе, и
- *Snoopy* протоколе.

Директоријумске шеме сакупљају и одржавају информације о томе где се копије неке линије налазе. Обично постоји централизовани контролер као део контролера главне меморије и директоријум смештен у главну меморију. Овај директоријум садржи информацију о глобалном стању различитих локалних кеш меморија. Када неки кеш контролер изда захтев, централизовани контролер проверава и издаје неопходне команде за пренос података између меморије и кешева или између самих кешева. Овај контролер је такође организован да информације о стању одржава ажурним, што значи да се свака локална акција која може да утиче на глобално стање линије мора пријавити централном контролеру.

Обично контролер одржава информацију о томе који процесор има копију које линије. Пре него процесор изврши упис у локалну копију линије он мора од контролера да захтева ексклузиван приступ тој линији. Пре него што контролер то одобри, он шаље поруку свим процесорима да њихова копија више није ваљана. Пошто прими потврду од свих процесора, издаје дозволу процесору који је извршио захтев. Ако неки други процесор покуша да прочита линију која је одобрена другом процесору, он шаље поруку о промашају контролеру. Тада контролер налаже процесору који држи дотичну линију да је упише назад у меморију. Сада та линија може да буде дељена ради читања од стране свих осталих процесора.

Директоријумске шеме имају недостатак који се огледа у уском грлу централизованог контролера и губицима због комуникације различитих кеш контролера и централног контролера.

Snoopy протоколи дистрибуирају одговорност за одржавање кеш кохеренције између свих кеш контролера у мултипроцесору. Кеш мора да препозна када је линија коју садржи дељена са другим кешевима. Када се врши ажурирање дељиве линије кеша о томе се обавештавају сви остали кешеви механизмом емитовања. Сваки контролер мора да буде у стању да "њуши" (*snoop* – њушкати) спрежну мрежу како би уочио обавештење и реаговао на одговарајући начин.

Snoopy протоколи су идеални за мултипроцесоре засноване на заједничкој магистрали, јер она пружа једноставне начине за емитовање и откривање поруке. Међутим, како је сврха локалних кеш меморија да се смањи саобраћај на магистрали мора се водити рачуна да се због овог протокола не изгуби корист која се добија употребом кеш меморија.

Постоје два основна приступа код *snoopy* протокла:

- Инвалидација при упису.
- Ажурирање при упису.

Код *инвалидације при упису* може постојати више читаоца али само један писац истовремено. У почетку више кеш меморија може да дели линију ради читања. Када нека од кеш меморија пожели да изврши упис, она најпре издаје поруку којом инвалидира дотичну линију у осталим кеш меморијама. На тај начин дотична линија постаје ексклузивна за кеш који врши упис. После тога процесор "власник" те линије може једноставно да врши своје локалне уписе све док неки други процесор не захтева ту линију.

Код *ажурирања при упису* може постојати и више писаца и више читалаца. Када процесор жели да ажурира дељиву линију он шаље свим осталим процесорима реч која се ажурира тако да и остале кеш меморије могу да је ажурирају.

Ниједан од ова два протокола није супериоран у односу на други под свим околностима. Перформансе зависе од броја локалних кеш меморија и шаблона читања и уписа у меморију. Неки системи имају адаптивне протоколе који могу да користе оба поменута протокола.

7.4.3 MESI *протокол*

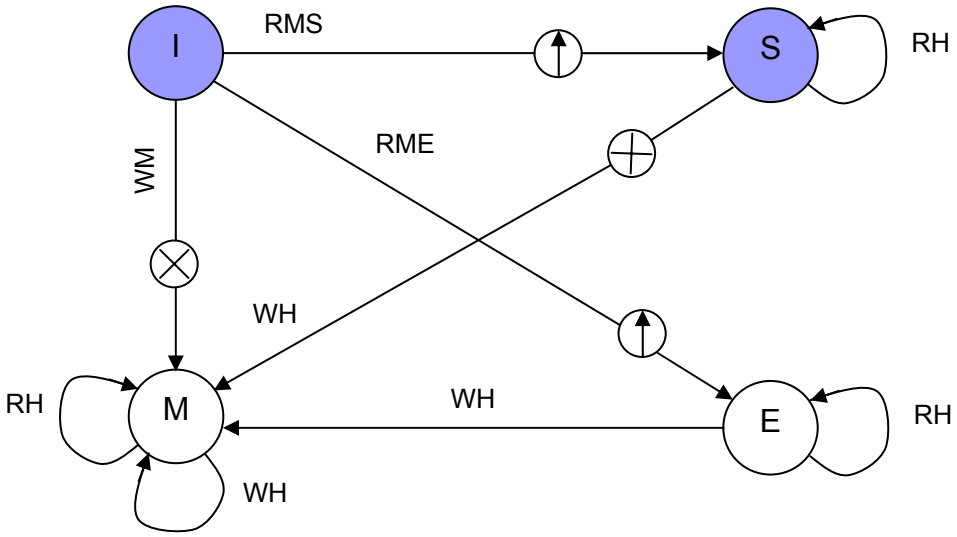
Приступ са инвалидацијом при упису широко је заступљен у комерцијалним мултипроцесорима као што су Pentium 4 и PowerPC. Протокол који се користи назива се MESI *протокол*. Код овог протокола тагу кеш линије придружена су два додатна бита тако да свака линија може бити у једном од четири стања.

- ***Модификовано***. Линија кеша је модификована (различита од оне у главној меморији) и расположива је једино у том кешу.
- ***Експклузивно***. Линија у кешу је иста као и у главној меморији и не постоји у осталим кеш меморијама.
- ***Дељиво***. Линија у кешу иста је као и у главној меморији и може постојати у другим кеш меморијама.
- ***Инвалидно***. Линија у кешу не садржи валидне податке.

У табели 7.2 приказано је значење ових стања. На сликама 7.6 и 7.7 приказани су дијаграми стања код MESI протокола и то за иницирајући и "њушећи" процесор, респективно.

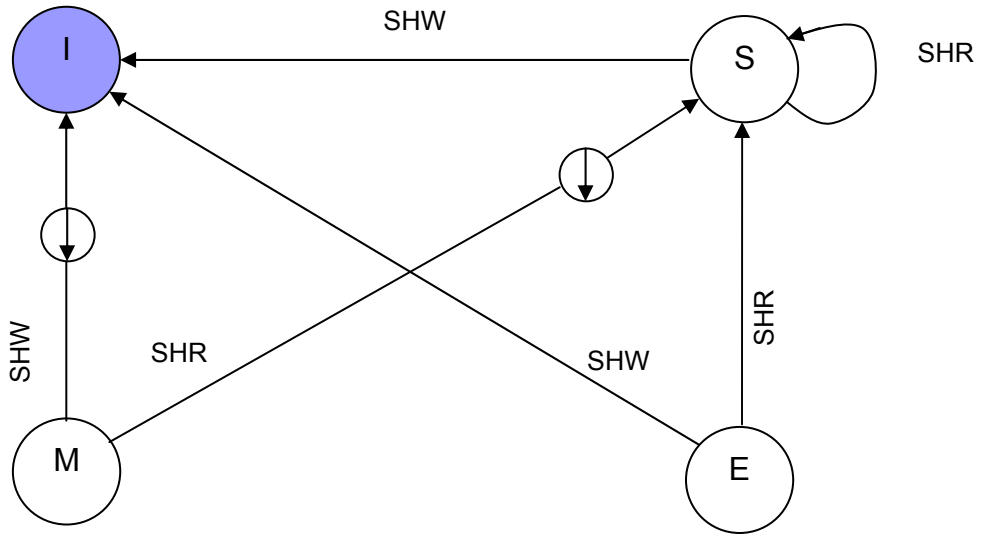
Таб. 7.2. Стања линија кеша код MESI протокола.

	M модификовано	E екслузивно	S дељиво	I инвалидно
Линија кеша је валидна?	Да	Да	Да	Не
Копија у меморији је ...	застарела	ваљана	ваљана	-
Копија постоји у другим кешевима?	Не	Не	Можда	Можда
Упис у ову линију ...	не иде на магистралу	не иде на магистралу	иде на магистралу и ажурира кешеве	иде директно на магистралу



Сл. 7.6. Дијаграм стања за линију кеша иницирајућег процесора.

Треба имати у виду да свака линија кеша има сопствене битове стања и, самим тим, сопствену реализацију дијаграма стања. На слици 7.8. приказано је значење скраћеница и других ознака које се јављају на сликама 7.6 и 7.7.



Сл. 7.7. Дијаграм стања за линију кеша који "њуши".

RH	погодак при читању
RMS	промашај при читању, дељиво.
RME	промашај при читању, ексклузивно
WH	погодак при упису
WM	промашај при упису
SHR	*snoop* погодак при читању
SHW	*snoop* погодак при упису или читање са намером модификације


Копирање модификоване линије


Трансакција инвалидирања


Читање са намером модификације


Пуњење линије кеша

Сл. 7.8. Значење скраћеница и ознака у дијаграмима стања.

Сагледајмо сада могуће прелазе из дијаграма стања приказаних на сликама 7.6 и 7.7 са нешто више детаља.

Промашај при читању

Када се јави у локалном кешу, процесор иницира читање меморије и поставља сигнал којима се упозоравају остали процесори и кеш меморије да “њуше” трансакцију. Постоји неколико могућих исхода:

- Ако неки други кеш има немодификовану копију линије у екслузивном стању он враћа сигнал којим указује да дели ту линију. Процесор који одговара мења стање своје копије од екслузивног на дељиво и иницирајући процесор чита линију из меморије мењајући јој стање од инвалидног на дељиво.

- Ако један или више кешева има немодификовану копију линије у дељивом стању, сваки од њих сигнализира да дели ту линију. Иницирајући процесор чита линију и мења јој стање у свом кешу од инвалидног на дељиво.
- Ако неки други кеш има модификовану копију линије тада он блокира читање меморије и предаје линију кешу који ју је захтевао преко магистрале. Процесор који одговара мења стање линије кеша од модификованог на дељиво.
- Ако ниједан кеш нема копију те линије (било немодификовану било модификовану) тада се не враћа никакав сигнал. Иницирајући процесор чита линију и мења јој стање у свом кешу од инвалидног на експклузивно.

Погодак при читању

Процесор једноставно чита линију из свог кеша не мењајући јој стање (стање остаје модификовано, дељиво или експклузивно).

Промашај при упису

Процесор иницира читање линије из главне меморије. У ту сврху он издаје сигнал који значи *читање са намером модификације* (RWITM). Када се линија напуни одмах се прогласи модификованом.

У односу на друге кеш меморије два могућа сценарија претходе пуњењу одговарајуће линије.

- Неки други кеш може да има модификовану копију те линије; тада он то сигнализира иницирајућем процесору. Иницирајући процесор предаје магистралу и чека. Други процесор приступа магистрали, уписује модификовану линију у меморију и мења стање линије кеша у инвалидно (зато што ће је иницирајући процесор модификовати). Потом иницирајући процесор поново издаје сигнал RWITM и чита линију из главне меморије, модификује је у кешу и пребацује стање на модификовано.
- Други сценарио је да ниједан други кеш нема модификовану копију захтеване линије. У том случају не враћа се никакав сигнал а иницирајући процесор чита линију из главне меморије и модификује је. За то време, ако један или више процесора има немодификовану копију те линије у дељивом стању, сваки од кешева инвалидира своју копију а такође и онај кеш који евентуално има немодификовану копију у експклузивном стању.

Погодак при упису

Ефекат поготка при упису линије у локални кеш зависи од њеног текућег стања:

- Дељиво. Пре ажурирања процесор мора да добије експклузивно власништво над линијом. Он сигнализира своју намеру преко магистрале. Сваки процесор који има дељиву копију линије у свом кешу мења њено стање на инвалидно. Иницирајући процесор тада врши ажурирање и мења стање линије од дељивог на модификовано.
- Експклузивно. Пошто већ има експклузивно власништво процесор једноставно изврши ажурирање и промени стање линије на модификовано.
- Модификовано. Процесор већ има експклузивно власништво над линијом у модификованом стању тако да само обавља ажурирање.

7.5 Кластери

Једна од најновијих области у пројектовању рачунарских система јесте пројектовање *кластера*. Кластери представљају алтернативу симетричним мултипроцесорима као приступ који пружа високе перформансе и високу расположивост, а који је нарочито атрактиван за серверске апликације.

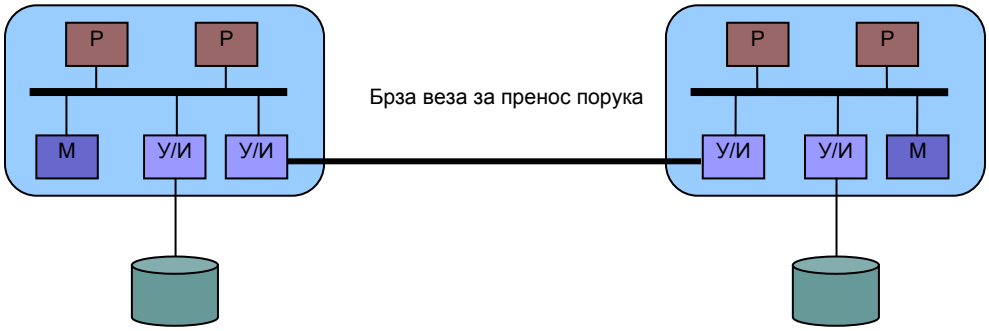
Кластер (*cluster* – енг. гомила) можемо дефинисати као групу повезаних *целих рачунара* који раде заједно као јединствени рачунарски ресурс стварајући илузију да се ради о једној машини. Појам цео рачунар значи да се ради о систему који може самостално да ради и када није део кластера. Сваки рачунар у кластеру обично се означава као *чвор*.

Кластери имају следеће предности:

- **Апсолутна скалабилност:** могуће је креирати велике кластере који далеко превазилазе могућности чак и највећих самосталних машина. Кластер може да има на десетине машина при чему је свака мултипроцесор.
- **Инкрементална скалабилност:** кластер се конфигурише на такав начин да је могуће додавати нове системе кластеру у малим количинама. Корисник може да започне са набавком скромнијег система а да га касније, према потребама, проширује. При том је таква врста проширења много једноставнија него иначе, када се мањи систем замењује већим.
- **Висока расположивост:** како је сваки чвор у кластеру самостална машина, отказ једног чвора не значи и губитак услуге. Код многих производа ове врсте се толеранцијом отказа управља аутоматски путем софтвера.
- **Супериорни однос цена/перформансе:** користећи градивне блокове широко распрострањене на тржишту могуће је саставити кластер са истом или већом моћи од неке велике машине, али по много мањој цени.

7.5.1 Класификација кластера

Кластери се могу класификовати на више начина. Најједноставнија класификација је заснована на томе да ли рачунари у кластеру деле приступ истим дисковима. На слици 7.9 је приказан кластер са два чвора који су повезани само преко брзе везе за размену порука. Ова веза може бити LAN који се дели са другим рачунарима који нису у кластеру или неко посвећено средство за повезивање. У другом случају један или више рачунара у кластеру су повезани на LAN или WAN тако да постоји веза између кластера сервера и удаљеног клијента. Сваки од рачунара је приказан као мултипроцесор што није неопходно, али повећа перформансе и расположивост.



Сл. 7.9. Сервер без дељивог диска.

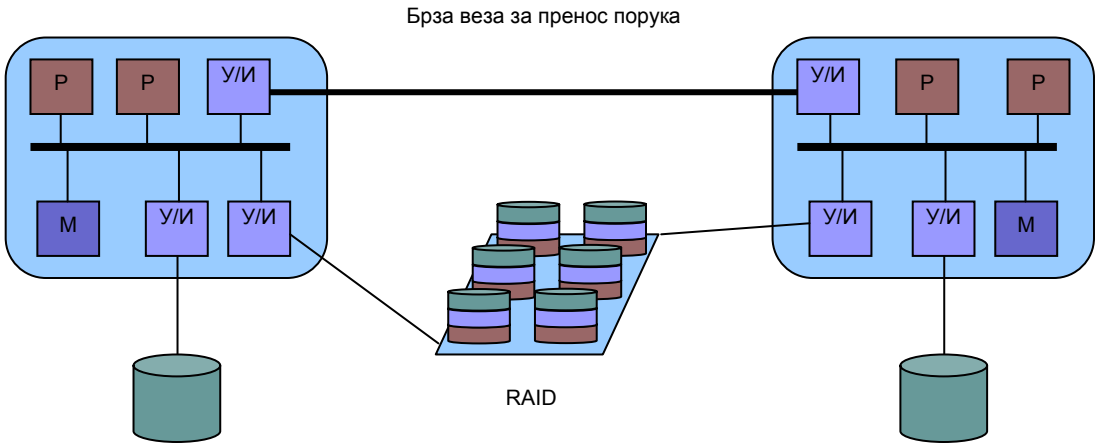
На слици 7.10 приказана је друга алтернатива, тј. кластер са дељивим диском. У овом случају још увек постоји веза за размену порука између чворова, али и подсистем дискова који је директно повезан са рачунарима у кластеру. Овде је као заједнички диск систем приказан RAID систем. Коришћење оваквог или сличног редудантног диск система је уобичајено код кластера како се расположивост постигнута већим бројем чворова не би угрозила отказом јединственог дељивог диск система.

Следећа класификација кластера врши се на основу функција:

- Пасивни *standby*.
- Активни секундарни.
    - Одвојени сервери.
    - Сервери повезани на дискове.
    - Сервери деле дискове.

Пасивни *standby* је старији метод код којег имамо један рачунар који обавља обраду а други је неактиван и служи као резерва у случају отказа примарног сервера. Ради координације ових машина примарни сервер периодично шаље поруку (*heartbeat*) секундарној машини. Ако таква порука

престане да долази, секундарна машина претпоставља да је примарни рачунар отказао и да сада треба да се активира. Овај повећава расположивост али не побољшава перформансе. Осим тога, ако је *heartbeat* једина информација коју рачунари измењују и ако не деле заједнички диск, онда секундарни рачунар нема приступ подацима које је до тада обрађивао примарни. Пасивни *standby* се у општем случају и не назива кластером, јер се појам кластера веже за више повезаних рачунара који активно учествују у обради при чему споља постоји утисак да се ради о једној машини.



Сл. 7.10. Сервер са дељивим системом дискова.

Активни секундарни је појам којим се једна таква конфигурација означава. Заправо имамо три врсте кластерских конфигурација:

- Одвојени сервери
- Сервери који не деле ништа.
- Сервери који деле меморију.

Сваки рачунар у *одвојеним серверу* има сопствени диск и нема дељивих дискова (види слику 7.9). На овај начин се постижу високе перформансе и расположивост. Нека врста управљачког софтвера је неопходна да би се долазећи захтеви клијената додељивали серверима уз постизање уравнотежености и високе искоришћености. Пожељно је да постоји могућност да један сервер преузме посао другог ако дође до његовог отказа (*failover*). Да би ово било могуће подаци морају стално да се копирају између система што повећава поузданост али смањује перформансе.

Да би се смањили губици услед комуникације већина кластера се састоји од сервера повезаних на заједничке дискове (види слику 7.10). У варијацији овиг приступа која се назива *сервери који не деле ништа*, заједнички дискови су подељени у партиције при чему сваку партицију користи један рачунар. У случају отказа неког рачунара кластер мора да се реконфигурише да би неки други рачунар стекао власништво над партицијом рачунара који је отказао.

Такође је могуће да више рачунара дели исте дискове истовремено (приступ *дељивих дискова*) тако да сваки рачунар има приступ свим партицијама на свим дисковима. Овај приступ захтева неки механизам који би омогућио узајмну искључивост над партицијама.

7.5.2 Карактеристике оперативних система за кластере

Да би се у пуној мери искористио хардвер кластера потребно је извршити нека побољшања у оперативним системима за самосталне рачунаре а која се односе на:

- Управљање отказима.
- Уравнотежење оптерећења.
- Паралелизацију израчунавања.

Како се врши *управљање отказима* зависи од метода кластеризације. У општем случају могу се употребити два приступа: високо расположиви кластери и кластери толерантни на отказе.

Високо расположиви кластери нуде велику вероватноћу да ће сви ресурси бити у функцији. Ако се отказ јави (пад система или губитак неког волумена на диску) онда су упити који су у току

изгубљени. Произвољни изгубљени упит који се обнови биће опслужен од стране другог рачунара у кластеру. Међутим, оперативни систем кластера не гарантује стање парцијално извршених трансакција, већ то треба да се обави на нивоу апликације.

Кластери толерантни на отказе обезбеђују да су сви ресурси увек расположиви. Ово се постиже коришћењем редудантних дељивих дискова и механизама за повраћај необављених трансакција и предају завршених трансакција. Функција пребацивања апликација и ресурса података од система који је отказао на алтернативни систем у кластеру назива се *failover*. Сродна функција која се састоји од обнављања апликације и ресурса података на оригинални систем када се он поправи назива се *failback*. Ова функција се може аутоматизовати али је пожељно да се она обави тек пошто је проблем заиста решен и када није вероватно да ће се поново јавити. У супротном, аутоматизовани *failback* би изазвао честе преносе ресурса са једног на други рачунар што значајно деградира перформансе и изазива проблеме у опоравку.

Кластери захтевају средство за *уравнотежење оптерећења* рачунара расположивих у кластеру. То средство мора да узме у обзир и захтев за инкременталном скалабилношћу кластера. Када се нови рачунар дода кластеру, средство за уравнотежење оптерећења треба аутоматски да укључи тај рачунар у апликације које се односе на планирање. Механизми мидлвера треба да препознају да се услуге могу јављати код различитих чланова кластера и да могу да мигрирају са једног на други чвор у кластеру.

Ефикасна употреба кластера захтева *паралелизовано извршење* једне апликације. Постоје три приступа овом проблему:

- **Паралелизовани компилатор**: у време компилације одређује делове апликације који се могу извршити паралелно. Ти делови се затим додељују ради извршења различитим рачунарима у кластеру. Перформансе зависе од природе проблема и квалитета компилатора.
- **Паралелизована апликација**: код овог приступа програмер пише апликацију свестан да ће се она извршавати на кластеру користећи метод размене порука за пренос података између кластера. Ово оптерећује програмера али је можда најбољи начин да се искористе све могућности.
- **Параметризовано израчунавање**: овај приступ се у суштини може користит ако срж апликације чини алгоритам или програм који се мора велики број пута извршити за различите податке. Добар пример су модели за симулацију који се извршавају за велики број различитих сценарија а онда се добијени резултати статистички обраде. Да би овакав приступ био ефикасан, неопходна су средства за параметарску обраду која ће организовати, извршавати и управљати пословима на уређен начин.

7.5.3 Архитектура кластера

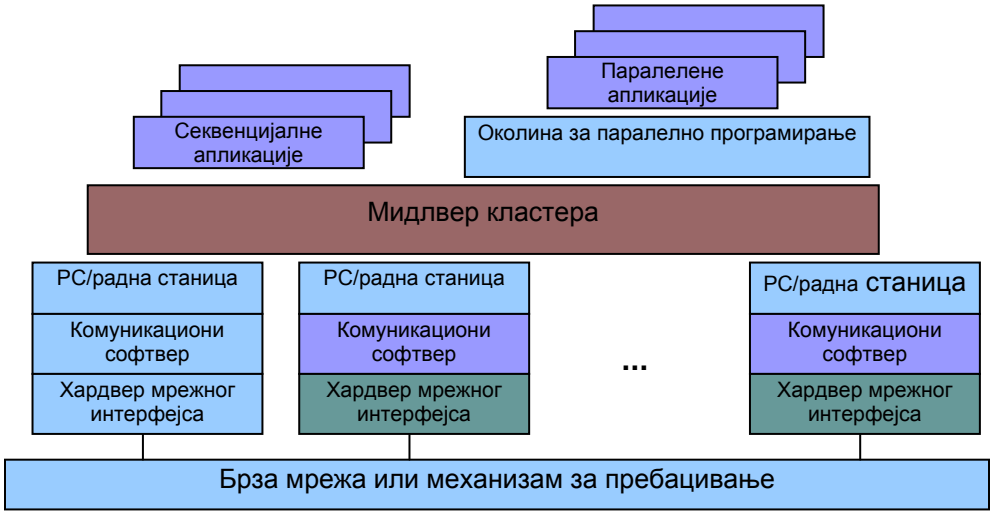
На слици 7.11 видимо типичну *архитектуру кластера*. Појединачни рачунари повезани су брзом LAN или хардвером за пребацивање. Сваки рачунар у стању је да ради независно. Уз то је *middleware* ниво софтвера инсталиран у сваком рачунару ради подршке кластерским операцијама. Мидлвер кластера обезбеђује кориснику униформну слику система која се назива *слика јединственог система*. Мидлвер је такође одговоран за поузданост, тј. расположивост система. Пожељно је да мидлвер пружа следеће услуге и функције:

- **Јединствена улазна тачка**: корисник се логује на кластер уместо на појединачни рачунар.
- **Јединствена хијерархија фајлова**: корисник види јединствену хијерархију директоријума испод истог корена.
- **Јединствена управљачка тачка**: постоји подразумевана радна станица која управља кластером.
- **Јединствена виртуелна мрежа**: сваки чвор може да приступи произвољној тачки у кластеру, чак иако се конфигурација кластера састоји од више повезаних мрежа. Рад се одвија у јединственој виртуелној мрежи
- **Јединствени меморијски простор**: дистрибуирана дељива меморија дозвољава програмима да деле променљиве.
- **Јединствени систем за управљање пословима**: корисник може да пријави посао не специфицирајући рачунар који ће га извршити.

- **Јединствен кориснички интерфејс**: заједнички графички интерфејс подржава све кориснике, без обзира са које радне станице су се пријавили на кластер.
- **Једнствени У/И простор**: сваки чвор може да приступи У/И уређају или диску при чему не мора да зна његову физичку локацију.
- **Јединствени простор процеса**: користи се униформна шема за идентификацију процеса. Процес на било ком чвору може да креира или приступа процесима на другим чворовима.
- **Проверавање (*checkpointing*)**: ова функција периодично памти стање процеса и међурезултате како би се омогућио опоравак после отказа.
- **Миграција процеса**: функција омогућује уравнотежавање оптерећења рачунара у кластеру.

Последње 4 функције односе се на расположивост а остале на пружање јединствене слике система.

Кластер треба да има и софтверске алате за ефикасно извршење програма који се могу паралелно извршавати.



Сл. 7.11. Архитектура кластера.

7.5.4 Поређење кластера и симетричних мултипроцесора

И кластери и SMP пружају конфигурацију са више процесора којом се подржавају захтевне апликације. Оба решења су комерцијално расположива мада SMP знатно дуже.

Главна снага SMP приступа лежи у томе што је управљање и конфигурисање SMP једноставније него код кластера. SMP је много ближи оригиналним једнопроцесорским моделима за које су скоро све апликације написане. Основна промена код преласка од једнопроцесора на SMP је у функцији планирања. Још једна предност SMP-а је што он захтева мање физичког простора и мање енергије за напајање. Најзад, предност SMP-а је што се ради о стабилном производу.

На дуже стазе, предности кластера ће вероватно резултирати ситуацијом да ће кластери бити доминантни на тржишту сервера високох перформанси. Кластери су далеко супериорнији у односу на SMP у смислу инкременталне и апсолутне скалабилности. Кластери су супериорни у односу на SMP и у погледу поузданости, јер се све компоненте система могу начинити високо редудантним.

7.6 NUMA

У новије време су се на тржишту појавили и мултипроцесори са *неуниформним меморијским приступом* (NUMA).

Што се тиче приступа меморији постоје три категорије:

- **UMA** – Сви процесори имају приступ свим деловима оперативне меморије помоћу *load* и *store* инструкција. Време приступа меморији за сваки процесор је исто без обзира којој се меморијској области приступ. У ову групу спадају SMP.
- **NUMA** – Сви процесори имају приступ свим деловима меморије помоћу *load* и *store* инструкција али време приступа зависи од дела меморије којем се приступа (што се опет разликује од процесора до процесора).
- **CC-NUMA** – NUMA систем са кеш кохеренцијом.

NUMA без кеш кохеренције су мање или више слични кластерима. Од интереса су CC-NUMA системи који се значајно разликују и од SMP и од кластера.

Код SMP система постоји практично ограничење у броју процесора који се могу употребити. Наиме, проблем лежи у повећаном саобраћају на магистрали. Ефикасна шема са кеш меморијом може да смањи овај саобраћај између произвољног процесора и главне меморије. Како број процесора расте, расте и саобраћај на магистрали. Међутим, и са кеш меморијама, са порастом броја процесора, обзиром да се сигнали за обезбеђивање кохеренције такође простиру магистралом, саобраћај на магистрали се увећава. Из тих разлога постоји број процесора иза кога магистрала постаје уско грло у систему те се перформансе деградирају ако би се даље повећавао број процесора. Овај лимит је код SMP обично на нивоу 16-64 процесора.

Једна од главних мотивација за развој кластерских система управо лежи у ограничењу броја процесора код SMP. Међутим, код кластера сваки чвор има приватну меморију и апликације не виде велику глобалну меморију. У суштини, кохеренција се одржава више софтверски него хардверски. Ова грануларност меморије утиче на перформансе а софтвер мора да буде писан за такво окружење како би се извукао максимум перформанси. Један од начина да се направе мултипроцесори са великим бројем процесора а да то личи на SMP јесте упараво NUMA.

Циљ NUMA организације је да системска меморија буде транспарентна, а да постоји више мултипроцесорских чворова при чему сваки од њих има сопствену магистралу или спрежну мрежу.

7.6.1 CC-NUMA организација

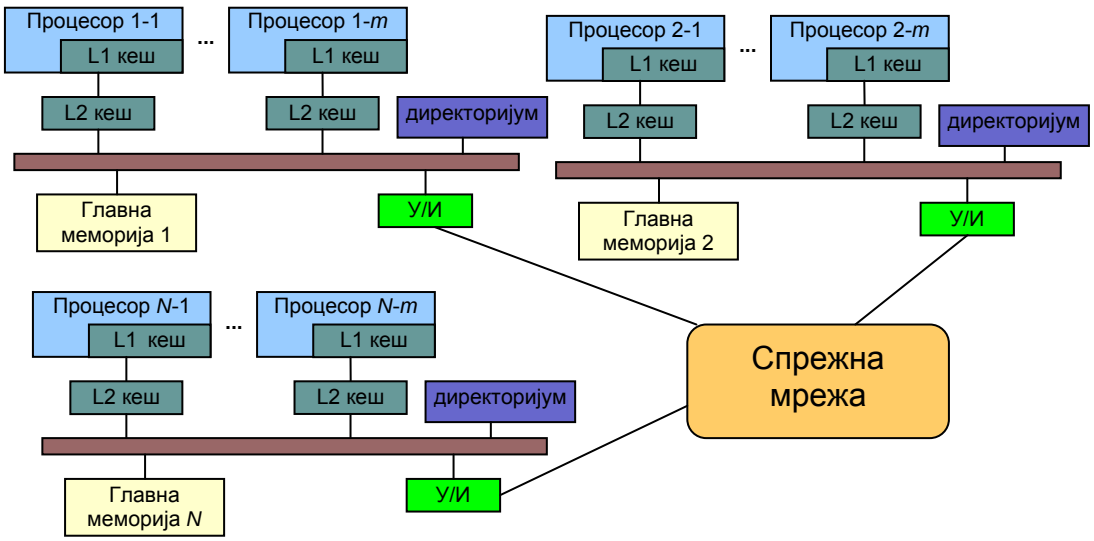
На слици 7.12 видимо типичну CC-NUMA организацију. Постоји више независних чворова где сваки од њих представља SMP организацију. Према томе, сваки чвори садржи више процесора, са L1 и L2 кешом и сопствену главну меморију. Чвор је основни градивни блок CC-NUMA организације. Чворови су повезани неком врстом спрежне мреже.

Сваки чвор садржи главну меморију, али са тачке гледишта процесора постоји јединствени адресни простор где свака локација има јединствену адресу у читавом систему. Када процесор иницира приступ меморији, ако потребна меморијска локација није у кешу тог процесора, онда L2 кеш иницира операцију прибављања. Ако је тражена линија у локалном делу главне меморије она се прибавља преко локалне магистрале. Но, ако је та линија у удаљеном делу главне меморије, тада се аутоматски захтев шаље остатку система како би се дотична линија прибавила преко спрежне мреже, затим предала локалној магистрали и, најзад, кешу прикљученом на локалну магистралу који је и упутио захтев. Све ове активности су аутоматске и транспарентне за процесор и његову кеш меморију.

Кеш кохеренција је главни проблем код ове конфигурације. Имплементације се могу разликовати у детаљима, али сваки чвор мора да садржи неку врсту директоријума који садржи локацију појединих делова меморије и стање кеша. Да би смо се боље упознали са овом шемом проучићемо један пример. Претпоставимо да процесор 3 у чвору 2 (P2-3) захтева меморијску локацију 798 која се налази у меморији чвора 1. Јавиће се следећа секвенца:

1. P2-3 издаје захтев за читањем локације 798 на "*snoopy*" магистралу чвора 2.
2. Директоријум чвора 2 прихвата захтев и открива да се та локација налази у чвору 1.
3. Директоријум чвора 2 шаље захтев чвору 1 који директоријум чвора 1 прихвата.
4. Директоријум чвора 1 ради као сурогат процесора P2-3 и захтева садржај локације 798.
5. Меморија чвора 1 одговара постављањем захтеваног податка на магистралу.
6. Директоријум чвора 1 преузима податак са магистрале.
7. Вредност се враћа директоријуму чвора 2.

8. Директоријум чвора 2 поставља податак на магистралу чвора 2 делујући као сурогат меморије која га је оригинално садржала.
9. Вредност се преузима са магистрале и смешта у кеш процесора P2-3 после чега је P2-3 преузима.



Сл. 7.12. CC-NUMA организација.

Претходна секвенца објашњава како се подаци читају из удаљене меморије коришћењем хардверских механизама који ову трансакцију чине транспарентном за процесор. Овај механизам мора да садржи неку врсту протокола за одржавање кеш кохеренције. Различити системи имају различите приступе, али у грубим цртама тај протокол претпоставља следеће:

- Као део претходне секвенце, директоријум чвора 1 чува информацију о томе да неки удаљени кеш има копију линије која садржи локацију 798.
- Затим, потребан је неки кооперативни протокол који ће бринути о модификацијама. Ако се модификација изврши у кешу о томе се обавештавају остали чворови.
- Директоријум сваког чвора прихвата такву информацију и на основу ње може да одреди да ли неки локални кеш има ту линију и, ако је тако, врши се њено инвалидирање. Линија ће остати инвалидирана све док се не јави *write-back*.
- Ако неки процесор (локални или удаљени) захтева инвалидирану линију, локални директоријум мора да изазове *write back* да би се меморија ажурирала пре него што се испоручи тражени података.

7.6.2 Предности и недостаци NUMA организације

Основна предност CC-NUMA је да пружа ефикасан рад на вишем нивоу паралелизма него SMP, а да се при том не захтевају значајне промене у софтверу. Са више NUMA чворова, саобраћај на магистрали појединог прозвољног чвора ограничен је на захтеве које та магистрала може да опслужи. Међутим, ако се значајан удео приступа меморији обавља у удаљеним чворовима перформансе почињу да опадају. Постоје разлози који нас наводе да се овај пад перформанси може избећи:

- Употреба кеш меморија нивоа L1 и L2 минимизира приступе меморији (локалној или удаљеној).
- Ако софтвер има добру просторну локалност и користи се виртуелна меморија тада ће се подаци који су потребни апликацији задржати на ограниченом броју често коришћених страница које се на почетку пуне у меморију која је локална за ту апликацију.

- Механизам виртуелне меморије се може побољшати тако што се у оперативни систем укључе механизми за миграцију страница ка чвору који их најчешће користи.

Постоје и недостаци овог приступа:

- Потребне су измене у оперативном систему и апликацијама да би са SMP-а могле да се користе на CC-NUMA (додела страница, додела процеса, уравнотежење оптерећења).
- Други недостатак тиче се расположивости. То је веома сложен проблем и зависи од конкретне имплементације.

Литература

[1] J. L. Hennesy, D. A. Patterson, *Computer Organization and Design*: *The Hardware/Software, interface*, 2/e, Morgan Kaufmann Publishers, inc. San Francisko, California, 1998.

[2] D. A. Patterson, J. L. Hennessy, *Computer Architecture*: *A Quantitative Approach*, 2/e, Morgan Kaufmann Publishers, inc. San Francisko, California, 1996.

[3] W. Stallings, *Computer Organization and Architecture*, 6/e, Prentice Hall, 2003.

Додатак А: Програмски језик Parallaxis

Програмски систем Parallaxis имплементиран је на Универзитету у Штутгарту 1989. године. Пројектован је тако да се може инсталирати и извршавати на IBM PC/AT, SUN Unix, Apollo DN3000 и Apple Macintosh рачунарима.

Програмски систем Parallaxis има два основне компоненте: Parallaxis преводилац и PARZ интерпретатор-симулатор.

Програм написан на овом језику садржи:

- Блокове секвенцијалне обраде.
- Блокове паралелне обраде.
- Опис архитектуре паралелне машине на којој се извршава.

Parallaxis преводилац преводи програм у псеудоасемблерски облик. На основу тога PARZ симулатор извршава паралелне и секвенцијалне делове програма на описаној машини. Превођење програма врши се командом

PA [opcije] ul_dat [-o izl_dat]

Изворни програм треба да има наставак **.p**. Преведени програм има наставак **.z**. Уколико је изостављено име датотеке izl_dat преведени програм ће бити у датотеци izl_dat.z. Извршење програма обавља се позивањем PARZ симулатора

PZ [opcije] ime_dat

Опције које се могу задати приликом превођења, односно извршавања, могу се добити позивом преводиоца, односно симулатора, без навођења аргумената.

А.1 Parallaxis модел вишепроцесорског система

Омогућено је паралелно програмирање независно од архитектуре на којој се програм извршава. Сваки програм садржи функционални опис вишепроцесорске структуре као и паралелни алгоритам за тако описану структуру. Апстрактни Parallaxis модел вишепроцесорске структуре симулира SIMD процесорско поље састављено од произвољног броја процесних елемената. У програму се дефинишу:

- Број процесних елемената.
- Димензионо уређење процесорског поља.
- Везе између процесних елемената.

Процесорска поља су хомогена, тј.

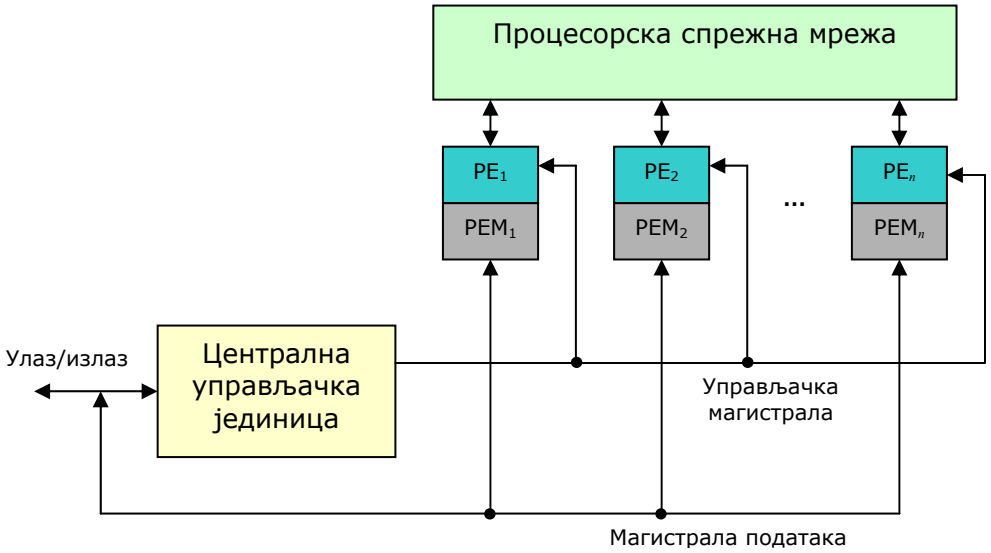
- процесорски елементи су функционално идентични, и
- имају исти број улазних и излазних портова.

У зависности од специфициране спрежне мреже процесни елементи су, преко портова, повезани или не са портовима других елемената.

На слици А.1 приказан је Parallaxis модел SIMD структуре рачунара. Карактеристике модела су:

- Симулира SIMD процесорску структуру која се састоји од централне управљачке јединице, променљивог броја процесних елемената и флексибилне спрежне мреже.
- Централна управљачка једница управља целокупним радом процесорског поља.
- Процесни елементи су идентични по процесорској и меморијској структури.
- Процесни елементи имају једнак број и исти распоред портова.
- Сваки процесни елемент има своју локалну меморију.
- Све операције у пољу извршавају се синхроно.

- Сви процесни елементи истовремено извршавају исту инструкцију над различитим подацима, тј. подацима из својих локалних меморија.
- Процесорска мрежа служи за повезивање процесних елемената и пренос података између њих.



Сл. А.1. Parallaxis модел SIMD структуре рачунара.

Избор топологије процесорске спрежне мреже зависи од алгоритма за дати проблем. На почетку сваког програма врши се специфицирање спрежне мреже да би се обезбедила статичност њене топологије за дату апликацију. Хост свим процесним елементима шаље истоветну инструкцију коју они извршавају над локалним подацима. Из тог разлога не постоји могућност индивидуалног програмирања процесних елемената.

A.2 Структура програма у Parallaxis-у

Сваки кориснички програм има следећу структуру:

```
SYSTEM Ime_programa;
--- definicija konstanti
--- definicija tipova podataka
--- specifikacija procesorske mreze
--- deklaracija skalarnih i vektorskih promenljivih
--- definicija potprograma
BEGIN
      --- telo programa
END Ime_programa.
```

Карактеристични делови програма су:

1. Специфицирање спрежне мреже процесорског поља.
2. Декларације скаларних и векторских података.
3. Извршење скаларних и векторских наредби.
4. Селеција процесних елемената.
5. Паралелни пренос података.
6. Редукција података.
7. Рад са потпрограмима.

Већина језичких конструкција преузета је из програмских језика Pascal и Modula2. Осим тога, постоје и нове језичке конструкције за дефинисање паралелних блокова обраде.

Присутне су и разлике при:

- дефинисању типова,
- одређивању приоритета оператора,
- увођењу константи и променљивих и
- имплементацији управљачких структура.

A.3 Кључне речи језика

Осим кључних речи које су веома сличне онима у језику Pascal, карактеристичне су оне намењене за паралелну обраду података:

CONFIGURATION
CONNECTION
SCALAR
VECTOR
PARALLEL
ENDPARALLEL
STORE
LOAD
PROPAGATE
REDUCE

A.4 Типови података

Елементарни типови података:

- Нумерички.
- Логички.
- Знаковни.

Нумерички типови:

- Целобројни.
- Реални.

Целобројни типови података:

- INTEGER у опсегу од -32768 до +32767.
- CARDINAL у опсегу од 0 до +32767.

Реални тип:

- REAL у опсегу, по апсолутној вредности, од 1.846329E-39 до 3.20282E+38

Логички тип:

- BOOLEAN

Знаковни тип:

- CHAR

На основу елементарних типова дефинишу се и изведени типови.
Структурни типови података су:

- Поља.
  Једнодимензиона поља дефинишу се као
  ARRAY *opseg* OF *tip*;
  док се дводимензиона поља могу дефинисати на два начина
  ARRAY *opseg*1*, opseg*2 OF *tip*;
  ARRAY *opseg*1 OF ARRAY *opseg*2 OF *tip*;
- Записи.
  Записи се дефинишу на следећи начин
  RECORD
  lista_promenljivih: *tip*;
  …
  END;
- Скупови.

Скупови немају ограничења по опсегу а дефиниција им је
SET OF *tip*;

A.5 Приоритет оператора

Приоритет опратора је сличан као и у многим другим програмским језицима. Овде су побројане групе оператора према опадању приоритета.

1. NOT, - (унарни), + (унарни);
2. ^;
3. *, /, DIV, MOD, AND или &;
4. +, -, OR;
5. =, <> или #, <, <=, >, >=, IN;

A.6 Управљачке структуре

IF … THEN … [ELSEIF … THEN …] [ELSE …] END
CASE … OF … [ELSE …] END
WHILE … DO … END
REPEAT … UNTIL
FOR … TO … [BY …] DO … END
LOOP … END
WITH … DO … END

Карактеристично је да је, за разлику од већине других језика, дозвољена употреба двоструких услова типа

IF (1 < x< 100) THEN …

A.7 Типови спрежних мрежа

Програмски језик Parallaxis дозвољава специфицирање основних топологија спрежних мрежа:

- Линеарна листа.
- Матрица.
- Хексагонална мрежа.

Ако се ове спрежне мреже допуне спрежним функцијама које повезују крајње елементе у пољу могу се описати и сложеније топологије мрежа:

- Циклична листа.
- Торус.

Уз помоћ компонованих спрежних функција описују се сложене мреже као:

- Бинарно стабло.
- Квадратно стабло.
- Мрежа “потпуно мешање – замена”.

Коришћење параметарских спрежних функција где се смер преноса одређује на основу вредности параметра омогућује специфицирање мрежа високе симетрије као што је *хиперкоцка*.

Конфигурација процесорског поља дефинише се спецификацијом CONFIGURATION, док се спрежна мрежа описује спецификацијом CONNECTION.

Како процесорско поље може бити једнодимензионо или вишедимензионо, у спецификацији CONFIGURATION се, осим имена конфигурације процесорског поља, наводе опсези по свим димензијама одвојени зарезом. Осег се може навести на два начина

- [a] за опсег од 0 до a-1, (a > 0),
- [a..b], за опсег од a до b (a ≤ b).

На пример, процесорско поље које има топологију линеарне листе са 10 елемената, означених од 5 до 14, се описује као:

CONFIGURATION lista[5..14];

За процесорско поље типа матрице (рецимо 3×4) спецификација је:

CONFIGURATION matrica[1..3],[1..4];

У спецификацији CONNECTION се одређују везе између процесних елемената тако што се наведу спрежне функције. Једна спрежна функција дефинише начин повезивања једног излазног порта сваког процесног елеменат. Број спрежних функција које се наводе једнак је броју излазних портова. Спрежне функције се раздвајају помоћу знака ';'. За сваку функцију наводе се, редом, име излазног порта, знак ':', име конфигурације са описом предајног елемента, ознака везе, име конфигурације са описом пријемног елемента, знак '.' и име улазног порта. Ознака једносмерне везе је '->' док је ознака двосмерне везе '<->'. Спецификација CONNECTION може бити изостављена (у том случају се наводи само кључна реч праћена знаком ';') док се спецификација CONFIGURATION мора навести.

Примера ради, опис спрежне мреже за линерану листу је

```
CONNECTION desno: lista[i]->lista[i+1].desno;
```

где је desno истовремено улазни и излазни порт у сваком од процесних елемената. За матрицу би одговарајућа спецификација била облика:

```
CONNECTION  desno: matrica[i,j]->matrica[i,j+1].levo;
            levo: matrica[i,j]->matrica[i,j-1].levo;
            gore: matrica[i,j]->matrica[i+1,j].dole;
            dole: matrica[i,j]->matrica[i-1,j].gore;
```

A.8 Потпрограми општег типа

Структура поптпрограма општег типа је

```
PROCEDURE Ime_pp(deklaracija argumenata);
--- deklaracija lokalnih promenljivih
   BEGIN
   --- telo potprograma
   END Ime_pp;
```

При декларацији аргумената за сваки аргумент се наводи:

- кључна реч VAR за позив по референци,
- име аргумента,
- знак ‘:’ и
- тип аргумента.

Декларације аргумената одвајају се знаком ‘;’.

У језику Parallaxis постоји 28 стандардних потпрограма општег типа. За навођење синтаксе позива ових потпрограма користићемо следеће ознаке:

ord – набројиви, изведени и целобројни типови података;

elem – тип елемента скупа;

skup – тип SET OF elem;

niz – променљиве типа ARRAY OF char.

- DEC(SCALAR VAR x: ord)
- DEC(VECTOR VAR x: ord)

Умањује вредност променљиве x за 1 или додељује променљивој вредност њеног непосредног претходника.

- DEC(SCALAR VAR x:ord;SCALAR n:INTEGER)
- DEC(VECTOR VAR x:ord;VECTOR n:INTEGER)

Умањује вредност променљиве x за n или додељује променљивој вредност њеног n-тог претходника.

- INC(SCALAR VAR x: ord)
- INC(VECTOR VAR x: ord)

Увећава вредност променљиве x за 1 или додељује променљивој вредност њеног непосредног следбеника.

- INC(SCALAR VAR x:ord;SCALAR n:INTEGER)
- INC(VECTOR VAR x:ord;VECTOR n:INTEGER)

Увећава вредност промењиве *x* за *n* или додељује промењивој вредност њеног *n*-тог следбеника.

- EXCL(SCALAR VAR x:skup;SCALAR y:elem)
- EXCL(VECTOR VAR x:skup;SCALAR y:elem))

Уклања елемент *y* из скупа *x*.

- INCL(SCALAR VAR x:skup;SCALAR y:elem)
- INCL(VECTOR VAR x:skup;SCALAR y:elem)

Уводи елемент *y* у скуп *x*.

- Write(SCALAR c:char)

Знак *c* се шаље на активни излазни канал.

- WriteInt(SCALAR i:INTEGER;SCALAR n:CARDINAL)

Слање целог броја *i* на активни излазни канал са записом од *n* знакова.

- WriteCard(SCALAR i,n:CARDINAL)
- WriteReal(SCALAR r:REAL;SCALAR n:CARDINAL)
- WriteFixPt(SCALAR r:REAL;SCALAR n,p:CARDINAL)
- WriteBool(SCALAR b:BOOLEAN)
- WriteString(SCALAR s:niz)
- Writeln
- Read(SCALAR VAR c:CHAR)
- ReadInt(SCALAR VAR i:INTEGER)
- ReadCard(SCALAR VAR c:CARDINAL)
- ReadReal(SCALAR VAR r:REAL)
- ReadBool(SCALAR VAR b:BOOLEAN)
- ReadString(SCALAR VAR s:niz)
- OpenInput(SCALAR s:niz)
- OpenOutput(SCALAR s:niz)
- CloseInput
- CloseOutput

A.9 Функцијски потпрограми

Структура функцијског поптпрограма је

```
PROCEDURE Ime_pp(dekl_arg): dekl_rez;
--- deklaracija lokalnih promenljivih
    BEGIN
    --- telo potprograma
    RETURN(rezultat);
    END Ime_pp;
```

Декларација аргумената је иста као и код потпрограма општег типа осим што не постоји кључна реч VAR. У декларацији резултата наводи се да ли је резултат скаларни или векторски, као и тип резултата. Аргумент наредбе RETURN је излазни аргумент потпрограма и може бити промењива или израз.

У програмском језику Parallaxis постоје 32 стандардна функцијска потпрограма. Осим раније коришћених ознака користићемо и:

tip – име типа податка

port – име порта

- COS(SCALAR r:REAL):SCALAR REAL;
- COS(VECTOR r:REAL):VECTOR REAL;
- SIN(SCALAR r:REAL):SCALAR REAL;
- SIN(VECTOR r:REAL):VECTOR REAL;
- TAN(SCALAR r:REAL):SCALAR REAL;
- TAN(VECTOR r:REAL):VECTOR REAL;
- ARCSIN(SCALAR r:REAL):SCALAR REAL;
- ARCSIN(VECTOR r:REAL):VECTOR REAL;
- ARCCOS(SCALAR r:REAL):SCALAR REAL;
- ARCCOS(VECTOR r:REAL):VECTOR REAL;
- ARCTAN(SCALAR r:REAL):SCALAR REAL;
- ARCTAN(VECTOR r:REAL):VECTOR REAL;

у опсегу $[-\pi/2, +\pi/2]$

- ARCTAN2(SCALAR r1,r2:REAL):SCALAR REAL;
- ARCTAN2(VECTOR r1,r2:REAL):VECTOR REAL;

$arctg(r1/r2) \in [-\pi, +\pi]$

- EXP(SCALAR r:REAL):SCALAR REAL;
- EXP(VECTOR r:REAL):VECTOR REAL;
- LN(SCALAR r:REAL):SCALAR REAL;
- LN(VECTOR r:REAL):VECTOR REAL;
- SQRT(SCALAR r:REAL):SCALAR REAL;
- SQRT(VECTOR r:REAL):VECTOR REAL;
- ABS(SCALAR x:INTEGER):SCALAR CARDINAL;
- ABS(VECTOR x:INTEGER):VECTOR CARDINAL;
- ABS(SCALAR x:REAL):SCALAR REAL;
- ABS(VECTOR x:REAL):VECTOR REAL;
- FLOAT(SCALAR i:INTEGER):SCALAR REAL;
- FLOAT(VECTOR i:INTEGER):VECTOR REAL;
- TRUNC(SCALAR r:REAL):SCALAR INTEGER;
- TRUNC(VECTOR r:REAL):VECTOR INTEGER;
- CAP(SCALAR c:CHAR):SCALAR CHAR;
- CAP(VECTOR c:CHAR):VECTOR CHAR;

Претвара знак у велико слово.

- CHR(SCALAR i:INTEGER):SCALAR CHAR;
- CHR(VECTOR i:INTEGER):VECTOR CHAR;
- ORD(SCALAR i:ord):SCALAR INTEGER;
- ORD(VECTOR i:ord):VECTOR INTEGER;
- ORD(SCALAR i:CHAR):SCALAR INTEGER;
- ORD(VECTOR i:CHAR):VECTOR INTEGER;
- EVEN(SCALAR i:INTEGER):SCALAR BOOLEAN;
- EVEN(VECTOR i:INTEGER):VECTOR BOOLEAN;
- ODD(SCALAR i:INTEGER):SCALAR BOOLEAN;
- ODD(VECTOR i:INTEGER):VECTOR BOOLEAN;

- MAX(TIP ord):SCALAR ord;

Највећи елемент који припада типу ord.

- MIN(TIP ord):SCALAR ord;

Најмањи елемент који припада типу ord.

- SIZE(SCALAR x:TIP):SCALAR INTEGER;
- SIZE(VECTOR x:TIP):VECTOR INTEGER;

Враћа број меморијских локација које заузима променљива x.

- STRCMP(SCALAR s1,s2:niz):SCALAR INTEGER;
- STRCMP(VECTOR s1,s2:niz):VECTOR INTEGER;

Пореди знаковне низове s1 и s2; резултат поређења је +1, 0 или -1 у зависности од тога да ли је низ s1 у лексикографском смислу већи, једнак или мањи од низа s2.

- STREQ(SCALAR s1,s2:niz):SCALAR BOOLEAN;
- STREQ(VECTOR s1,s2:niz):VECTOR BOOLEAN;

Пореди знаковне низове s1 и s2; враћа TRUE ако је s1 у лексикографском смислу већи од низа s2; иначе FALSE.

- SBRANDOM():SCALAR BOOLEAN;
- VBRANDOM():VECTOR BOOLEAN;
- SCRANDOM():SCALAR CHAR;
- VCRANDOM():VECTOR CHAR;
- SIRANDOM():SCALAR INTEGER;
- VIRANDOM():VECTOR INTEGER;
- SRRANDOM():SCALAR REAL;
- VRRANDOM():VECTOR REAL;
- VAL(TIP ord;SCALAR n:INTEGER):SCALAR ord;
- VAL(TIP CHAR;SCALAR n:INTEGER):SCALAR CHAR;
- VAL(TIP ord;VECTOR n:INTEGER):VECTOR ord;
- VAL(TIP CHAR;VECTOR n:INTEGER):VECTOR ord;

Преводи цео број *n* у одговарајући елемент редног типа. Важи

ORD(VAL(ord,n)) = n
VAL(CHAR,n) = CHAR(n)

- IN_Connected(PORT ul_port):VECTOR BOOLEAN;
- OUT_Connected(PORT izl_port):VECTOR BOOLEAN;
- IN_LineConnected(PORT izl_port,ul_port):VECTOR BOOLEAN;
- OUT_LineConnected(PORT izl_port,ul_port):VECTOR BOOLEAN;

Литература

[1] D. Milosavljević, *Praktikum za vežbe na računaru iz predmeta Paralelni računarski sistemi*, Elektronski fakultet, 1995.

Додатак Б: Архитектура IA-64

Фамилија микропроцесора, започета још пре 25 година са 8086, а која је била најуспешнији рачунарски производ који се икада појавио, се са рачунаром Pentium 4 полако ближи крају. Intel се удружио са познатом компанијом Hewlett-Packard (HP) да би развио нову 64-битну архитектуру названу IA-64. Ова архитектура није 64-битно проширење Intel-ове 32-битне фамилије нити адаптација HP-ove 64-битне архитектуре PA-RISC. Ради се о новој архитектури која је пројектована на основу вишегодишњих истраживања ове две компаније као и више универзитета. Архитектура IA-64 користи кола велике брзине која су се јавила у најновијој генерацији микрочипова и направљена је са намером да на систематски начин користи паралелизам. Ова архитектура је значајан помак у коришћењу суперскаларности која је доминантна код новијих процесора.

Б.1. Мотивација

Основни концепти који карактеришу архитектуру IA-64 су:

- Паралелизам на нивоу инструкција који је експлицитно изражен у скупу машинских инструкција уместо да њиме управља процесор у току извршења.
- Дуге или веома дуге речи инструкција (LIW/VLIW - /Very/ Long Instruction Words).
- Предикација гранања (овај појам не треба мешати са предикцијом, тј. предвиђањем гранања).
- Спекулативно пуњење.

Intel и HP ову комбинацију концепата, односно одговарајућу технологију, називају **EPIC** (Explicitly Parallel Instruction Computing). IA-64 је архитектура која је имплементирана коришћењем EPIC технологије. Први Intel-ов производ заснован на IA-64 назван је **Itanium**.

У табели Б.1 приказане су основне разлике између IA-64 и традиционалних суперскаларних приступа.

Таб. Б.1. Поређење традиционалног суперскаларног приступа и IA-64 архитектуре.

Суперскларани	**IA-64**
Инструкције у стилу RISC-а, једна по речи	Инструкције у стилу RISC-а груписане по три
Више паралелних извршних јединица	Више паралелних извршних јединица
Измена редоследа и оптимизација инструкција у време извршења	Измена редоследа и оптимизација инструкција у време компилације
Предвиђање гранања са спекулативним извршењем по једној грани	Предвиђање гранања са спекулативним извршењем по обе гране
Пуњење података из меморије само када су потребни при чему се покушава да се најпре пронађу у кешу	Спекулативно пуњење података пре него што су потребни уз покушај да се најпре пронађу у кешу

За Intel је прелазак на нову архитектуру која није хардверски компатибилна са фамилијом x86 представљао велику одлуку. Међутим, технологија је диктирала доношење овакве одлуке. Када је фамилија x86 настала крајем 70-тих чип процесора је имао десетине хиљада транзистора и у суштини је представљао скаларни уређај, што значи да су се инструкције обрађивале по једна истовремено са мало или нимало проточности. Како је број транзистора нарастао на стотине хиљада средином 80-тих, Intel је увео проточност. У међувремену су други произвођачи покушавали да искористе повећање густине и брзине чипова помоћу RISC приступа, што је омогућило ефикаснију проточност а касније и комбинацију суперскаларног и RISC приступа која укључује више извршних јединица. Код Pentiuma је Intel направио скроман покушај да употреби суперскаларну технику

омогућујући да се две CISC инструкције извршавају истовремено. Потом су Pentium Pro и Pentium II до Pentium 4 поседовали пресликавање са CISC инструкција на микрооперације које су биле налик RISC инструкцијама а такође су агресивније коришћене суперскаларне технике. Овакав приступ је омогућио ефикасну употребу чипова са милион транзистора. Али, код следеће генерације чипова са десетинама милиона транзистора, која треба да уследи после Pentiuma, потребан је приступ који ће ефикасно искористити могућности хардвера.

Пројектанти процесора су имали на располагању неколико могућности да искористе тај велики број транзистора. Један приступ био би да се додатни транзистори искористе за уградњу већих *on-chip* кеш меморија. Већа кеш меморија може да побољша перформансе до извесног степена али се на крају дође до тачке од које повећање капацитета кеш меморије не даје одговарајуће повећање фактора поготка. Алтернативни приступ је да се повећа степен суперскаларности додавањем извршних јединица. Међутим, то многи усложњава процесор и отежава управљање, јер захтева побољшање у смислу предвиђања гранања, извршења изван редоследа и повећања броја проточних степени. Већи број проточних степени значи већи губитак када се изврши погрешно предвиђање гранања. Извршење изван редоследа захтева већи износ преименовања регистара и сложену логику за откривање и разрешавање међузависности. Резултат ових проблема је да се код најбољих савремених процесора извршава највише шест инструкција по циклусу па и мање од тога.

Да би савладали овај проблем Intel и HP су дефинисали општи приступ који омогућава ефикасно искоришћење процесора са више паралелних извршних јединица. У суштини овог приступа налази се концепт експлицитног паралелизма. Код овог приступа компилатор статички планира инструкције у време компилације уместо да препусти процесору да врши динамичко планирање у време извршења. Компилатор одређује које се инструкције могу извршавати паралелно и ту информацију укључује у машинске инструкције. Процесор користи те информације у паралелном извршењу. Предност овог приступа је што EPIC процесор не захтева превише компликовану логику као што је то случај код суперскаларног процесора са извршењем изван редоследа. Осим тога, док процесор има на располагању тек неколико наносекунди да одреди потенцијални паралелизам, компилатор има на располагању неколико редова величине дуже време да испита кôд натенане сагледавајући при том програм као целину.

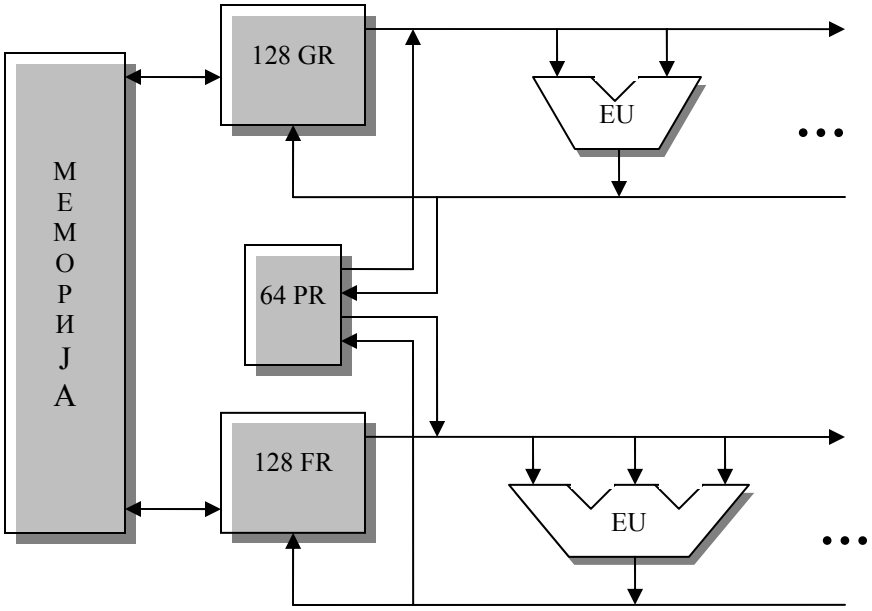
Б.2. Општа организација

Као и свака друга архитектура, IA-64 се може имплементирати у више организација. Слика Б.1 приказује једну општу организацију ове архитектуре. Кључне карактеристике ове организације су следеће:

- **Велики број регистара:** Формат инструкција код IA-64 претпоставља коришћење 256 регистара и то 128 64-битних регистара опште намене за целе и логичке вредности, 128 82-битних регистара за вредности у покретном зарезу или за графику и 64 једнобитна регистра за предикатско извршење.
- **Више извршних јединица:** Типичне суперскаларне машине које су данас расположиве на тржишту могу да имају и до четири паралелна проточна система, што значи да имају четири паралелене извршне јединице како за целобројна израчунавања тако и за израчунавања у покретном зарезу. Очекује се да ће IA-64 бити имплементирана на системима са 8 или више паралелних јединица.

Регистарско поље је прилично велико у поређењу са већином RISC и суперскаларних машина. Разлог за ово налази се у чињеници да је велики број регистара потребан да би се подржао висок степен паралелизма. Код традиционалних суперскаларних машина машински и асемблерски језик користи мали број видљивих регистара које процесор пресликава на већи број преосталих регистара користећи технике за преименовање регистара и анализу међузависности. Због намере да се паралелизам учини експлицитним и процесор ослободи терета преименовања регистара и анализе међузависности, неопходан је велики број експлицитних регистара.

Број извршних јединица је у функцији броја транзистора који је расположив у конкретној имплементацији. Процесор ће искористити паралелизам до крајњих расположивих граница. Примера ради, ако низ инструкција машинског језика указује да се 8 целобројних инструкција може извршити паралелно, процесор са четири проточне целобројне извршне јединице ће ове инструкције извршити у два дела по четири. Процесор који има 8 проточних система могао би ове инструкције да изврши симултано.



GR – Регистри опште намене или интерни регистри
FR – Регистри у покретном зарезу или графички регистри
PR – Једнобитни предикатски регистри
EU – Извршна јединица

Сл. Б.1. Општа организација архитектуре IA-64.

Архитектуром IA-64 дефинисана су четири типа извршних јединица:

- **I–јединица:** Намењена је целобројној аритметици, померању са сабирањем, логичким операцијама, инструкцијама поређења и целобројним мултимедијалним инструкцијама.
- **M-јединица:** Намењена је инструкцијама типа *load* и *store* као и неким целобројним ALU операцијама.
- **B-јединица:** Намењена је инструкцијама гранања.
- **F-јединица:** Намењена је инструкцијама у покретном зарезу.

Свака од инструкција архитектуре IA-64 сврстана је у једну од шест група. Табела Б.2 приказује те типове инструкција и типове извршних јединица на којима се те инструкције могу извршити.

Таб. Б.2. Однос између типова инструкција и типова извршних јединица.

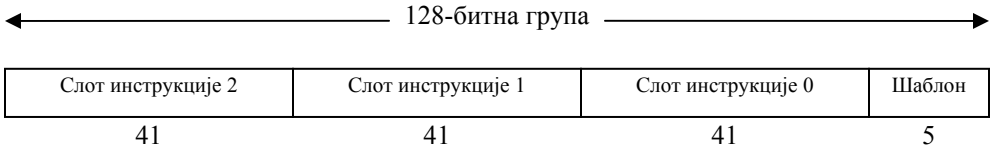
Тип инструкције	**Опис**	**Тип извршне јединице**
A	Целобројне ALU инструкције	I-јединица или M-јединица
I	Целобројне не-ALU инструкције	I-јединица
M	Меморијске	M-јединица
F	Инструкције у покретном зарезу	F-јединица
B	Гранања	B-јединица
L + X	Проширене инструкције	I-јединица

Б.3. Предикација, спекулација и софтверска проточност

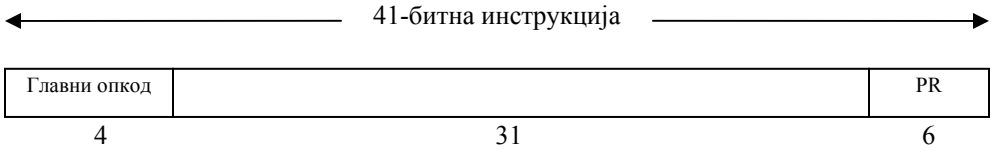
Извршићемо преглед кључних карактеристика архитектуре IA-64 које представљају подршку паралелизму на нивоу инструкција. Почећемо са форматима инструкција који су предвиђени код ове архитектуре.

Б.3.1. Формат инструкција

IA-64 дефинише 128-битну групу (***bundle***) која садржи три инструкције које се називају ***слогови*** и поље шаблона (слика Б.2а). Процесор може да прибавља инструкције по једну или више група истовремено. У свакој групи налазе се три инструкције. Поље шаблона садржи информације које указују које инструкције могу да се извршавају паралелно. Интерпретација поља шаблона није ограничена на једну групу. Процесор анализира више група како би одредио које инструкције могу да се изврше паралелно. На пример, низ инструкција може да буде такав да се осам инструкција може извршити у паралели. Компилатор ће преуредити инструкције тако да се ових осам инструкција нађе у суседним групама и поставити битове шаблона тако да процесор зна да не постоје међузависности међу овим инструкцијама.



а) Група инструкција код IA-64.



б) Општи формат инструкције код IA-64.

в) Типични формат инструкције код IA-64.

PR – Предикатски регистар
GR – Регистар опште намене или регистар покретног зареза

Сл. Б.2. Формат инструкције код IA-64.

Груписане инструкције не морају да одговарају оригиналном програмском редоследу. Даље, због флексибилности поља шаблона, компилатор може да смести независне и зависне инструкције у исту групу. Насупрот неким претходним VLIW архитектурама, код IA-64 није потребно убацивати безефектне инструкције (NOP) да би се попунила група.

У табели Б.3 приказан је начин интерпретације могућих вредности за 5-битно поље шаблона (неке од вредности су резервисане и нису тренутно у употреби). Ово поље има две намене:

1. Специфицира пресликавање слотова инструкција у типове извршних јединица. Нису расположива сва могућа пресликавања.
2. Указује на присуство ***стопера***. Стопер указује хардверу да једна или више инструкција пре стопера има неку врсту зависности по ресурсима са једном или више инструкција после стопера. У табели дебела вертикална црта означава стопер.

Таб. Б.3. Кодирање поља шаблона и пресликавање скупа инструкција.

Шаблон	Слот 0	Слот 1	Слот 2
00	M-јединица	I-јединица	I-јединица
01	M-јединица	I-јединица	I-јединица
02	M-јединица	I-јединица	I-јединица
03	M-јединица	I-јединица	I-јединица
04	M-јединица	L-јединица	X-јединица
05	M-јединица	L-јединица	X-јединица
08	M-јединица	M-јединица	I-јединица
09	M-јединица	M-јединица	I-јединица
0A	M-јединица	M-јединица	I-јединица
0B	M-јединица	M-јединица	I-јединица
0C	M-јединица	F-јединица	I-јединица
0D	M-јединица	F-јединица	I-јединица
0E	M-јединица	M-јединица	F-јединица
0F	M-јединица	M-јединица	F-јединица
10	M-јединица	I-јединица	B-јединица
11	M-јединица	I-јединица	B-јединица
12	M-јединица	B-јединица	B-јединица
13	M-јединица	B-јединица	B-јединица
16	B-јединица	B-јединица	B-јединица
17	B-јединица	B-јединица	B-јединица
18	M-јединица	M-јединица	B-јединица
19	M-јединица	M-јединица	B-јединица
1C	M-јединица	F-јединица	B-јединица
1D	M-јединица	F-јединица	B-јединица

Свака инструкција има 41-битни фиксирани формат (слика Б.2б). Ово је нешто дуже него код традиционалне 32-битне дужине која се среће код RISC и суперскаларних RISC машина, али је знатно краће од 118-битних микрооперација код процесора Pentium 4. Два фактора су утицала на увођење додатних битова. Прво, IA-64 користи више регистара него типична RISC машина: 128 за целобројне вредности и 128 регистара за вредности у покретном зарезу. Друго, да би се прилагодила техници предикатског извршења IA-64 машина садржи 64 предикатска регистра. Њихова употреба биће објашњена касније.

Слика Б.2в приказује типични формат инструкције са више детаља. Све инструкције садрже 4-битни главни опкод и референцу на предикатски регистар. Мада садржај поља главног опкода може бити само једна од 16 могућности, интерпретација овог поља зависи од вредности шаблона и локације инструкције у оквиру групе (види табелу Б.3), тако да постоји више расположивих кôдова операција. Типична инструкција такође садржи три поља која референцирају регистре, а преосталих 10 битова представљају друге неопходне информације за спецификацију инструкције.

Б.3.2. Формат асемблерског језика

Као и у случају сваког другог скупа машинских инструкција, асемблерски језик представља симболичку презентацију тих инструкција која олакшава посао програмеру. Асемблер сваку од асемблерских инструкција преводи у одговарајућу 41-битну IA-64 машинску инструкцију. Општи формат инструкције асемблерског језика је

```
[qp]  mnemonic[.comp]   dest=srcs
```

где

qp Специфицира 1-битни предикатски регистар који квалификује инструкцију. Ако је вредност у том регистру 1 (*true*) у време извршења, инструкција се извршава и резултат утврђује хардверски. Ако је та вредност *false* резултат се не израчунава а инструкција се одбацује. Већина IA-64 инструкција се могу квалификовати предикатом али то није увек потребно. Да би се означила инструкција која није

предикатска, вредност *qp* треба да је 0 а предикатски регистар 0 увек садржи вредност 1.

menemonic Специфицира име инструкције.

comp Специфицира један или више комплетера инструкције који су одвојени тачкама и који се користе да квалификују мнемоник. Не захтевају све инструкције коришћење комплетера.

dest Специфицира један или више одредишних операнада, при чему је типичан случај да постоји само једно одредиште.

srcs Специфицира један или више изворних операнада. Највећи број инструкција има два или више изворних операнада.

У свакој линији се низ знакова десно од "//" третира као коментар. Групе инструкција и стопери раздајају се двоструким знаком тачка и зарез. Група инструкција се дефинише као секвенца инструкција које немају RAW или WAW зависности. Процесор може да изда ове инструкције без хардверских провера регистарских међузависности. Ево једног једноставног примера:

```
ld8       r1 = [r5] ;;       // Prva grupa
add       r3 = r1, r4        // Druga grupa
```

Прва инструкција чита 8-бајтну вредност из меморијске локације чија адреса је у регистру r5 и смешта је у регистар r1. Друга инструкција сабира садржаје регистара r1 и r4 и резултат смешта у r3. Због тога што друга инструкција зависи од вредности у r1, која се мења првом инструкцијом, ове две инструкције не могу да буду у истој групи за паралелно извршење.

Ево нешто сложенијег примера, са више регистарских међузависности.

```
ld8       r1 = [r5]          // Prva grupa
sub       r6 = r8, r9 ;;     // Prva grupa
add       r3 = r1, r4        // Druga grupa
st8       [r6] = r12         // Druga grupa
```

Последња инструкција смешта садржај r12 у меморијску локацију чија адреса је у r6.

Сада ћемо извршити преглед четири кључна механизма у архитектури IA-64 којима се подржава паралелизам на нивоу инструкција:

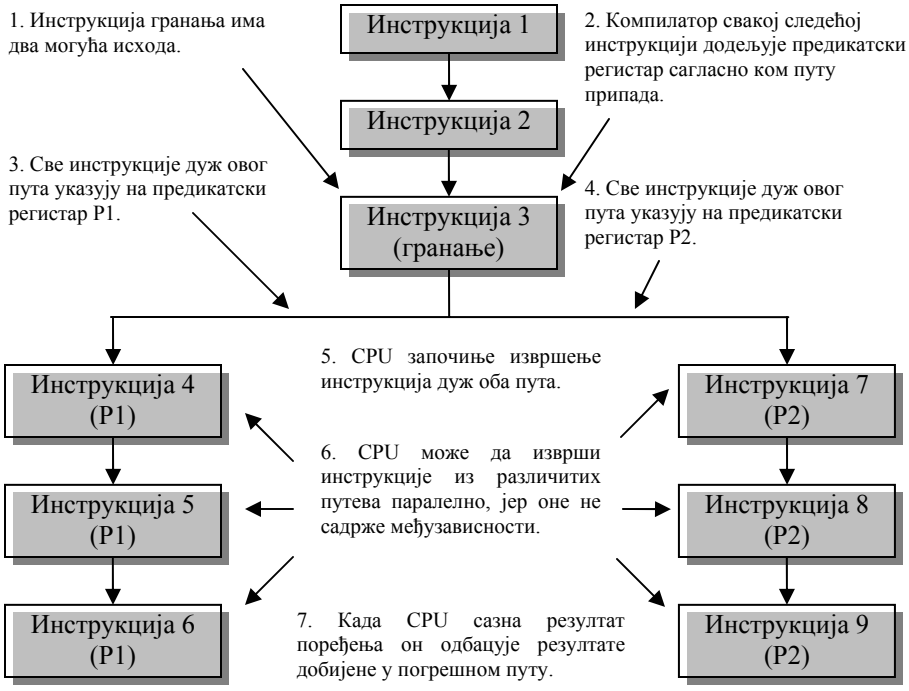
- Предикација
- Управљачка спекулација
- Спекулација података
- Софтверска проточност

Б.3.3. Предикатско извршење

Предикација је техника код које компилатор одређује које се инструкције могу извршити паралелно. У току овог процеса компилатор елиминише гранања из програма тако што користи условно извршење. Типичан пример из вишег програмског језика је **if-then-else** конструкција. Традиционални компилатор убацује условно гранање у **if** тачки конструкције. Ако услов има један од два могућа исхода, гранање се не врши и извршава се блок инструкција који представља **then** грану; на крају овог блока јавља се безусловно гранање на место иза следећег блока који представља **else** грану. Ако је услов имао други могући резултат, гранање се врши прескачући **then** грану и наставља се са **else** граном. Два тока инструкција се поново уједињују на крају **else** блока. Међутим, компилатор за IA-64 ће поступити на следећи начин (види слику Б.3):

1. У **if** тачки програма, убацује се инструкција поређења која креира два предиката. Ако је поређење истинито, први предикат се поставља на *true* а други на *false*. Ако је поређење неистинито, онда се први предикат се поставља на *false* а други на *true*.
2. Свакој инструкцији у **then** грани додаје се референца на предикатски регистар који садржи вредност првог предиката, док се свакој инструкцији у **else** грани додаје референца на предикатски регистар који садржи вредност другог предиката.
3. Процесор наставља извршење инструкција по оба пута. Када резултат поређења постане познат, одбацују се резултати добијени дуж једног од два пута извршења а уважавају резултати по другом путу. Ово омогућава процесору да допрема инструкције

на основу оба пута извршења у проточни систем а да при том нема чекања на завршетак операције поређења.



Компилатор може да преуреди инструкције у овом редоследу, упарујући инструкције 4 и 7, 5 и 8 и 6 и 9 ради паралелног извршења.

Инструкција 1	Инструкција 2	Инструкција 3
Инструкција 4	Инструкција 7	Инструкција 5
Инструкција 8	Инструкција 6	Инструкција 9

Сл. Б.3. Предикација.

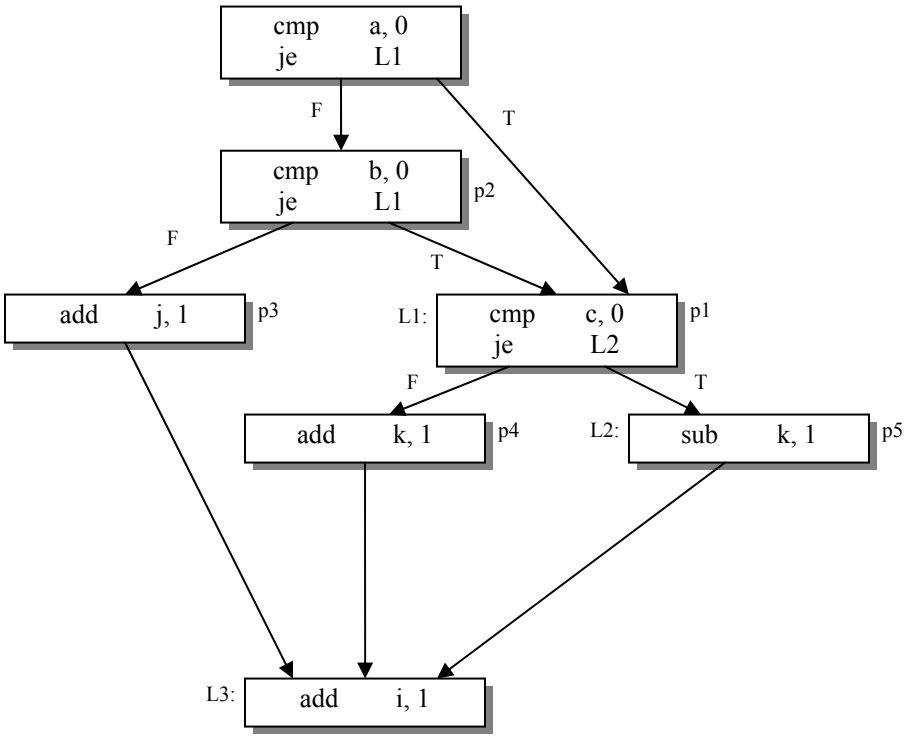
Као пример размотримо следећи изворни кôд:

```
if (a&&b)
   j = j + 1;
else
   if (c)
         k = k + 1;
   else
         k = k – 1;
i = i + 1;
```

Два **if** исказа заједно креирају три могућа пута извршења. Ово се може копиловати у следећи кôд (користимо асемблерски језик за Pentium). Програм ће имати три инструкције условног гранања и једну инструкцију безусловног гранања.

```
            cmp   a, 0  ; poredjenje a sa 0
            je    L1    ; grananje na L1 ako je a=0
            cmp   b, 0
            je    L1
            add   j, 1  ; j = j + 1;
            jmp   L3
L1:         cmp   c, 0
            je    L2
            add   k, 1  ; k = k + 1
            jmp   L3
L2:         sub   k, 1  ; k = k - 1
L3:         add   i, 1  ; i = i + 1
```

На слици Б.4 приказан је дијаграм тока за овај асемблерски кôд. На дијаграму можемо видети да је асемблерски програм подељен у различите блокове кôда. За сваки блок који се извршава условно компилатор додељује по један предикат и они су означени на слици Б.4.



Сл. Б.4. Пример предикације.

Ако претпоставимо да су сви предикати иницијализовани на *false* онда је резултујући кôд за IA-64 следећи:

```
(1)          cmp.eq      p1, p2 = 0, a ;;
(2)    (p2)  cmp.eq      p1, p3 = 0, b
(3)    (p3)  add         j = 1, j
(4)    (p1)  cmp.ne      p4, p5 = 0, c
(5)    (p4)  add         k = 1, k
(6)    (p5)  add         k = -1, k
(7)          add         i = 1, i
```

Инструкција (1) пореди садржај симболичког регистра `a` са нулом. Као резултат тог поређења ће се вредност прекидатског регистра p1 поставити на 1 (*true*) а прекидатског регистра p2 на 0 (*false*) ако је релација тачна, односно обрнуто ако је релација нетачна. Инструкција (2) ће се извршити једино ако је предикат p2 са вредношћу *true*. Процесор ће прибавити, декодирати и започети извршење ових инструкција али ће одлуку о томе да ли да прихвати резултате ових инструкција донети тек пошто се одреди да ли је вредност предикатског регистра p1 једнака 1 или 0. Приметимо да је инструкција (2) инструкција генерисана предикатом и да је и сама предикатска. Ова инструкција захтева три поља за предикатске регистре у свом формату.

Ако поново погледамо на наш Pentium програм, прва два условна гранања у асемблерском кôду за Pentium се преводе у две предикатске инструкције поређења за IA-64. Ако инструкција (1) постави p2 на *false*, инструкција (2) се неће извршити. После инструкције (2) у програму за IA-64, p3 је једино тачно ако је спољни **if** исказ у изворном кôду тачан, тј. предикат p3 је тачан само ако је израз a AND b тачан (односно, a ≠ 0 AND b ≠ 0). **Then** грана спољног **if** исказа повезана је са предикатом у p3 из тог разлога. Инструкција (4) у програму за IA-64 одлучује да ли се у спољној **else** грани извршава инструкција сабирања или инструкција одузимања. Најзад, инкрементирање i се извршава безусловно. Ако посматрамо изворни кôд а потом и предикатски кôд, видећемо да ће се само једна од инструкција (3), (5) и (6) извршити. Код обичног суперскаларног процесора би морали да користимо предвиђање гранања да би, евентуално, погодили која ће се од ове три инструкције извршити и продужили том граном тока. У случају погрешне процене проточни систем би морао да се испразни. Процесор IA-64 може да започне извршење све три инструкције а када се одреде вредности у предикатским регистрима да прихвати резултате проистекле из извршења валидне инструкције.

Б.3.4. Управљачка спекулација

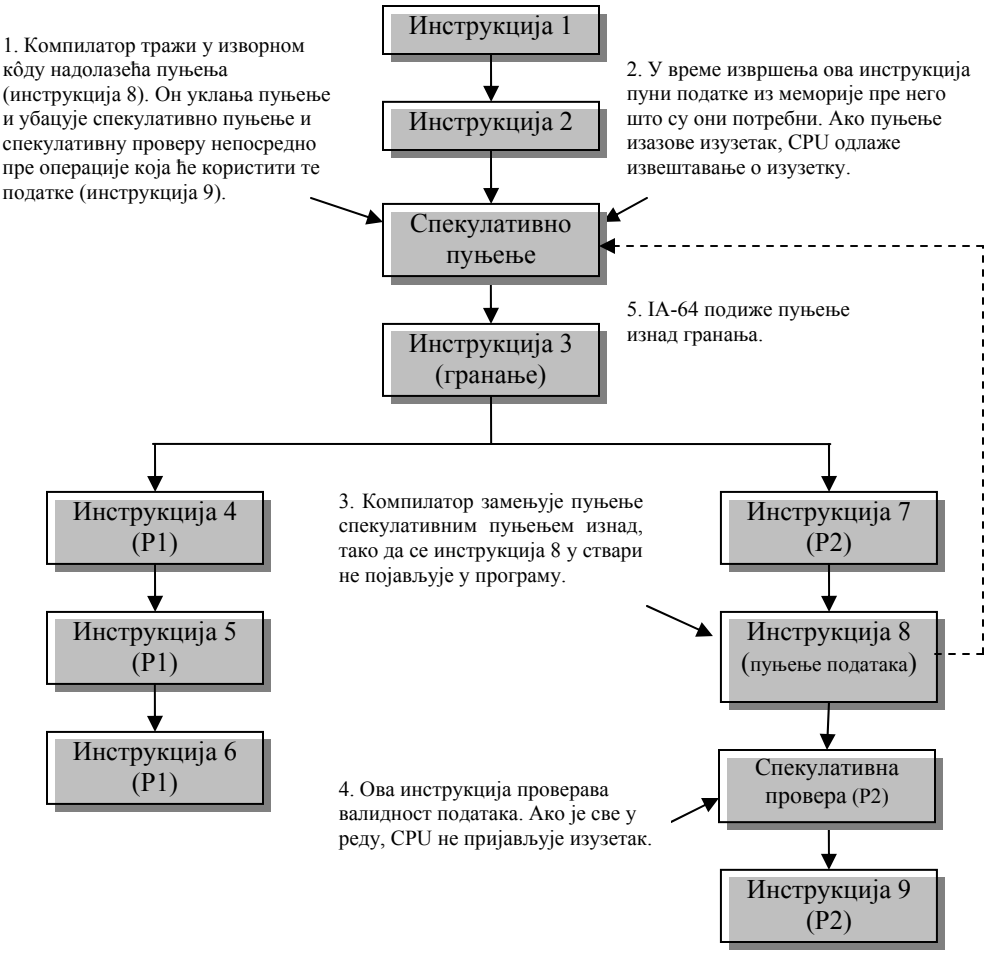
Још једна кључна иновација код IA-64 је управљачка спекулација, такође позната и као спекулативно пуњење. Ова особина омогућује процесору да напуни податке из меморије пре него што су они потребни програму како би се избегла каснија кашњења. Такође, процесор одлаже извештавање о изузецима док то заиста не постане неопходно. Појам *дизања* се користи да означи померање инструкције пуњења у ранију тачку тока инструкција.

Минимизација кашњења услед пуњења података из меморије је кључна за побољшање перформанси. Обичну у неком кôдном блоку постоје инструкције пуњења које податке из меморије копирају у регистре. Како је меморија, чак и она опремљена са једним или два нивоа кеша, много спорија од процесора, пуњење података из меморије представља уско грло у систему. Да би се овај проблем минимизирао потребно је преуредити кôд тако да се инструкције пуњења јаве што је пре могуће. Ово, до извесне мере, може да обави компилатор. Проблем се јавља када покушамо да померимо инструкције пуњења преко неке управљачке структуре. Не може се безусловно померити пуњење изнад инструкције гранања, јер се то пуњење можда неће стварно одиграти. Инструкција пуњења се може померити условно, помоћу предиката, тако да се подаци могу прибавити из меморије али не и заиста напунити у неки од регистара док не буде позната вредност предиката. Могуће је алтернативно користити технике предвиђања гранања, али је проблем у томе што се може јавити изузетак услед инвалидне адресе или промашаја странице, што ће натерати процесор да се бави овим изузетком и изазвати кашњење.

Решење које користи IA-64 је управљачка спекулација која раздваја понашање инструкције пуњења која допрема неку вредност од понашања изузетака (слика Б.5). Свака инструкција пуњења у оригиналном програму замењује се са следеће две инструкције:

- Спекулативно пуњење (ld.s) извршава прибављање из меморије, откривање изузетка, али не позива руковаоце изузетака. Инструкције ld.s подиже се до одговарајуће тачке према почетку програма.
- Инструкција провере (chk.s) остаје на месту оригиналне инструкције пуњења и по потреби позива руковаоца изузецима. Ова инструкција може бити предикатска тако да ће се извршити само ако је вредност предиката *true*.

Ако инструкција ld.s открије изузетак она поставља бит придружен циљном регистру који се назива *Not a Thing* (NaT). Ако се одговарајућа инструкција chk.s извршава а овај бит је постављен, онда се врши гранање на одговарајућу рутину за руковање изузетком.



Сл. Б.5. Спекулативно пуњење.

Размотримо следећи једноставан пример. Оригинални програм је:

```
(p1)     br    neka_labela       // ciklus 0
         ld8   r1 = [r5] ;;      // ciklus 1
         add   r2 = r1, r3       // ciklus 3
```

Прва инструкција врши гранање ако је вредност у регистру p1 једнака 1. Запазимо да су гранање и инструкција пуњења у истој групи инструкција, иако се пуњење неће извршити ако се гранање одигра. IA-64 гарантује да, ако се гранање одигра, касније инструкције, чак и оне из исте групе, неће се извршити. Имплементације IA-64 могу да користе предвиђање гранања ради побољшања ефикасности али морају да се обезбеде од некоректних резултата. Најзад, запазимо да се инструкција сабирања одлаже за најмање један тактни период због меморијских кашњења изазваних операцијом пуњења.

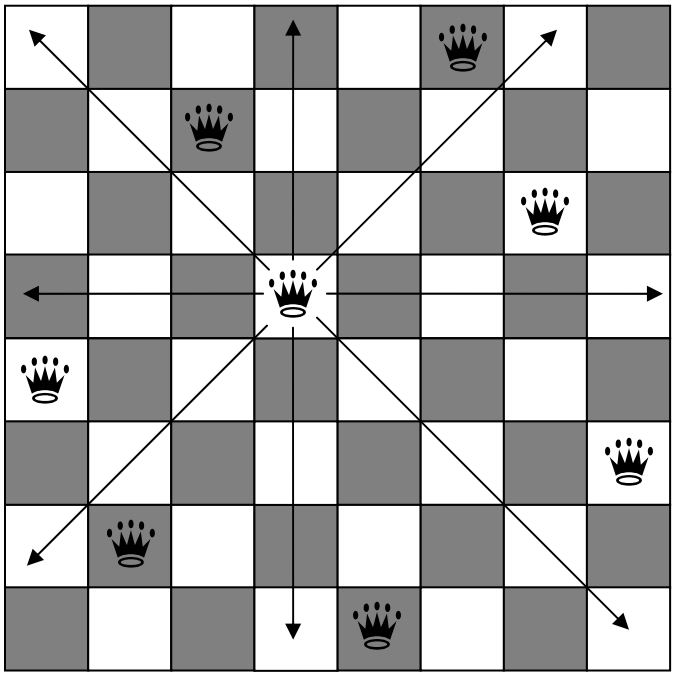
Компилатор може да преправи овај кôд користећи управљачко спекулативно пуњење и спекулативну проверу:

```
         ld8.s r1 = [r5] ;;      // ciklus -2
                                 // druge instrukcije
(p1)     br    neka_labela       // ciklus 0
         chk.s r1, recovery      // ciklus 0
         add   r2 = r1, r3       // ciklus 0
```

Не можемо да тек тако померимо инструкцију пуњења изнад инструкције гранања, јер инструкција гранања може да изазове изузетак (нпр., r5 може да садржи нулу). Уместо тога, претворићемо инструкцију пуњења у инструкцију спекулативног пуњења ld8.s и онда је померити. Инструкција спекулативног пуњења неће одмах да сигнализира изузетак ако га открије; једноставно ће записати ту чињеницу постављајући NaT бит за циљни регистар (у овом случају је то r1). Спекулативно пуњење се сада извршава безусловно бар два циклуса пре гранања. Инструкција chk.s тада проверава садржај бита NaT и ако он није постављен извршава се наредна инструкција. Међутим, ако је тај бит постављен, управљање се преноси на програм који је руковаоц изузетком. Запазимо да се инструкције гранања, провере и сабирања извршавају у истом циклусу. Хардвер обезбеђује да резултат спекулативног пуњења неће да промени стање апликације (тј. промени садржаје регистара r1 и r2) док се не испуне два услова: гранање се не одиграва (p1 = 0) и провера није детектовала одложени изузетак (r1.NaT = 0).

Треба запазити још једну важну чињеницу. Ако нема изузетка, онда је спекулативно пуњење у ствари стварно пуњење и дешаве се пре гранања које би требало да му претходи. Ако се гранање одиграло, тада се јавило пуњење које, према оригиналном програму, није требало да се јави. Наш пример подразумева да се садржај регистра r1 не чита на путу који одговара ситуацији када нема гранања. Међутим, ако би то био случај, компилатор би морао да искористи још један регистар за памћење резултата спекулативног пуњења.

Размотримо сада нешто сложенији пример који се користи како за илустрацију спекулативног пуњења, тако и као бенчмарк програм за предикатске програме. Ради се о познатом проблему осам краљица. Циљ је распоредити осам краљица на шаховској табли тако да ниједна не напада неку другу краљицу. Слика Б.6 илуструје једно од решења.



Сл. Б.6. Једно решење проблема осам краљица.

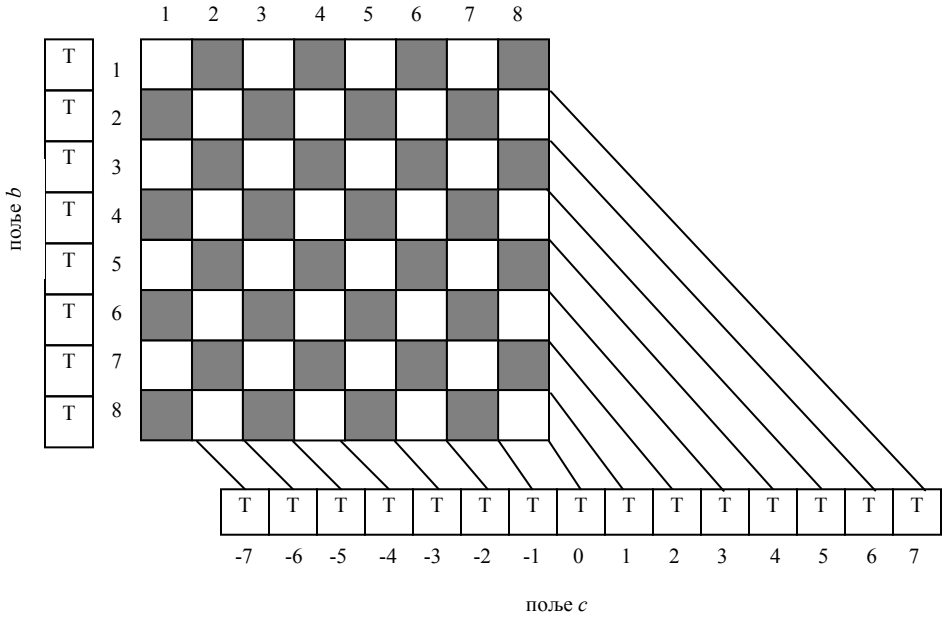
Кључна линија изворног кôда, у унутрашњој петљи, је следећа:

```
if ((b[j] == true) && (a[i+j] == true) && (c[i-j] == true))
```

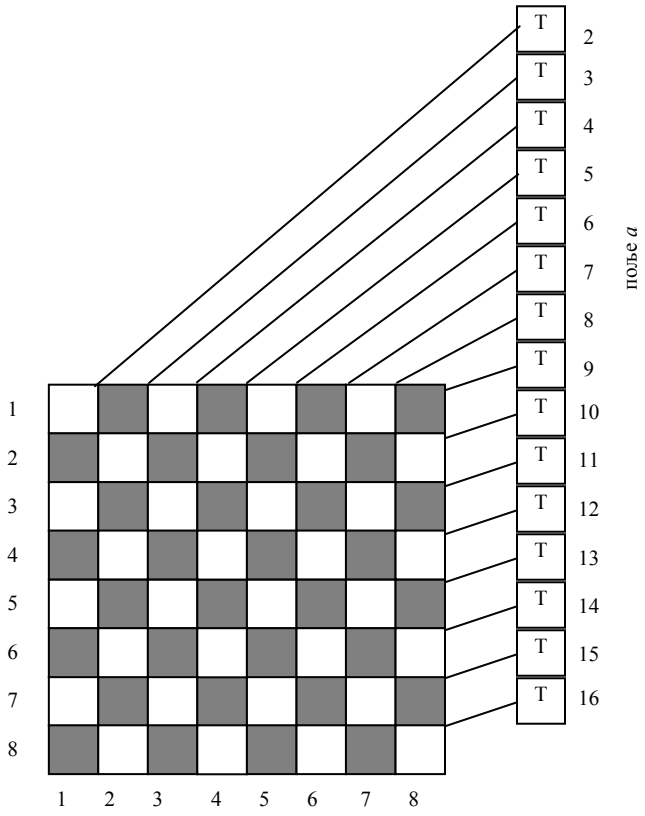
где $1 \leq i, j \leq 8$.

Механизам за праћење конфликата међу краљицама састоји се од три логичка поља која памте стање краљице за сваку врсту и дијагоналу. Вредност TRUE значи да нема ниједне краљице у

дотичној врсти или дијагонали; FALSE значи да постоји већ нека краљица на дотичној врсти односно дијагонали. Слика Б.7 показује како се ова поља пресликавају на шаховску таблу.



а) Поља *b* и *c*.



б) Поље *a*.

Сл. Б.7. Поља *a*, *b* и *c*.

Сви елементи поља су иницијализовани на TRUE. Елементи поља *b* са индексима 1-8 одговарају врстама 1-8 на табли. Краљица у врсти *n* поставља *b*[*n*] на FALSE. Елементи поља *c* имају индексе од -7 до 7 и одговарају разлици између бројева колона и врста и представљају дијагонале одозго на доле и на десно. Краљица у колони 1 и врсти 1 поставља *c*[0] на FALSE. Краљица у колони 1 и врсти 8 поставља *c*[-7] на FALSE. Елементи поља *a* су индексирани од 2 до 16 и одговарају суми бројева колона и врста. Краљица у колони 1 и врсти 1 поставља *a*[2] на FALSE. Краљица у колони 3 и врсти 5 поставља *a*[8] на FALSE.

Општи програм се креће кроз колоне, смештајући краљице по једну у сваку колону тако да нова краљица не напада ни на једну претходно смештену краљицу било дуж врсте или дуж неке од две дијагонале.

Асемблерски програм за Pentium садржи три пуњења и три гранања:

```
(1)             mov   r2, &b[j]   ; kopira sadrzaj lokacije b[j] u r2
(2)             cmp   r2, 1
(3)             jne   L2
(4)             mov   r4, &a[i+j]
(5)             cmp   r4, 1
(6)             jne   L2
(7)             mov   r6, &c[i-j]
(8)             cmp   r6, 1
(9)             jne   L2
(10)      L1:   <kôd za then granu>
(11)      L2:   <kôd za else granu>
```

У горњем програму нотација &x означава адресу локације x. Коришћењем спекулативних пуњења и предикатског извршења добијамо следеће:

```
(1)           mov          r1 = &b[j]  // kopira adresu od b[j] u r1
(2)           mov          r3 = &a[i+j]
(3)           mov          r5 = &c[i-j+7]
(4)           ld8          r2 = [r1]   // indirektno punjenje preko r1
(5)           ld8.s        r4 = [r3]
(6)           ld8.s        r6 = [r5]
(7)           cmp.eq       p1, p2 = 1, r2
(8)    (p2)   br           L2
(9)           chk.s        r4, recovery_a    // sredi punjenje a
(10)          cmp.eq       p3, p4 = 1, r4
(11)   (p4)   br           L2
(12)          chk.s        r6, recovery_b    //sredi punjenje b
(13)          cmp.eq       p5, p6 = 1,r5
(14)   (p6)   br           L2
(15)   L1:    <kôd za then granu>
(16)   L2:    <kôd za else granu>
```

Асемблерски програм се може поделити у три основна кôдна блока где је сваки од њих пуњење праћено условним гранањем. Инструкције 4 и 7 које постављају адресе у асемблерском кôду за Pentium су једноставна аритметичка израчунавања и могу се извршити у било које време, тако да их је компилатор померио на врх. Потом је компилатор суочен са три једноставна блока од којих се сваки састоји од пуњења, израчунавања услова и условног гранања. Изгледа да нема много наде да би нешто од овога могло да се извршава паралелно. Даље, ако претпоставимо да пуњење захтева два или више циклуса, постоји извесно изгубљено време пре него што је могуће извршити условано гранање. Оно што ће компилатор урадити јесте подизање другог и трећег пуњења (инструкције 5 и 8 у програму за Pentium) изнад гранања. Ово је урађено уметањем спекулативног пуњења на врх (инструкције 5 и 6 у IA-64 кôду) и остављање провера у оригиналном блоку кôда (инструкције 9 и 12 у IA-64 кôду).

Ова трансформација омогућава извршење сва три пуњења у паралели и отпочињање пуњења раније тако да се минимизира или чак у потпуности избегава кашњење услед пуњења. Компилатор може да иде и даље агресивније користећи предикацију и елиминишући два од три гранања.

```
(1)          mov          r1 = &b[j]
(2)          mov          r3 = &a[i+j]
(3)          mov          r5 = &c[i-j+7]
(4)          ld8          r2 = [r1]
(5)          ld8.s        r4 = [r3]
(6)          ld8.s        r6 = [r5]
(7)          cmp.eq       p1, p2 = 1, r2
(8)    (p1)  chk.s        r4, recovery_a
(9)    (p2)  cmp.eq       p1, p2 = 1, r4
(10)   (p3)  chk.s        r6, recovery_b
(11)   (p3)  cmp.eq       p5, p4 = 1,r5
(12)   (p6)  br           L2
(13)   L1:   <kôd za then granu>
(14)   L2:   <kôd za else granu>
```

Већ смо имали поређење које генерише два предиката. У последњој верзији кôда, уместо гранања у случају нетачног предиката, компилатор квалификује извршење обе провере и следеће поређење у случају тачног предиката. Елиминација два гранања значи елиминацију два потенцијална погрешна предвиђања, тако да је уштеда и већа од две инструкције.

Б.3.5. Спекулација подацима

Код управљачке спекулације пуњења се померају у раније делове кôда да би се компензовало кашњење а провере осигуравају да се изузетак не јавља ако пуњење није успело. Код спекулације подацима пуњење се помера пре инструкције уписа у меморију која може да промени садржај меморијске локације која је извор за пуњење. Потом се врши провера која ће осигурати да је напуњења исправна вредност из меморије. Да би разјаснили овај механизам размотримо следећи програмски сегмент:

```
st8      [r4] = r12         // ciklus 0
ld8      r6 = [r8] ;;       // ciklus 0
add      r5 = r6, r7 ;;     // ciklus 2
st8      [r18] = r5         // ciklus 3
```

Овако како је написано кôд захтева четири циклуса за извршење. Ако регистри r4 и r8 не садрже исту меморијску адресу, онда смештање у меморију преко регистра r4 не утиче на вредност која се налази на адреси која је у регистру r8. Под овим условом, безбедно је преуредити инструкције пуњења и смештања да би се брже допремила вредност у r6 која је неопходна за наредну инструкцију. Међутим, адресе у r4 и r8 могу да буду исте или да се односе на преклопљене делове меморије, па у општем случају оваква замена места инструкцијама није безбедна. IA-64 превазилази овај проблем помоћу технике *напредног пуњења*.

```
ld8.a r6 = [r8] ;;       // ciklus -2 ili raniji; napredno punjenje
                         // druge instrukcije
st8   [r4] = r12         // ciklus 0
ld8.c r6 = [r8]          // ciklus 0; provera punjenja
add   r5 = r6, r7 ;;     // ciklus 0
st8   [r18] = r5         // ciklus 1
```

Овде смо померили инструкцију ld унапред и претворили је у напредно пуњење. Уз обављање наведеног пуњења, инструкција ld8.a уписује своју изворну адресу (адресу која је у регистру r8) у хардверску структуру података која се назива табела адреса напредног пуњења (***A**dvanced **L**oad **A**ddress **T**able* – ALAT). Свака инструкција смештања у меморију код IA-64 проверава да ли ALAT садржи ставке које се преклапају са њеном циљном адресом. Ако се такво поклапање нађе, ставка се уклања из структуре података. Када се оригинална инструкција ld8 претвори у инструкцију ld8.a и помери, на оригиналној позицији те инструкције се убацује инструкција провере пуњења ld8.c. Када се провера пуњења извршава онда та инструкција проверава да ли ALAT садржи адресе локација које се поклапају. Ако се таква адреса нађе, ниједна инструкција смештања у меморију између напредног пуњења и провере пуњења није изменила изворну адресу пуњења и никаква акција се не

предузима. Али, ако инструкција провере пуњења не пронађе одговарајућу ставку у структури ALAT, инструкција пуњења се извршава поново да би се осигурао коректан резултат.

Можемо, такође, и да спекулативно извршавамо инструкције које су зависне по подацима са инструкцијом пуњења, заједно са самим пуњењем. Почевши од истог оригиналног програма, претпоставимо да смо померили нагоре и инструкцију пуњења и инструкцију сабирања:

```
          ld8.a r6 = [r8] ;;// ciklus -3 ili raniji; napredno
                            // punjenje
                            // druge instrukcije
          add   r5 = r6, r7 // ciklus -1; sabiranje koje koristi r6
                            // druge instrukcije
          st8   [r4] = r12  // ciklus 0
          chk.a r6, recover // ciklus 0; provera
   back:                    // tacka povratka posle skoka na recover
          st8   [r18] = r5  // ciklus 0
```

Овде смо користили инструкцију chk.a уместо ld8.c да би проценили исправност напредног пуњења. Ако инструкција chk.a пронађе да пуњење није извршено, онда се не врши само поновно извршење инструкције пуњења. Уместо тога врши се гранање на рутину за обраду тог случаја:

```
recover: ld8   r6 = [r8];;       // ponovno punjenje r6 iz [r8]
         add   r5 = r6, r7 ;;    // ponovno izvrsenje sabiranja
         br    back              // skok nazad na glavni kod
```

Ова техника је ефикасана само ако постоје мале шансе за преклапање локација које користе инструкције пуњења и смештања.

Б.3.6. Софтверска проточност

Размотримо следећи петљу:

```
L1:   ld4   r4 = [r5], 4;;// ciklus 0; punjenje i postinkrement za 4
      add   r7 = r4, r9 ;;// ciklus 2
      st4   [r6] = r7, 4  // ciklus 3; smestanje i postinkrement za 4
      br.cloop    L1 ;;   // ciklus 3
```

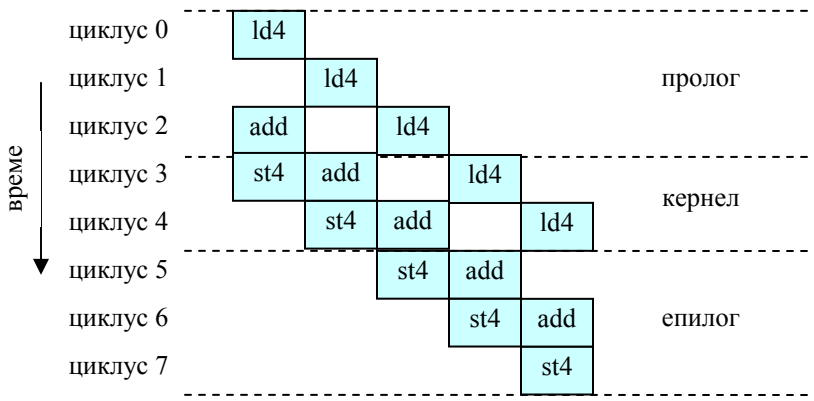
Ова петља додаје константу вектору и памти резултат у други вектор ($y[i] = x[i] + c$). Инструкција ld4 пуни 4 бајта из меморије. Квалификатор ", 4" на крају инструкције указује да је то измењени облик основне инструкције; адреса у r5 се инкрементира за 4 после извршења пуњења. Слично томе, инструкција st4 смешта четири бајта у меморију а адреса у r6 се инкрементира за 4 после тога. Инструкција br.cloop се назива и гранање бројачке петље и користи LC (Loop Count) регистар. Ако је садржај регистра LC већи од 0, његов садржај се декрементира и потом врши гранање. Почетна вредност овог регистра је број итерација петље.

У овом програму привидно нема могућности за паралелизам на нивоу инструкција у оквиру петље. Осим тога, инструкције у итерацији n се све извршавају пре него што почне итерација n+1. Међутим, ако нема адресних конфликата између *load* и *store* инструкција (тј. r5 и r6 указују на непреклопљене меморијске локације), онда се ефикасност повећава померањем независних инструкција из итерације n+1 у итерацију n. Још један начи да се ово каже је да ако одмотамо кôд петље тако што ћемо исписати нови скуп инструкција за сваку итерацију, онда постоји могућност да се увећа паралелизам. Погледајмо шта се може урадити са пет итерација:

```
      ld4   r32 = [r5], 4 ;;  // ciklus 0
      ld4   r33 = [r5], 4 ;;  // ciklus 1
      ld4   r34 = [r5], 4     // ciklus 2
      add   r36 = r32, r9 ;;  // ciklus 2
      ld4   r35 = [r5], 4     // ciklus 3
      add   r37 = r33, r9     // ciklus 3
      st4   [r6] = r36, 4 ;;  // ciklus 3
      ld4   r36 = [r5], 4     // ciklus 3
      add   r38 = r34, r9     // ciklus 4
```

```
st4   [r6] = r37, 4 ;;  // ciklus 4
add   r39 = r35, r9     // ciklus 5
st4   [r6] = r38, 4 ;;  // ciklus 5
add   r40 = r36, r9     // ciklus 6
st4   [r6] = r39, 4 ;;  // ciklus 6
st4   [r6] = r40, 4 ;;  // ciklus 7
```

Овај програм ће завршити пет итерација у седам циклуса, што је много боље у поређењу са 20 циклуса у оригиналном програму. Овде се подразумева да постоје два меморијска порта тако да се *load* и *store* могу извршавати паралелно. Ово је пример софтверске проточности. Слика Б.8 илуструје овај процес.



Сл. Б.8. Пример софтверске проточности.

Паралелизам се постиже груписањем инструкција из различитих итерација. Да би ово функционисало привремени регистри који су коришћени у оквиру тела петље морају се изменити за сваку итерацију да би се избегао регистарски конфликт. У овом случају користе се два привремена регистра (r4 и r7 у оригиналном програму). У развијеном програму бројеви регистара расту за сваку итерацију и морају се одабрати да буду довољно далеко како не би дошло до преклапања.

На слици Б.8 видимо да софтверски проточни систем има три фазе. Током **пролог фазе** иницира се нова итерација у сваком циклусу и проточни систем се постепено пуни. За време **кернел фазе** проточни систем је пун па је достигнут максимални ниво паралелизма. У нашем примеру се три инструкције извршавају паралелно у току ове фазе али је ширина проточног система једнака четири. Током **епилог фазе** се у сваком циклусу комплетира по једна итерација.

Софтверска проточност помоћу одмотавања петље оставља компилатору или програмеру терет да на исправан начин изврше доделу регистара. Осим тога, за дугачке петље са много итерација, одмотавање резултује значајним повећањем величине кôда. За петље чији број итерација није познат у време компилације овај посао се компликује потребом да се изврши делимично одмотавање и управљање бројем петљи. IA-64 предвиђа хардверску подршку софтверској проточности уз помоћ које нема проширења кôда а минималан терет посла је на компилатору. Кључна средства која подржавају софтверску проточност су следећа:

- **Аутоматско преименовање регистара:** Део регистарских поља за предикатске регистре и регистре у покретном зарезу фиксираних величина (p16 до p63; fr32 до fr127) и област регистара опште намене програмабилне величине (максималног опсега од r32 до r127) се може ротирати. То значи да се током сваке итерације петље са софтверском проточношћу регистарске референце аутоматски инкрементирају. Ако је регистар r32 коришћен у првој итерацији, у наредној се аутоматски користи r33 итд.
- **Предикација:** Свака инструкција у петљи је предикатска на ротирајућем предикатском регистру. Сврха овога је да се одреди када је проточни систем у фази пролога, кернела или епилога.
- **Специјалне инструкције за терминирање петљи:** То су инструкције гранања које изазивају ротацију регистара и смањење бројача петље.

Ради се о релативно сложеном питању. Овде ћемо представити пример који илуструје неке од могућности IA-64 везане за софтверску проточност. Узећемо оригинални програм са петљом који смо и малопре користили и показаћемо како се програмира софтверска проточност под претпоставком да петља има 200 пролаза а да постоје два меморијска порта.

```
            mov   lc = 199     // postavlja registar brojaca petlje na
                               // 199, petlja se vrti dok ova
                               // vrednost ne bude -1
            mov   ec = 4       // postavlja registar brojaca epiloga na
                               // broj stepeni epioga + 1
            mov   pr.rot = 1 << 16  // pr16 = 1; ostali = 0
L1:   (p16) ld4   r32 = [r5], 4     // ciklus 0
      (p17) ---                     // prazan stepen
      (p18) add   r35 = r34, r9     // ciklus 0
      (p19) st4   [r6] = r36, 4     // ciklus 0
            br.ctop     L1 ;;       // ciklus 0
```

Таб. Б.4. Трагови извршења петље у примеру софтверске проточности.

циклус	Извршна једница/Инструкција				Стање пре br.ctop					
	M	I	M	B	P16	P17	P18	P19	LC	EC
0	ld4			br.ctop	1	0	0	0	199	4
1	ld4			br.ctop	1	1	0	0	198	4
2	ld4	add		br.ctop	1	1	1	0	197	4
3	ld4	add	st4	br.ctop	1	1	1	1	196	4
…	…	…	…	…	…	…	…	…	…	…
100	ld4	add	st4	br.ctop	1	1	1	1	99	4
…	…	…	…	…	…	…	…	…	…	…
199	ld4	add	st4	br.ctop	1	1	1	1	0	4
200		add	st4	br.ctop	0	1	1	1	0	3
201		add	st4	br.ctop	0	0	1	1	0	2
202			st4	br.ctop	0	0	0	1	0	1
					0	0	0	0	0	0

Кључне тачке у овом програму су:

1. Тело петље подељено је на више *степени*, са нула или више инструкција по степену.
2. Извршење петље одвија се кроз три фазе. Током пролог фазе се сваки пут започиње нова итерација петље додајући један степен проточном систему. У току кернел фазе се једна итерација петље започиње а једна комплетира сваки пут; проточни систем је пун са максималним бројем активних степени. У епилог фази се не започиње ниједна нова итерација а у сваком циклусу се комплетира по једна итерација.
3. Предикат се додељује сваком степену да би управљао активацијом инструкција у том степену. Током пролог фазе, p16 је *true* а p17, p18 и p19 су *false* током прве итерације. За другу итерацију p16 и p17 су *true*; током треће итерације p16, p17 и p18 су *true*. Током кернел фазе сви предикати су *true*. У епилог фази се предикати један по један враћају на *false* почев од p16. Промене у вредностима предикатских регистара постижу се њиховом ротацијом.
4. Сви регистри опште намене са бројевима већим од 31 се ротирају у свакој итерацији. Ротација се врши ка већим бројевима регистара на цикличан начин. На пример, вредност у регистру *x* смешта се у регистар *x*+1 после једне ротације; ово се не ради копирањем вредности већ хардверским преименовањем регистара. Тако се у нашем примеру вредност коју инструкција *load* уписује у r32 чита од стране инструкције *add* две итерације (и две ротације) касније као r34. Слично, вредност коју инструкција *add* уписује у r35 чита се од стране инструкције *store* после једне итерације као r36.
5. Код инструкције br.ctop гранање ће се десити ако је или LC > 0 или EC > 1. Извршење br.ctop има два ефекта: ако је LC > 0 тада се LC декрементира што се дешава током пролог и кернел фаза. Ако је LC = 0 и EC > 1, декрементира се EC и то је случај у епилог фази. Ова

инструкција такође управља ротацијом регистара. Ако је LC > 0 свако извршење br.ctop смешта 1 у p63. После ротације p63 постаје p16 пунећи непрекидни низ јединица у предикатске регистре током пролог и кернел фаза. Ако је LC = 0, тада br.ctop поставља 0 у p63 пунећи нулама предикатске регистре током епилог фазе.

Табела Б.4 показује трагове извршења за овај пример.

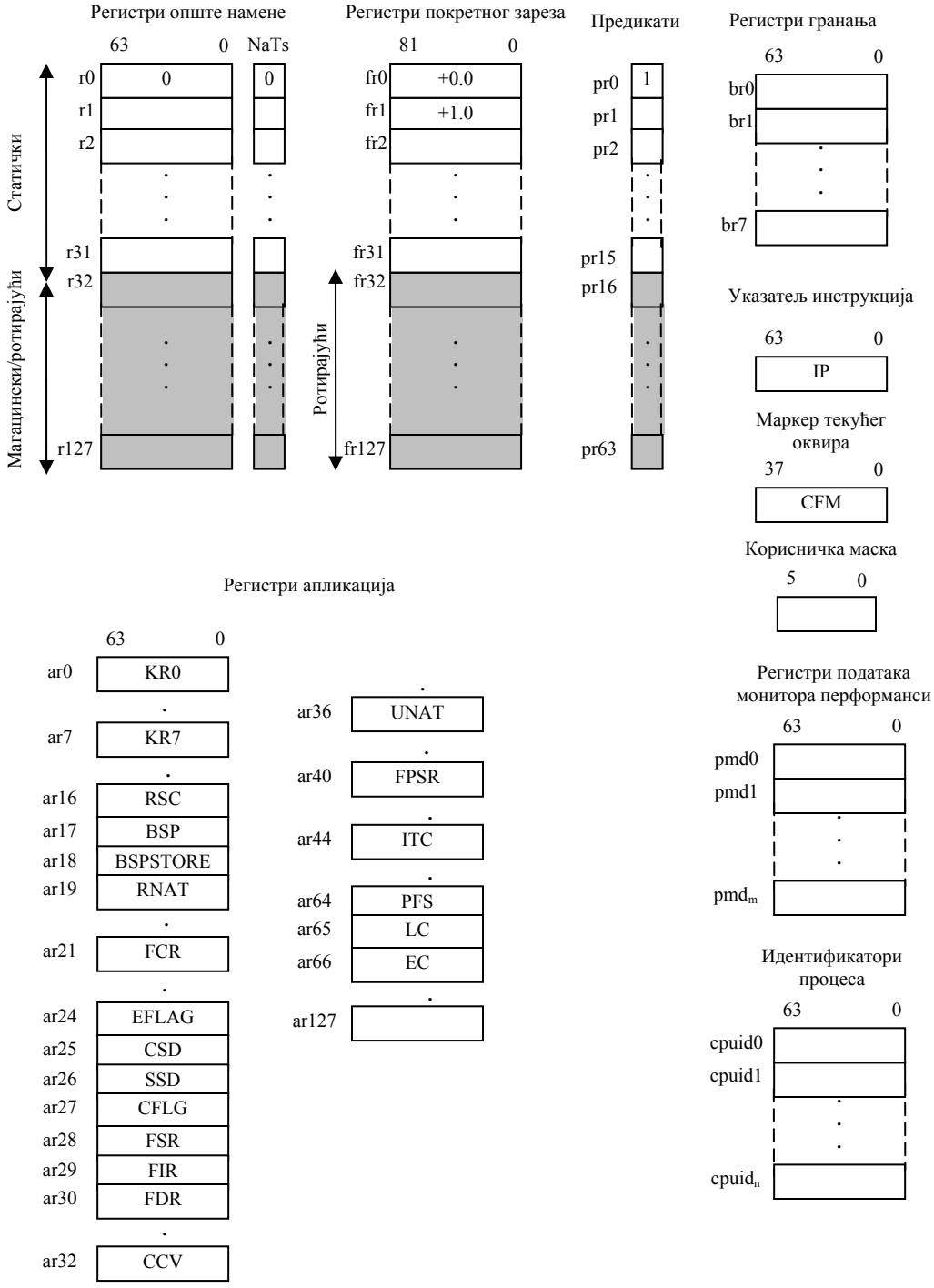
Б.4. Архитектура IA-64

На слици Б.9 приказан је скуп регистара расположив за апликативне програме. Ови регистри су видљиви апликацијама и могу се читати и у већини случајева и уписивати. Скуп регистара обухвата следеће регистре:

- **Регистри опште намене:** 128 64-битних регистара опште намене. Сваком регистру придружен је NaT бит који се користи за праћење одложених спекулативних изузетака, као шго је то раније објашњено. Регистри r0-r31 су статички, што значи да се програмске референце на неки од њих увек дословно интерпретирају. Регистри r32-r127 могу се користити као ротирајући регистри за софтверску проточност и имплементацију регистарског магацина, што ће бити касније објашњено. Референце на ове регистре су виртуелне а хардвер може динамички да врши преименовање регистара.
- **Регистри покретног зареза:** Постоји 128 82-битних регистара за бројеве у покретном зарезу. Ова величина је довољна да прихвати двоструки проширени формат према стандарду IEEE 754. Регистри fr0-fr31 су статички а регистри fr32-fr127 могу да се користе као ротирајући регистри код софтверске проточности.
- **Предикатски регистри:** 64 једнобитна регистра користе се као предикати. Регистар pr0 је увек постављен на 1 да би се омогућиле непредикатске инструкције. Регистри pr0-pr15 су статички док се регистри pr16-pr63 могу користити као ротирајући регистри код софтверске проточности.
- **Регистри гранања:** Постоји 8 64-битних регистара који се користе за гранања.
- **Указатељ инструкција:** Садржи групну адресу за 64-битну инструкцију која се тренутно извршава.
- **Маркер текућег оквира:** Садржи статусну информацију која се односи на текући оквир магацина регистара опште намене као и информације о ротирању за fr и pr регистре.
- **Корисничка маска:** Скуп једнобитних вредности који се користи за трапове поравнања, мониторе перформанси и монитор употребе регистара за покретни зарез.
- **Регистри података монитора перформанси:** Користе се за подршку хардверу монитора перформанси.
- **Идентификатори процесора:** Описују особине процесора које зависе од имплементације.
- **Апликациони регистри:** Колекција регистара специјалне намене. У табели Б.5 дате су кратке дефиницјије за сваки од њих.

Б.4.1. Регистарски магацин

Механизам регистарског магацина код IA-64 служи за избегавање непотребних копирања података у и из регистара приликом позива процедура и повратака из њих. Овај механизам аутоматски позваној процедури придружује нови оквир који се састоји од највише 96 регистара (r32-r127) приликом отпочињања процедуре. Компилатор специфицира број регистара које захтева процедура помоћу инструкције alloc која специфицира који број међу њима представља локалне регистре а колико њих представља излаз (тј. који преносе параметре процедури позваној од стране дотичне процедуре). Када се јави позив процедуре хардвер IA-64 врши преименовање регистара тако да се локални регистри из претходног оквира скривају а излазни регистри позивајуће процедуре постају сада регистри чији бројеви почињу од r32 у позваној процедури. Физички регистри у опсегу од r32 до r127 додељују се по принципу кружног бафера виртуелним регистрима који су придружени процедури. Према томе, следећи додељени регистар после r127 је r32. Када је то потребно, хардвер копира садржај регистара у меморију и обрнуто како би ослободио додатне регистре приликом позива процедуре, тј. обновио садржај регистара из меморије приликом повратака из процедуре.

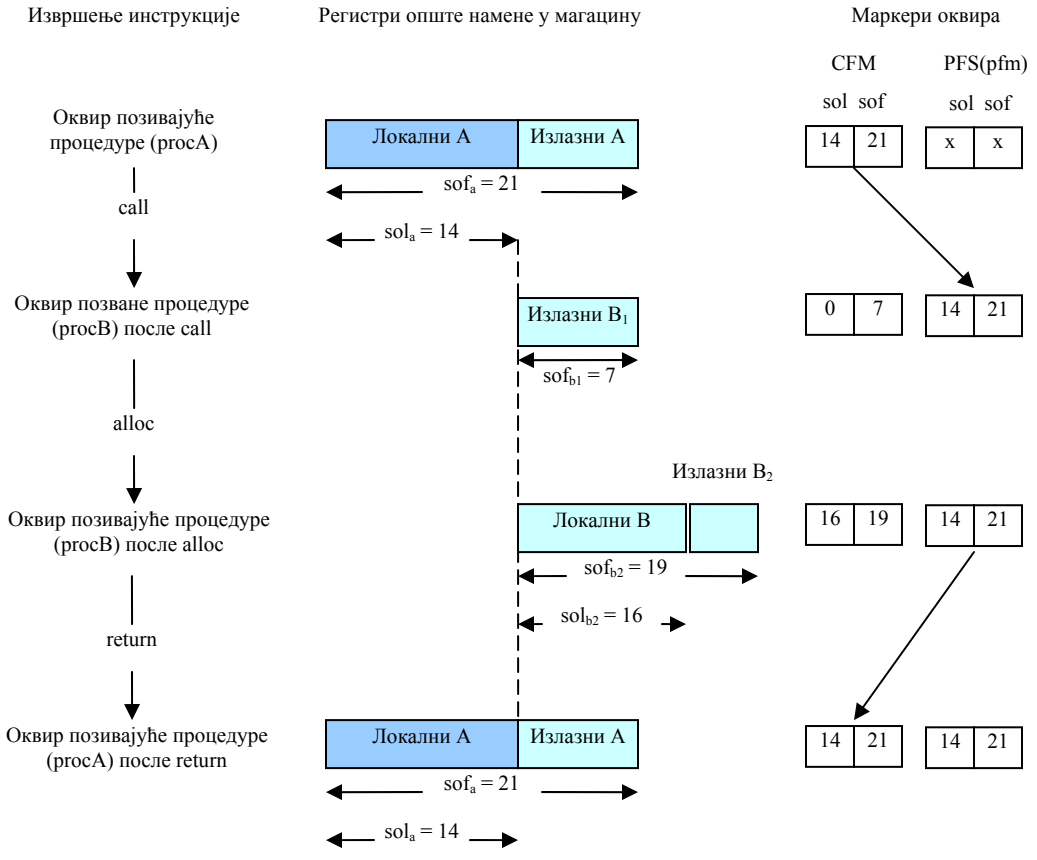


Сл. Б.9. Регистри код IA-64.

На слици Б.10 приказано је понашање регистарског магацина. Инструкција alloc има за операнде sof (*size of frame*) и sol (*size of locals*) који специфицирају потребан број регистара. Ове вредности смештене су у регистар CFM. Када се јави позив, вредности sof и sol из CFM се смештају у sof и sol поља апликационог регистра стања претходне функције (PFS) (види слику Б.11).

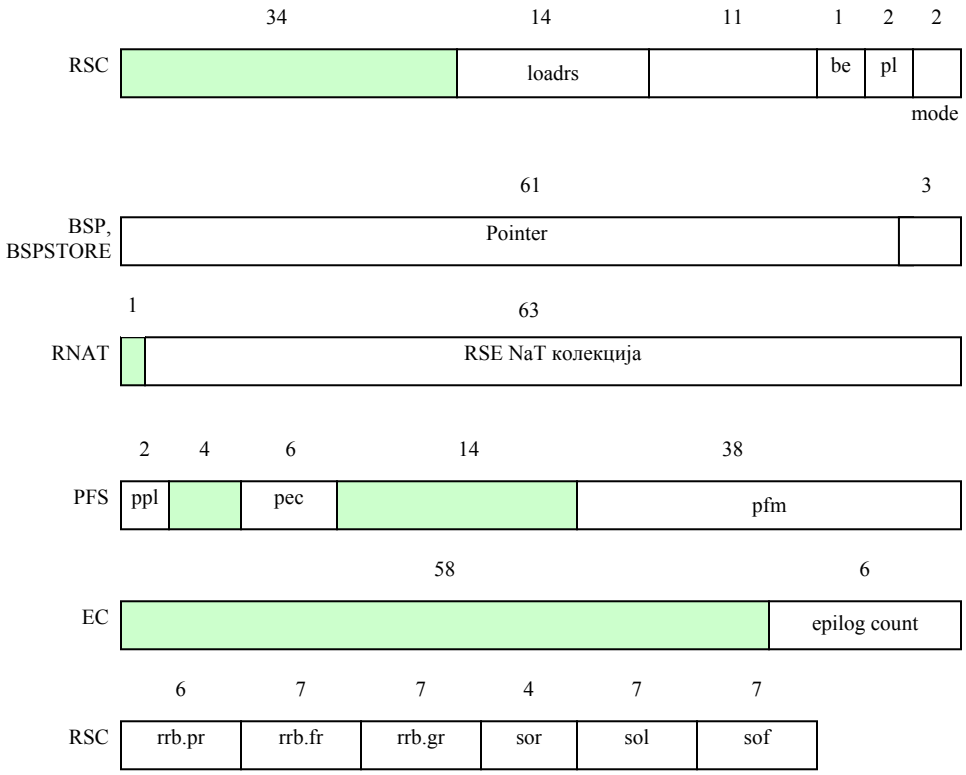
Таб. Б.5. Апликациони регистри код IA-64.

Кернел регистри (KR0-7)	Преноси информације од ОС-а ка апликацији.
Конфигурација регистарског магацина (RSC)	Управља радом механизма регистарског магацина (RSE)
RSE *Backing store* указатељ (BSP)	Садржи меморијску адресу која је локација за памћење r32 у текућем оквиру магацина.
RSE *Backing store* указатељ на меморијске локације (BSPSTORE)	Садржи меморијску адресу где ће RSE прелити следећу вредност.
Регистар колекцијеRSE NaT (RNAT)	Користи га RSE за привремено чување NaT битова када се преливају регистри опште намене.
Вредности за поређење и размену (CCV)	Садржи вредност за поређење која се користи као трећи изворни операнд у инструкцији cmpxchg.
Кориснички регистар NaT колекције (UNAT)	Користи се за привремено чување NaT битова код памћења и обнављања регистара опште намене код инструкција ld8.fill и st8.spill.
Регистар стања покретног зареза (FPSR)	Управља траповима, начином заокруживања, прецизношћу, маркерима и другим управљачким битовима за инструкције у покретном зарезу.
Бројач интервала времена (ITC)	Врши одбројавање на основу фиксираног односа са фреквенцијом тактног сигнала процесора.
Стање претходне функције (PFS)	Памти вредност у регистар CFM и друге сродне информације.
Бројач петље (LC)	Користи се у бројачким петљама и декрементира се код гранања типа бројачких петљи.
Бројач епилога (EC)	Користи се за бројање завршних стања (епилога) у петљама које се планирају по модулу.



Сл. Б.10. Понашање регистарског магацина приликом позива и повратка.

По повратку sol и sof вредности се морају обновити из PFS у CFM. Да би се омогућили угњежђени позиви и повратци, претходне вредности PFS поља се морају сачувати током узастопних позива тако да се могу обновити током сукцесивних повратака. Ово је функција инструкције alloc која назначава који се регистар опште намене користи за памћење текуће вредности PFS поља пре него што се она препишу од стране CFM поља.



Сл. Б.11. Формати неких регистара код IA-64.

Б.4.2. Маркер текућег оквира и стање претходне функције

Регистар CFM описује стање текућег оквира магацина регистара опште намене који је придружен тренутно активној процедури. Он обухвата следећа поља:

- **sof:** Величина оквира магацина.
- **sol:** Величина локалног дела оквира магацина.
- **sor:** Величина ротирајућег дела оквира магацина; представља подскуп локалног дела који је посвећен софтверској проточности.
- **Вредности базе за преименовање регистара:** Вредности које се користе код обављања ротирања регистара опште намене, регстара за покретни зарез и предикатских регистара.

Апликациони регистар PFS садржи следећа поља:

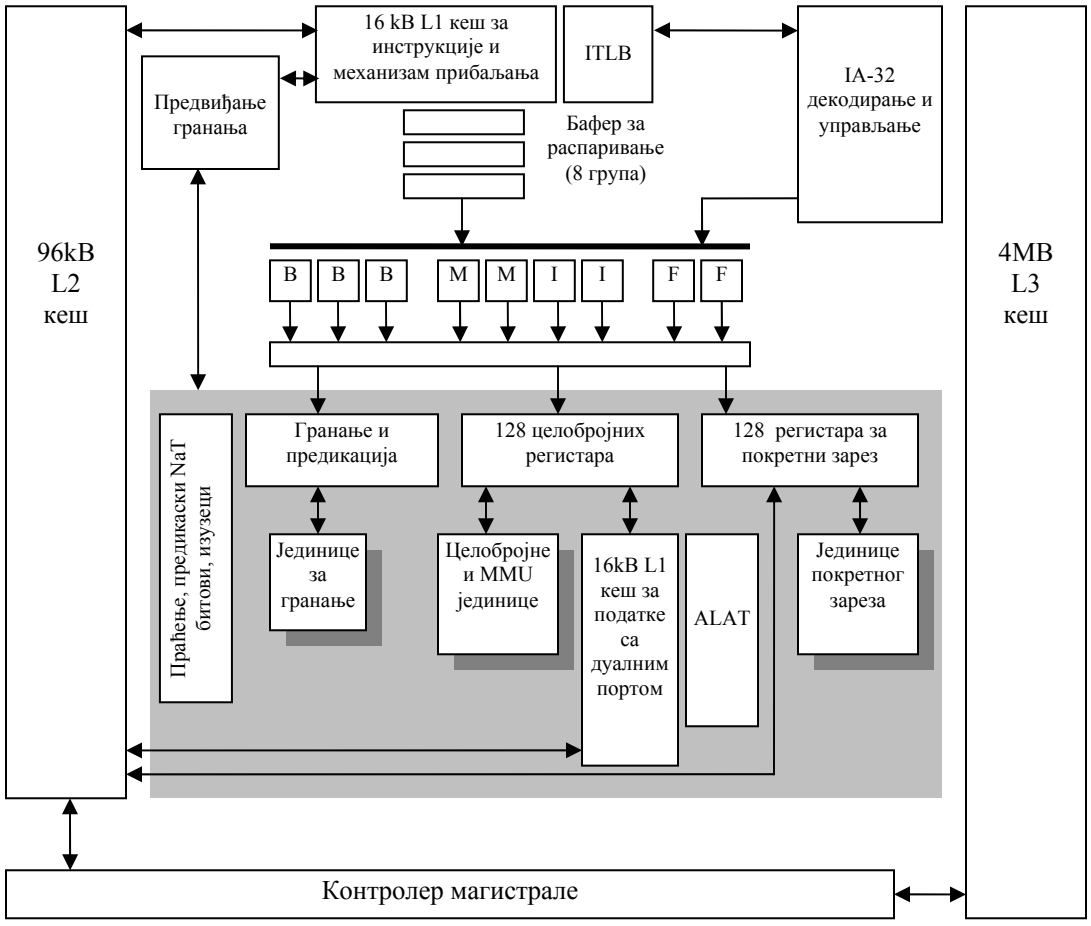
- **pfm:** Маркер претходног оквира; садржи сва поља регистра CFM.
- **pec:** Претходни број епилога.
- **ppl:** Претходни ниво привилегије.

Б.5. Организација Itanium-а

Intel-ов процесор Itanium је прва имплементација архитектуре IA-64. Организација Itanium-а сједињује суперскаларне карактеристике са подршком јединственим карактеристикама архитектуре

IA-64 које се односе на EPIC. Међу суперскаларним карактеристикама су шестоструки хардверски проточни систем дубине 10, динамичко прибављање унапред, предвиђање гранања и праћење регистара ради оптимизације онога што није било одређено у време копилације. Хардвер који се односи на EPIC обухвата подршку предикатском извршењу, управљачкој спекулацији и спекулацији података, као и софтверској проточности.

Слика Б.12 представља општи блок дијаграм организације Itanium-a. Itanium садржи 9 извршних јединица: две целобројне, две за покретни зарез, две меморијске и три за гранање. Инструкције се прибављају преко L1 кеша за инструкције и пуне у бафер који може да садржи до 8 група инструкција. Када одлучује о распоређивању инструкција у функционалне јединице, процесор сагледава највише две групе инструкција истовремено. Процесор може да изда највише шест инструкција по једном тактном периоду.



Сл. Б.12. Организација процесора Itanium.

Организација је на известан начин једноставнија од конвенционалних савремених суперскаларних организација. Itanium не користи станице за резервацију, бафере за промену редоследа, бафере за уређење меморије, јер је све то замењено једноставнијим хардвером за спекулацију. Хардвер за пресликавање регистара је једноставнији од онога за доделу алиаса регистрима код типичних суперскаларних машина. Логика за детекцију зависности међу регистрима је замењена директивама експлицитног паралелизма унапред израчунатим софтверским путем.

Коришћењем предвиђања гранања, механизам за прибављање и прибављање унапред може спекулативно да пуни L1 кеш инструкција како би минимизирао кеш промашаје код прибављања

инструкција. Прибављени кôд се пуни у бафер за распаривање који може да садржи до осам група инструкција.

Користе се три нивоа кеша. L1 кеш је подељен у кеш за инструкције и кеш за податке (сваки капацитета 16kB) који користе четвороструко скупно-асоцијативно пресликавање са величином линије од 32B. L2 кеш је капацитета 96kB и користи шестоструко скупно-асоцијативно пресликавање са линијама дужине 64B. L3 кеш је величине 4MB и користи четвороструко скупно-асоцијативно пресликавање са величином линије од 64B. L1 и L2 кешеви су у чипу процесора, док је L3 ниво *off*-chip, али у истом пакету са процесором.

Литература

[1] W. Stallings, *Computer Organization and Architecture*, 6/e, Prentice Hall, 2003.

Садржај